Сергей Балакирев

Добрый, добрый Python



2026

Сергей Балакирев

ДОБРЫЙ, ДОБРЫЙ PYTHON

2026

Курсы серии «Добрый, добрый» на платформе Stepik
(для отработки практических навыков)

1. Python: https://stepik.org/course/100707
2. Python ООП: https://stepik.org/a/116336
3. Стандартная библиотека Python: https://stepik.org/a/259466
4. Django + Docker: https://stepik.org/a/183363
5. C/C++: https://stepik.org/a/193691
6. ООП C++: https://stepik.org/a/205781
7. Машинное обучение: https://stepik.org/course/209247/
8. Нейронные сети на PyTorch: https://stepik.org/a/227582

Телеграм-канал: https://t.me/selfedu_sb

Телеграм-чат: https://t.me/python_selfedu

Видео-каналы: Rutube | Youtube

Ручная работа. Данная книга не содержит ни одной строчки, сгенерированной с помощью ИИ. Все от начала и до конца написано автором – Балакиревым С.М.

Данное учебное пособие можно свободно распространять в электронном виде в сети Интернет в некоммерческих, образовательных целях.

© Дизайн обложки: Андрей Дубовой,
(Телеграм: DispeoNet1111)

© Автор книги: Сергей Балакирев

Версия издания 1.0

Оглавление

ЧАСТЬ I. ОСНОВЫ ЯЗЫКА PYTHON....8
ГЛАВА I. С ЧЕГО НАЧИНАЕТСЯ PYTHON? 10
1.1. Установка интерпретатора языка Python 10
1.2. Варианты выполнения команд 13
1.2.1. Форматирование текста программ....14
1.2.2. Интегрированная среда PyCharm....15
ГЛАВА II. МОИ ПЕРВЫЕ ШАГИ В PYTHON....22
2.1. Переменные и оператор присваивания....22
2.1.1. Функция type() 25
2.1.2. Правила написания имен переменных 25
2.2. Числа и операции над ними....27
2.2.1. Основные арифметические операции....27
2.2.2. Приоритеты арифметических операций....31
2.2.3. Некоторые дополнительные арифметические операции....31
2.3. Математические функции и модуль math 32
2.3.1. Модуль math....33
2.4. Функции print() и input() 35
2.4.1. Знакомство с функцией print()....35
2.4.2. Знакомство с функцией input()....37
2.5. Логический тип Bool....39
2.5.1. Операции not, and, or и их приоритеты 41
2.5.2. Приведение к типу bool....43
ГЛАВА III. ПОСТИЖЕНИЕ СТРОК И СПИСКОВ....44
3.1. Строки в языке Python....44
3.1.1. Базовые операции над строками 45
3.1.2. Индексы строк 48
3.1.3. Срезы строк....49
3.1.4. Изменение строк....50
3.1.5. Основные методы строк....51
3.1.6. Спецсимволы и экранирование символов....58
3.2. Форматирование строк. F-строки....62
3.2.1. Метод format()....62
3.2.2. F-строки....63
3.2.3. Спецификации форматирования F-строк....65
3.3. Списки и операции над ними 67
3.3.1. Список – изменяемый тип данных....68
3.3.2. Функции работы со списками 69
3.3.3. Базовые операции со списками 70
3.3.4. Срезы списков....72
3.3.5. Сравнение списков 76
3.3.6. Методы списков....76
3.3.6. Вложенные списки 81
ГЛАВА IV. УСЛОВНЫЙ ОПЕРАТОР if. КОНСТРУКЦИЯ if-elif-else....86
4.1. Условный оператор if....86
4.2. Конструкция if-else условного оператора if....88
4.3. Вложенные условия. Конструкция if-elif-else....89
4.3.1. Оператор elif....91
4.4. Тернарная условная операция....93
4.5. Вложенная тернарная операция....95
ГЛАВА V. ЦИКЛЫ, ИТЕРАТОРЫ И ГЕНЕРАТОРЫ СПИСКОВ....96
5.1. Оператор цикла while....96
5.2. Операторы break, continue и else....100
5.2.1. Оператор break....100
5.2.2. Оператор continue....101
5.2.3. Оператор else....102
5.3. Оператор цикла for. Функции range() и enumerate()....104

5.3.1. Функция range() 106
5.3.2. Примеры работы оператора цикла for. Функция enumerate() 108
5.3.3. Операторы break, continue и else в цикле for 111
5.4. Итератор и итерируемые объекты. Функции iter() и next() 114
5.5. Вложенные циклы 117
5.5.1. Треугольник Паскаля 122
5.6. Генераторы списков (List comprehension) 124
5.6.1. Условия в генераторах списков 126
5.6.2. Вложенные циклы в list comprehension 127
5.6.3. Вложенные генераторы списков 129
ГЛАВА VI. СЛОВАРИ, КОРТЕЖИ, МНОЖЕСТВА 131
6.1. Словари языка Python 131
6.1.1. Операции и функции работы со словарями 134
6.1.2. Методы словаря 135
6.2. Кортежи (tuple) 140
6.3. Множества (set) 145
6.3.1. Методы добавления/удаления элементов множества 148
6.3.2. Операции над множествами 150
6.3.3. Сравнение множеств 154
6.4. Генераторы множеств и словарей 156
6.5. Выражения-генераторы 158
6.6. Моржовая операция присваивания 161
6.6.1. Преимущества моржовой операции присваивания 162
6.5.2. Недостатки моржовой операции присваивания 163
ГЛАВА VII. ФУНКЦИИ 165
7.1. Что такое функции. Их объявление и вызов 165
7.1.1. Объявление и вызов пользовательских функций 167
7.1.2. Параметры функций 169
7.2. Оператор return в функциях 170
7.3. Варианты объявлений и вызовов функций 173
7.4. Алгоритм Евклида для нахождения НОД 176
7.5. Позиционные и именованные аргументы. Параметры со значениями 180
7.5.1. Параметры со значениями по умолчанию 181
7.6. Функции с произвольным числом аргументов. Коллекции args и kwargs 184
7.7. Операции упаковки и распаковки коллекций 187
7.8. Символ звездочка в параметрах функций 191
7.9. Символ слэш в параметрах функции 193
7.9.1. Совместное использование символов / и * 195
7.10. Рекурсивные функции 196
7.10.1. Примеры использования рекурсивных функций 200
7.10.2. Хвостовая рекурсия 203
7.11. Анонимные (lambda) функции 205
7.12. Функции-генераторы 208
7.13. Области видимости. Ключевые слова global и nonlocal 211
7.13.1. Ключевое слово global 213
7.13.2. Ключевое слово nonlocal 214
7.14. Замыкания в Python. Вложенные функции 216
7.15. Декораторы функций 220
7.15.1. Декораторы функций без параметров 220
7.15.2. Применение декоратора с помощью операции @ 224
7.15.3. Сохранение имени и описания декорируемой функции 225
7.15.4. Декораторы функций с параметрами 227
ГЛАВА VIII. МОДУЛИ И ПАКЕТЫ 233
8.1. Импорт стандартных модулей. Команды import и from 233
8.2. Импорт собственных модулей 237
8.3. Пакеты (package) 241
8.3.1. Файл __init__.py 243
8.3.2. Коллекция __all__ 245

8.3.3. Вложенные пакеты 246
8.4. Установка сторонних пакетов 247
ГЛАВА IX. РАБОТА С ФАЙЛАМИ 252
9.1. Функция open(). Чтение данных из текстового файла 252
9.2. Обработка исключения FileNotFoundError. Менеджер контекста 257
9.2.1. Менеджер контекста для файлов 259
9.3. Запись данных в текстовый файл 260
9.4. Чтение и запись в бинарном режиме доступа 263
9.5. Модуль pickle 264
9.5.1. Функции pickle.dumps() и pickle.loads() 264
ГЛАВА X. НЕКОТОРЫЕ ПОЛЕЗНЫЕ ФУНКЦИИ ЯЗЫКА PYTHON 268
10.1. Функции type() и isinstance() 268
10.2. Функция map() 270
10.3. Функция filter() 274
10.4. Функция zip() 277
10.5. Сортировка с помощью метода sort() и функции sorted() 279
10.5.1. Параметр key метода sort() и функции sorted() 281
10.6. Функции all() и any() 284
10.6.1. Функция all() 284
10.6.2. Функция any() 285
10.6.3. Практический пример использования функции all() 286
10.6.4. Практический пример использования функции any() 287
ГЛАВА XI. ПОГРУЖЕНИЕ В ЧИСЛА. АННОТАЦИИ ТИПОВ. КОНСТРУКЦИЯ match/case 288
11.1. Форматы записи чисел 288
11.1.1. Экспоненциальная форма записи чисел 288
11.1.2. Двоичный формат записи чисел 289
11.1.3. Шестнадцатеричный формат записи чисел 290
11.1.4. Восьмеричный формат записи чисел 291
11.2. Битовые операции И, ИЛИ, НЕ, XOR 292
11.2.1. Битовая операция НЕ 292
11.2.2. Битовая операция И 293
11.2.3. Битовая операция ИЛИ 294
11.2.4. Битовая операция исключающее ИЛИ (XOR) 295
11.2.5. Операции смещения бит 296
11.3. Модуль random 297
11.3.1. Функции генерации псевдослучайных величин 298
11.3.2. Функции для работы с последовательностями 299
11.3.3. Функция seed() 300
11.4. Аннотации базовыми типами 301
11.4.1. Аннотация типов в функциях 302
11.4.2. Модуль typing и типы Union, Optional, Any, Final 305
11.5. Аннотирование различных коллекций 307
11.5.1. Аннотация списков 307
11.5.2. Аннотация кортежей 308
11.5.3. Аннотация словарей и множеств 309
11.5.4. Комбинирование типов 310
11.5.5. Аннотации для вызываемых объектов (Callable) 311
11.6. Аннотации классами. Инструмент mypy проверки аннотирования 312
11.6.1. Отличие между object и typing.Any 313
11.6.2. Инструмент mypy 314
11.6.3. Аннотация типами Type и TypeVar 315
11.6.4. Аннотации внутри классов 317
11.6.5. Когда следует использовать аннотацию типов 318
11.7. Введение в конструкцию match/case 319
11.7.1. Шаблоны сравнений оператора case 321
11.7.2. Шаблоны проверки типов 322
11.7.3. Guard (защитник) 324
11.8. Конструкция match/case с упорядоченными коллекциями данных 325

11.9. Конструкция match/case со словарями и множествами 329
11.9.1. Обработка словарей....329
11.9.2. Обработка множеств 332
11.10. Некоторые примеры и ограничения конструкции match/case....333

ЧАСТЬ II. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ 340
ГЛАВА I. ПЕРВЫЕ ШАГИ В ООП....346
1.1. Классы и объекты. Атрибуты классов и объектов 346
1.1.1. Добавление атрибутов в класс. Функции getattr() и setattr()....349
1.1.2. Удаление атрибутов из класса. Функции hasattr() и delattr()....350
1.1.3. Атрибуты экземпляров (объектов) классов 351
1.2. Методы классов. Параметр self....352
1.3. Инициализатор __init__() и финализатор __del__() 357
1.3.1. Инициализатор __init__()....357
1.3.2. Финализатор __del__() 361
1.4. Магический метод __new__()....362
1.4.1. Пример использования метода __new__ в реализации паттерна Singleton....364
1.5. Методы класса (classmethod) и статические методы (staticmethod)....367
1.5.1. Методы класса (classmethod)....368
1.5.2. Статические методы класса (staticmethod)....370
ГЛАВА II. РЕЖИМЫ ДОСТУПА, СВОЙСТВА, ДЕСКРИПТОРЫ 372
2.1. Режимы доступа public, private, protected. Сеттеры и геттеры....372
2.2. Паттерн «Моносостояние» (Monostate)....377
2.3. Объект-свойство property. Декоратор property 379
2.3.1. Добавление делитера (deleter) в объект-свойство property....381
2.3.2. Методы getter(), setter() и deleter() объекта-свойства property....383
2.3.3. Декоратор property....384
2.4. Пример разработки класса Person....385
2.5. Дескриптор данных (data descriptor)....390
2.6. Дескриптор без данных (non-data descriptor)....396
2.7. Вложенные классы 398
ГЛАВА III. МАГИЧЕСКИЕ МЕТОДЫ КЛАССОВ....402
3.1. Методы __setattr__, __getattribute__, __getattr__ и __delattr__....402
3.1.1. Магический метод __getattribute__....404
3.1.2. Магический метод __getattr__....407
3.1.3. Магический метод __setattr__....408
3.1.4. Магический метод __delattr__....409
3.2. Магический метод __call__. Классы-декораторы....410
3.2.1. Класс-декоратор без параметров....413
3.2.2. Класс-декоратор с параметрами....415
3.3. Магические методы __str__() и __repr__()....418
3.4. Магические методы __len__() и __abs__()....420
3.5. Магические методы __add__(), __radd__(), __iadd__()....421
3.5.1. Магический метод __add__()....422
3.5.2. Магический метод __radd__()....424
3.5.3. Магический метод __iadd__()....425
3.6. Магические методы остальных арифметических операций....426
3.7. Магические методы операций сравнения 428
3.8. Магический метод __hash__() и его взаимосвязь с методом __eq__()....431
3.9. Магический метод __bool__()....436
3.10. Магические методы __getitem__(), __setitem__() и __delitem__()....438
3.10.1. Магический метод __getitem__()....438
3.10.2. Магический метод __setitem__()....440
3.10.3. Магический метод __delitem__()....441
3.11. Магические методы __iter__() и __next__()....442
3.11.1. Магический метод __next__()....443
3.11.2. Магический метод __iter__()....444
3.11.3. Пример реализации итератора для перебора атрибутов объекта класса....445

ГЛАВА IV. НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ 449
4.1. Наследование в объектно-ориентированном программировании 449
4.1.1. Расширение и переопределение 454
4.2. Функция issubclass(). Наследование от стандартных типов 454
4.2.1. Наследование от стандартных типов данных 457
4.3. Функция super() и делегирование 458
4.4. Поведение атрибутов private и protected при наследовании 462
4.4.1. Атрибуты private и protected объектов классов 462
4.4.2. Атрибуты private и protected самих классов 465
4.5. Полиморфизм и абстрактные методы 466
4.5.1. Использование исключения NotImplementedError 469
4.5.2. Модуль abc (Abstract Base Classes) 470
4.6. Множественное наследование 473
4.6.1. Порядок перебора базовых классов (MRO) 476
4.6.2. Вызов методов с одинаковыми именами из базовых классов 479
4.7. Коллекция __slots__ 480
4.8. Коллекция __slots__ с объектами property и дескрипторами 483
4.9. Коллекция __slots__ при наследовании классов 486
ГЛАВА V. ДАТА-КЛАССЫ (DATA CLASSES) 489
5.1. Введение в дата-классы (data classes) 489
5.2. Атрибуты со значениями по умолчанию в data classes 491
5.3. Функция field() 493
5.4. Метод __post_init__() 496
5.4.1. Параметры типа InitVar 498
5.5. Параметры декоратора dataclass 499
5.6. Наследование дата-классов (data calsses) 503
5.6.1. Вызов метода __post_init__() в иерархии дата-классов 505
5.7. Функция make_dataclass() 507
ГЛАВА VI. ОБРАБОТКА ИСКЛЮЧЕНИЙ. МЕНЕДЖЕРЫ КОНТЕКСТА 509
6.1. Введение в обработку исключений. Операторы try/except 510
6.2. Блоки finally и else оператора try. Вложенные блоки try 514
6.3. Инструкция raise и пользовательские исключения 518
6.3.1. Пользовательские классы исключений 520
6.4. Менеджеры контекста. Оператор with 523
ГЛАВА VII. МЕТАКЛАССЫ 528
7.1. Введение в метаклассы. Объект type 529
7.2. Пользовательские метаклассы. Параметр metaclass 530
7.3. Учебный пример разработки метакласса 533

ЧАСТЬ I. ОСНОВЫ ЯЗЫКА PYTHON

Дорогие друзья! Мы с вами начинаем изучать язык Python – один из самых простых, практичных и широко распространенных среди огромного множества языков программирования. Иногда его еще называют Питон. Почему именно Python? Это нужно спросить у Гвидо ван Россума – главного идеолога и первоначального разработчика этого языка.



Задуман он был еще в 1980-х, а первая версия вышла в 1991 году. Конечно, с тех пор он претерпел множество улучшений, особенно в версии Python 3.0, выпущенной в декабре 2008 года. Отличия оказались настолько значительными, что программы предыдущих версий 2.x далеко не всегда можно запустить в интерпретаторе Python 3.x. Однако сейчас это уже не проблема, так как с 2008 года прошло много времени и все важные программы были адаптированы к версии 3. Именно эту последнюю, современную версию языка Python мы с вами и будем изучать.

Язык оказался настолько удачным, что он стал широко применяться при разработке алгоритмов искусственного интеллекта, в частности, в нейронных сетях. При разработке серверной части сайтов, используя известные фреймворки Django и FastAPI. Например, на нем разработаны сайты Youtube, поиск от Google, DropBox и многие другие.







Также он используется в многочисленных научных проектах, где требуются сложные математические вычисления или реализация алгоритмов обработки

данных, в том числе и больших данных (Big Data). В последнее время Python стали применять и для создания игр. А если вы еще школьник, то этот язык используется в ОГЭ и ЕГЭ по информатике, а также в олимпиадном программировании.

Но, все же, почему этот язык приобрел такую широкую популярность и завоевал любовь программистов по всему миру? Считается, что все благодаря его простым, понятным, явным конструкциям, хорошо читаемому тексту программ, богатой библиотеке модулей для реализации самых разных задач и возможность использования практически на всех платформах: Windows, Linux, Mac OS, Android, iOS и др. Хотя, на самом деле, понять почему один проект «выстреливает», а большинство уходят в небытие, еще никому не удалось. Но Python – это яркая история успеха, правда, не лишенный и своих недостатков. Главные из них – более медленная скорость работы программ и больший объем используемой памяти, по сравнению, например, с аналогичным кодом, написанном на языке C/C++. Но, все же, Python позволяет заметно быстрее реализовывать сложные алгоритмы и этим качеством он затмевает большинство современных языков программирования. А скорость работы критического фрагмента кода можно увеличить, если реализовать его на том же C/C++, а затем вызывать из Python-программ. Например, так делается при проектировании и обучении нейронных сетей. Благодаря этому получаем удобство программирования и высокую скорость исполнения.

В первой части этого пособия мы с вами изучим базовые конструкции языка, начиная от обычных переменных, и заканчивая, функциями, работой с файлами, и некоторыми дополнительными полезными конструкциями. А во второй части перейдем на следующий уровень и познакомимся с объектно-ориентированным программированием на Python. Поехали!

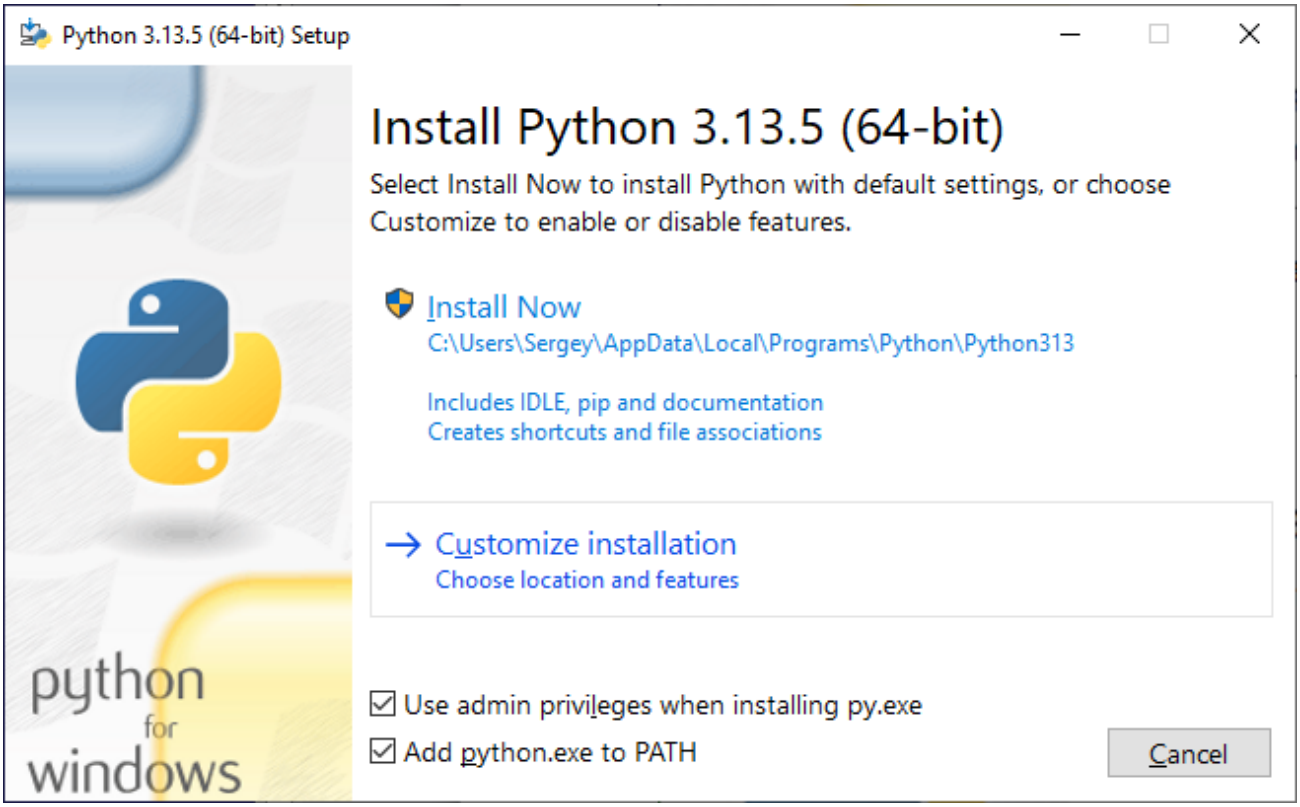
ГЛАВА I. С ЧЕГО НАЧИНАЕТСЯ PYTHON?

1.1. Установка интерпретатора языка Python

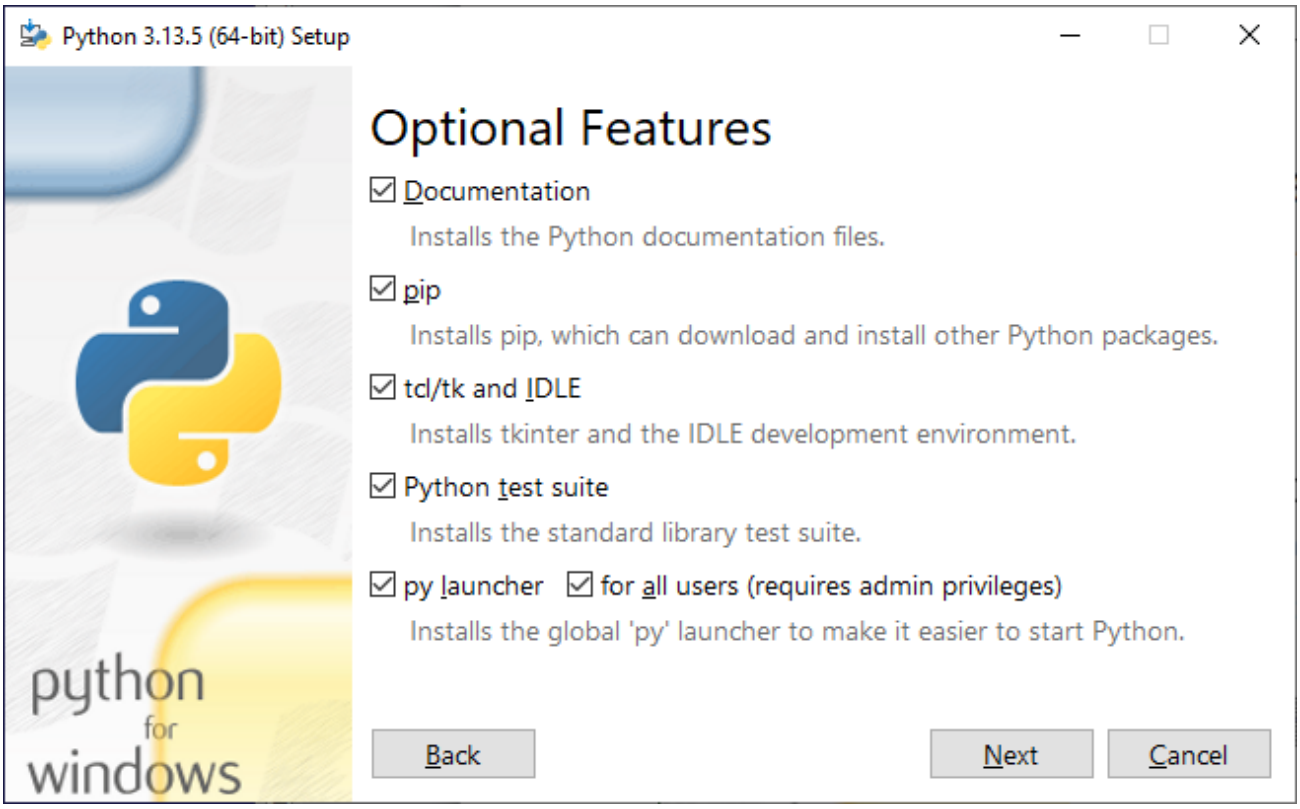
Прежде чем начинать писать свои первые программы на Python, необходимо выполнить установку интерпретатора этого языка на свое устройство. У меня – это стационарный компьютер с ОС Windows 10. При нажатии на клавиши Win + Pause, появляется окно с подробной информацией о системе. В частности, здесь указано название ОС и ее разрядность x64. Конечно, у вас может быть любое другое устройство и ОС – это не существенно, в любом случае, вначале нужно перейти на официальный сайт языка Python:

https://www.python.org

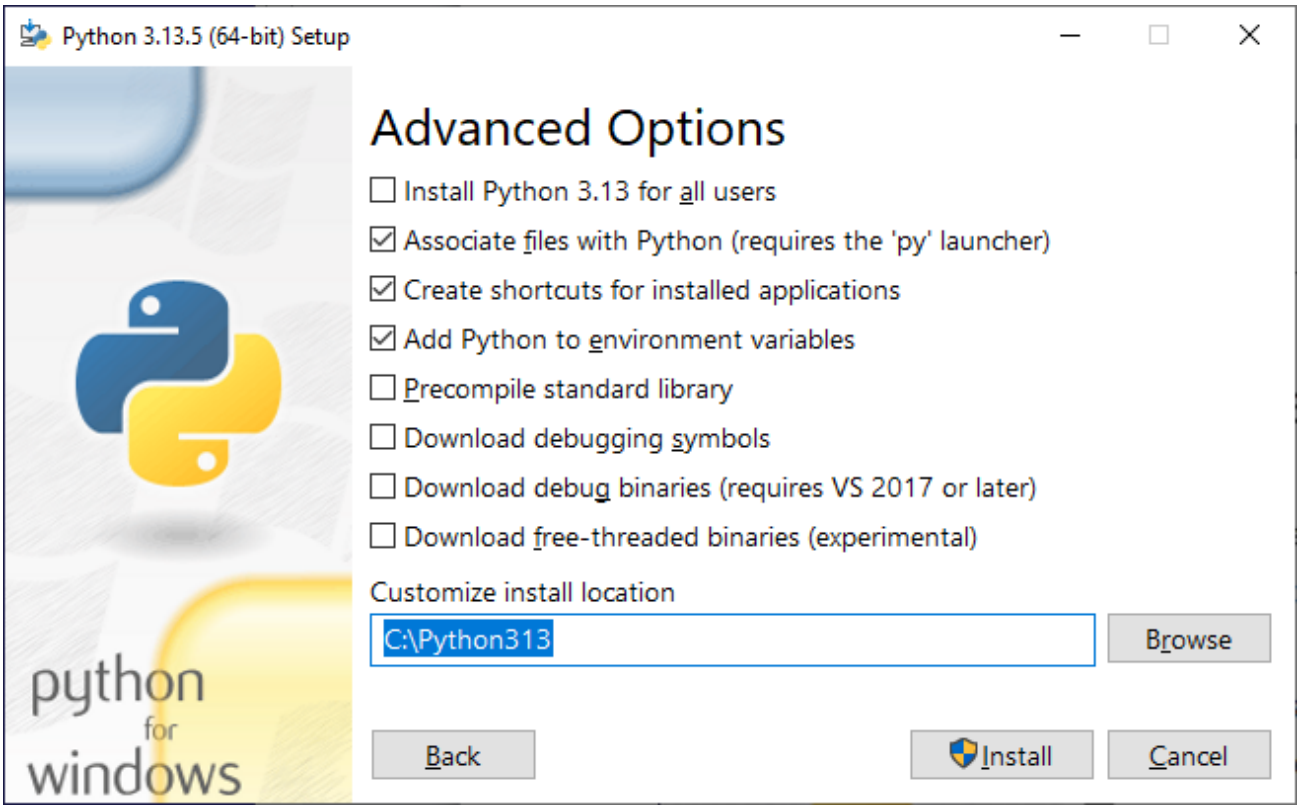
Затем выбрать вкладку «Downloads», где отобразится список различных ОС. Вам нужно найти свою ОС. Так как у меня Windows, то я выбираю этот пункт. Появляется страница с выбором различных версий языка. Конечно же, следует взять последнюю стабильную для Python 3. На момент написания книги – это версия 3.13.5. Скачиваем версию установщика и запускаем его. Появляется окно, в котором обязательно следует отметить галочкой опцию «Add Python to PATH». Она позволит в дальнейшем запускать интерпретатор языка без указания полного пути к нему:



Затем выбираем режим установки «Customize installation». Здесь все опции должны быть отмечены галочками:



Нажимаем «Next» и в следующем окне рекомендуется путь, который прописан по умолчанию, изменить на другой. А именно, убрать все промежуточные подкаталоги и установить интерпретатор в корень выбранного диска:

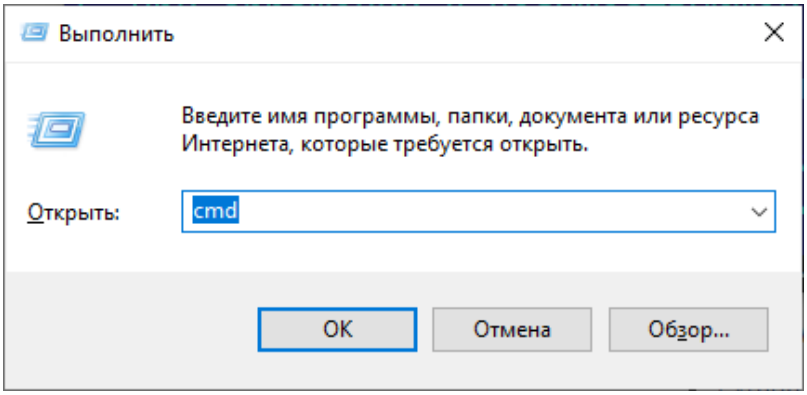


У меня получился путь «C:\Python313». Здесь дополнительно указана версия языка 3.13. Это также рекомендуется делать, так как разных версий на одном устройстве может быть несколько и чтобы переключаться между ними, они

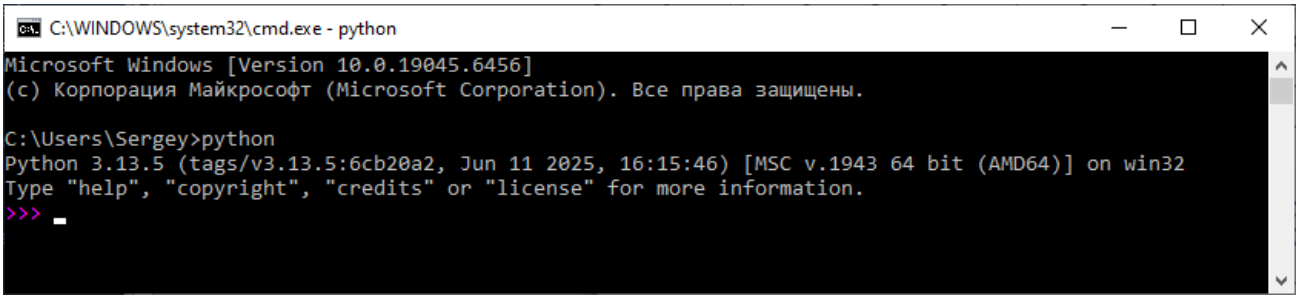
должны находиться в разных директориях.

После этого нажимаем на кнопку «Install» и программа устанавливается в указанный каталог.

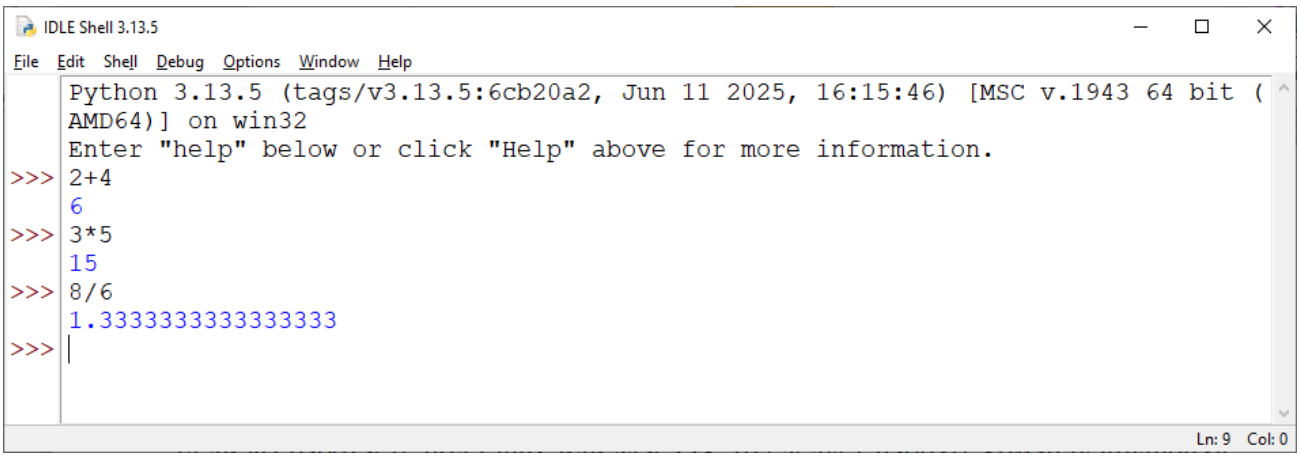
Осталось проверить работоспособность языка Python. Во-первых, мы можем открыть окно выполнения команд в ОС Windows (Win+R) и в появившемся окне набрать «cmd»:



Далее, в консольном командном окне набираем «python» и должны увидеть текущую версию интерпретатора этого языка (если вы работаете под Linux или Mac OS, то следует набирать команду «python3»):



Но писать и исполнять команды самого языка лучше через среду IDLE, которая устанавливается вместе с интерпретатором. Для ее запуска достаточно нажать на кнопку «Пуск» и выбрать оболочку IDLE. Появится окно, в котором можно в интерактивном режиме выполнять любые команды языка. Например, математические:

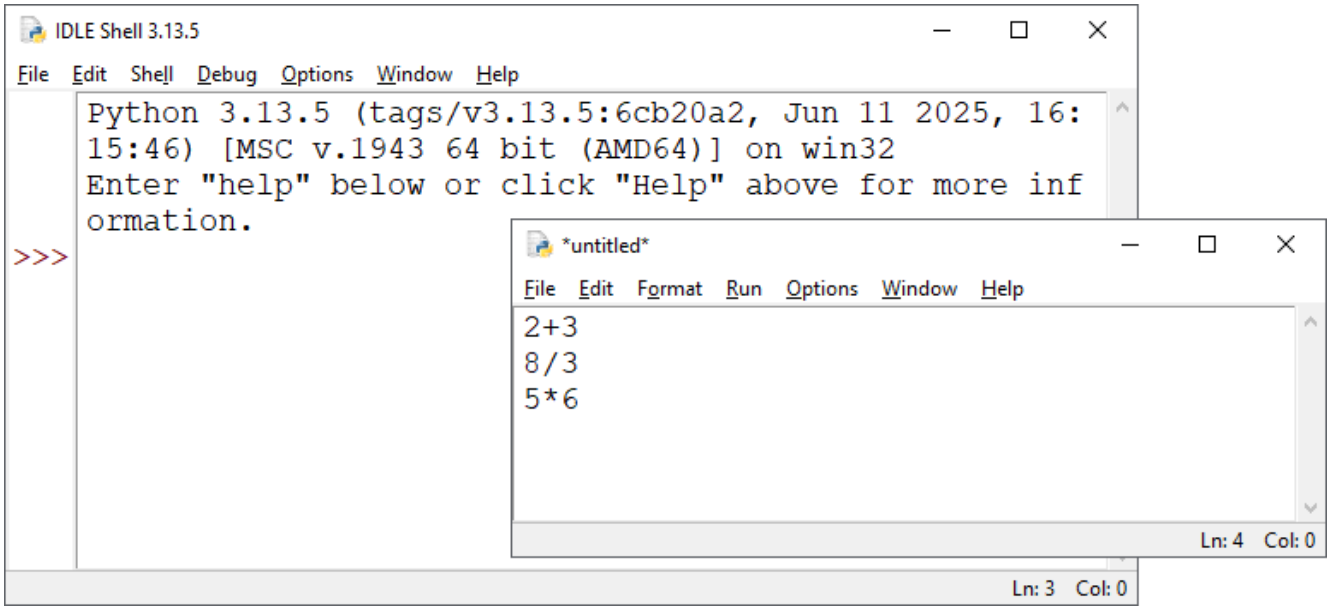


Мы о них еще подробно будем говорить. Здесь просто приводится пример того, как выполняются простейшие команды языка Python.

1.2. Варианты выполнения команд

Надеюсь вы все успешно установили интерпретатор языка на свое устройство и смогли повторить команды из прошлого занятия в среде IDLE. Так вот, это, так называемый, интерактивный режим их выполнения, когда каждая конструкция языка отрабатывает сразу же после нажатия на кнопку Enter. Преимущество такого подхода – мгновенное получение требуемого результата. Это бывает полезно при отладке программы, чтобы, например, быстренько посмотреть значения текущих данных и сделать для себя некоторые выводы.

Недостаток интерактивного режима – потеря ранее записанных команд. Например, если перезапустить IDLE, то все ранее написанное пропадает, не сохраняется. Поэтому писать полноценные программы в таком режиме крайне неудобно. Для этого следует использовать другой, файловый режим исполнения команд. В оболочке IDLE это делается следующим образом. Выбираем меню «File», затем, «New File» и у нас появляется второе окно:



Здесь можно написать сразу несколько команд:

```
2+3
8/3
5*6
```

Сохранить их в файл (CTRL+S) и, затем, выполнить программу (F5). Результат должен отображаться в командном окне IDLE. Но мы ничего не видим. Почему? Дело в том, при файловом режиме команды выполняются строчка за строчкой, то есть, последовательно, сверху-вниз, но результаты автоматически не отображаются, как это происходит в интерактивном режиме. Можно подумать, что это плохо. Возможно мы хотели бы видеть результат действия наших арифметических операций? Однако это разумное поведение интерпретатора.

Часто целью программ является не вывод всех промежуточных результатов, а только требуемого конечного значения. Но как его вывести? Для этого в языке Python существует специальная функция print. Если все наши арифметические операции записать в виде:

```
print(2+3)
print(8/3)
print(5*6)
```

то в консоли увидим три строчки, три вывода. В интерактивном режиме, кстати, она работает абсолютно так же – выводит некоторое сообщение в консоль. На данный момент, просто запомните: то, что находится внутри круглых скобок функции print, то и выводится. Причем эту функцию можно записать и с пустыми скобками:

```
print()
```

тогда просто отобразится пустая строка. Также можно указать число:

```
print(5)
print(5.6)
```

или строку:

```
print("hello")
```

Пока это просто демонстрация возможностей функции print. Подробнее о ней речь еще впереди. А до тех пор будем использовать ее в таком упрощенном виде.

1.2.1. Форматирование текста программ

Давайте вернемся к нашей программе. Как я уже говорил, операторы выполняются последовательно сверху вниз. Причем перед ними не должно быть начальных отступов (в виде пробелов) от левого края. Если добавить перед любым print хотя бы один пробел:

```
print(2+3)
 print(8/3)
print(5*6)
```

то возникнет ошибка форматирования текста программы. Python очень требователен к формату записи текста. И первое, что здесь нужно запомнить, – все начальные инструкции языка должны записываться без отступов слева и каждый print следует начинать с новой строчки. Это правила руководства PEP 8 по рекомендациям оформления текста программы на Python:

https://www.python.org/dev/peps/pep-0008/

Если его открыть, то увидим, что это достаточно объемный документ. Я буду стараться придерживаться этих указаний и делать оформление программ в соответствии с ним, чтобы вы сразу приучались к правильному стилю оформления. Конечно, для ясности и лучшей читаемости текста, можно добавлять

пустые строки, например, так:

```
print(2+3)
print(8/3)

print(5*6)
```

Подведем начальный итог:

- программа выполняется последовательно сверху-вниз;
- текст программы следует оформлять в соответствии с PEP 8;
- начальные отступы слева у команд должны отсутствовать, а каждая команда записываться с новой строки.

1.2.2. Интегрированная среда PyCharm

Писать и отлаживать программы в среде IDLE не лучшее занятие. Поэтому программисты на Python часто используют другие среды разработки. Наиболее популярной является среда PyCharm, разработанная специально для написания программ на Python.

Для установки программы PyCharm нужно перейти на официальный сайт:

https://www.jetbrains.com/pycharm/

и выбрать пункт «Download». Будет выполнен переход на следующую страницу:

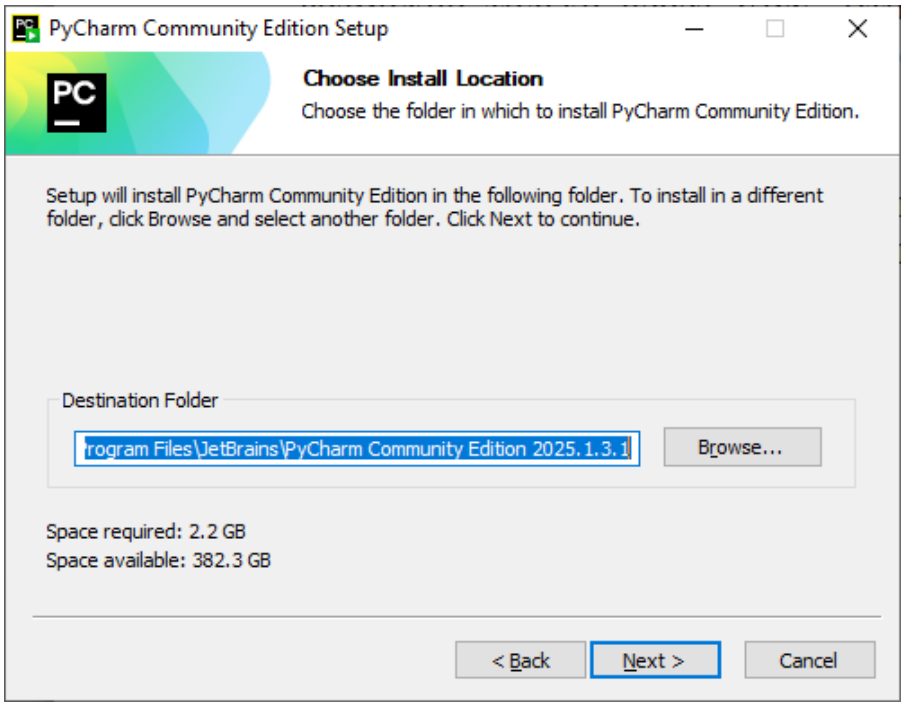
https://www.jetbrains.com/pycharm/download/

Здесь можно скачать единую (унифицированную) версию программы для свободного использования в целях обучения и для коммерческого применения.

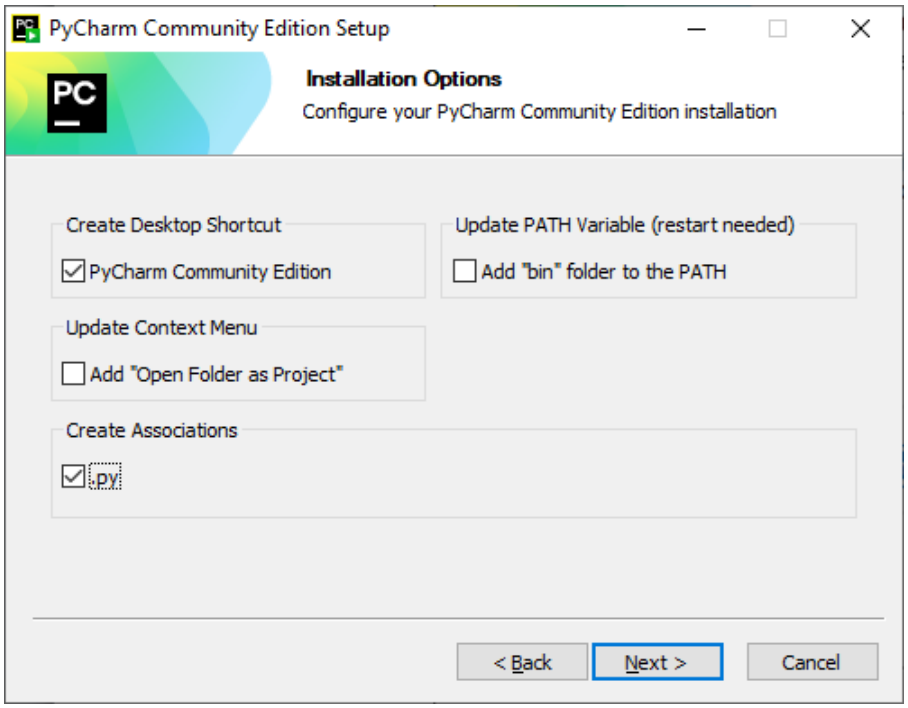
К сожалению, на момент написания книги скачать PyCharm из России возможно только через VPN. Чтобы решить эту проблему, можно воспользоваться более ранней версией PyCharm Community по следующей ссылке:

https://disk.yandex.ru/d/5PilJvsZctsGMA

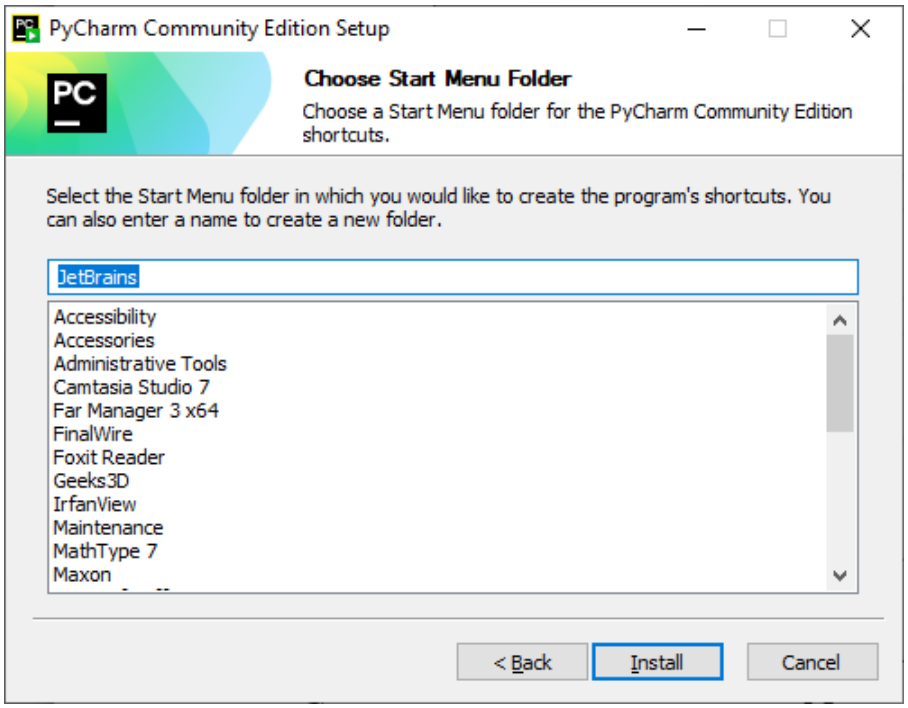
При запуске этого установщика появится окно, в котором нужно нажать кнопку «Next» и в следующем окне отобразится путь для установки PyCharm:



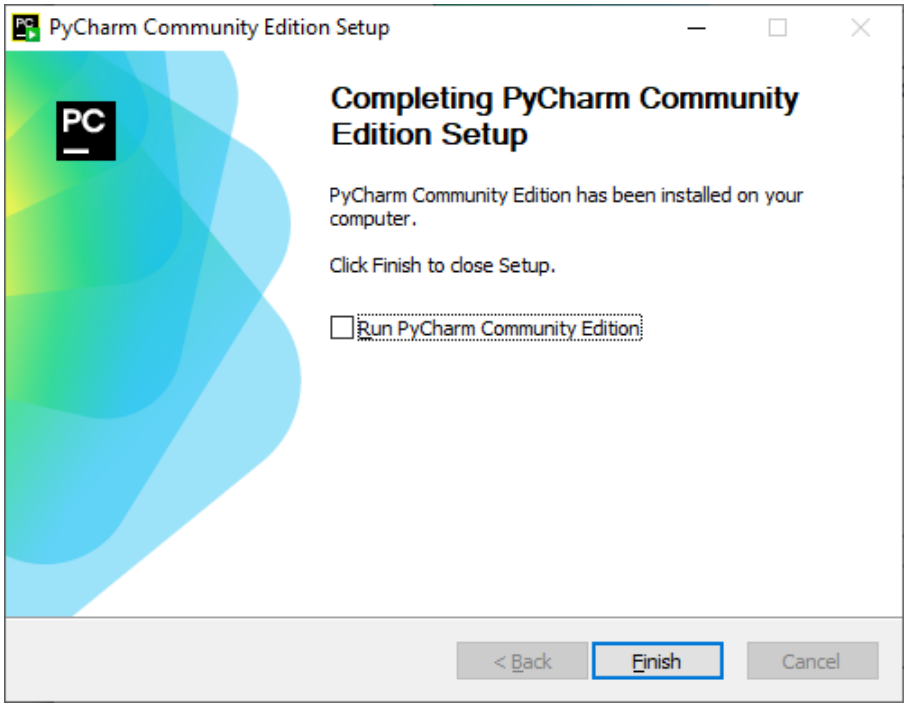
Его можно оставить без изменений. После нажатия на «Next» появится очередное окно, в котором рекомендуется поставить две галочки:



Снова нажимаем на кнопку «Next» и в следующем окне на кнопку «Install»:

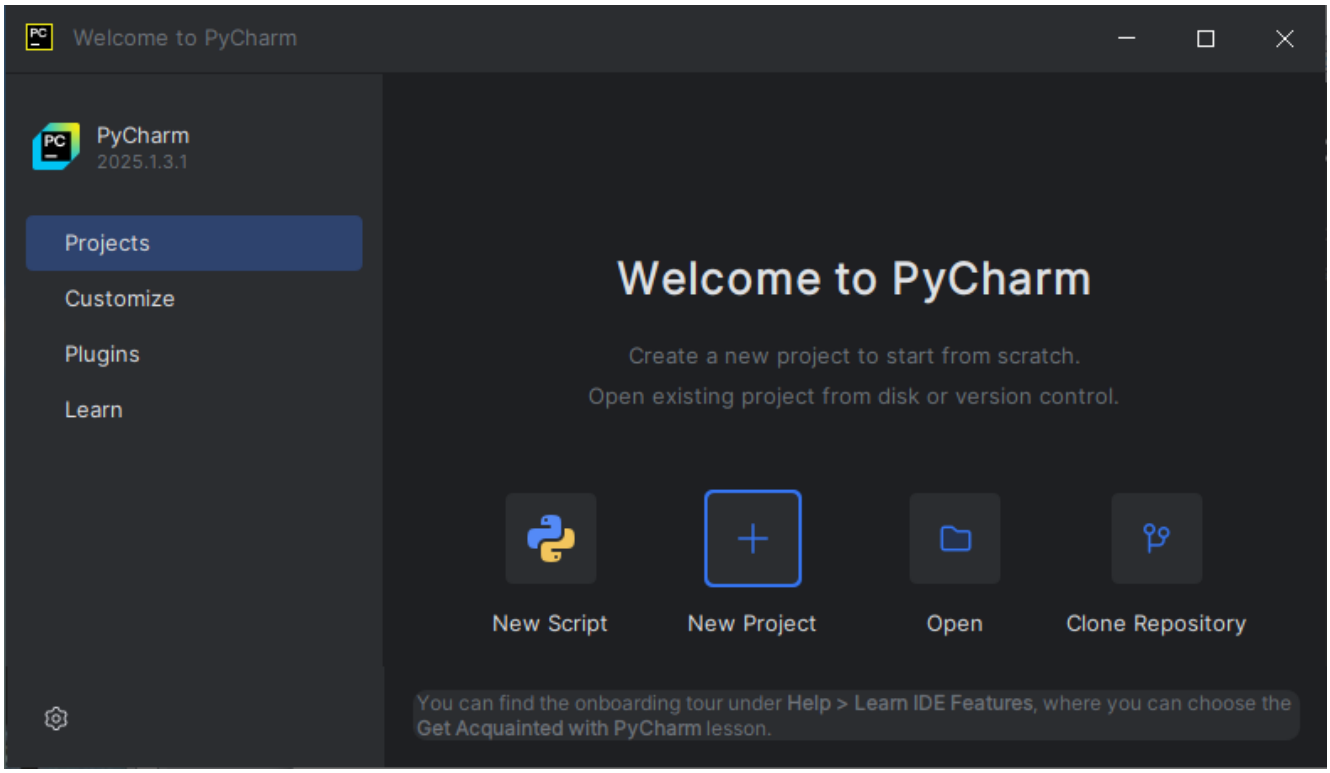


Программа начнет устанавливаться. После завершения установки нажмите на кнопку «Finish»:



После всех действий на рабочем столе должна появиться иконка с программой PyCharm. Кликаем по ней и запускаем интегральную среду.

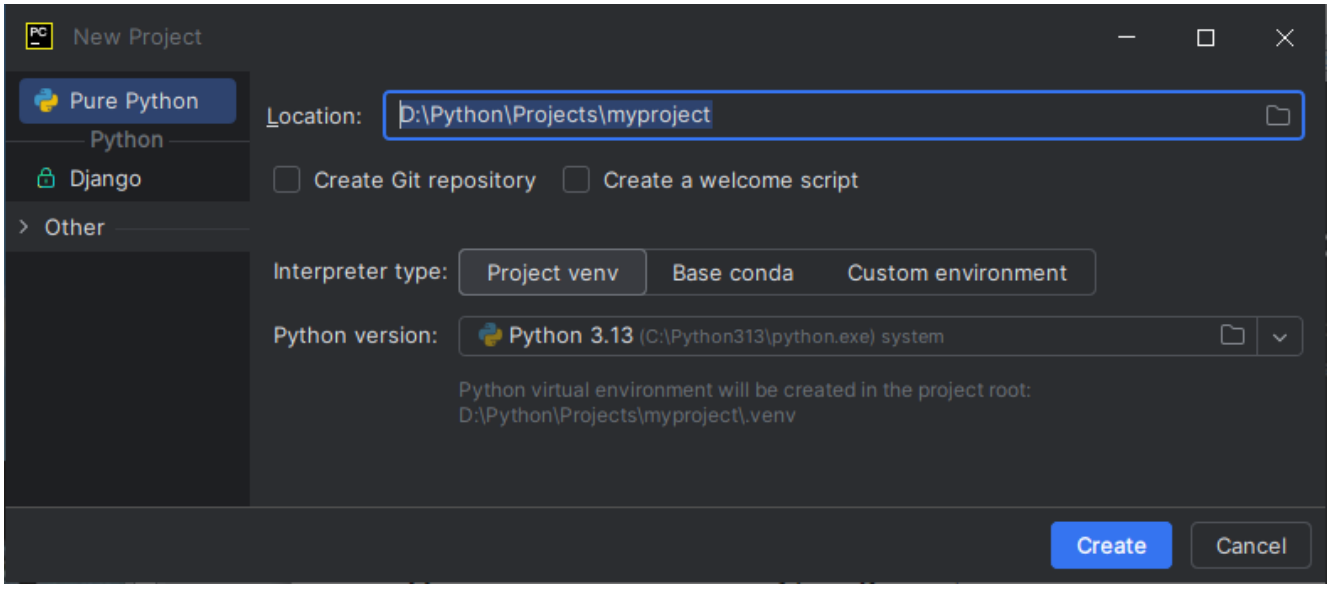
При первом запуске появится окно, в котором потребуется создать новый проект:



Нажимаем на кнопку «New Project» и в следующем окне указываем папку с желаемым расположением нового создаваемого проекта. В моем случае – это путь:

D:\Python\Projects\myproject

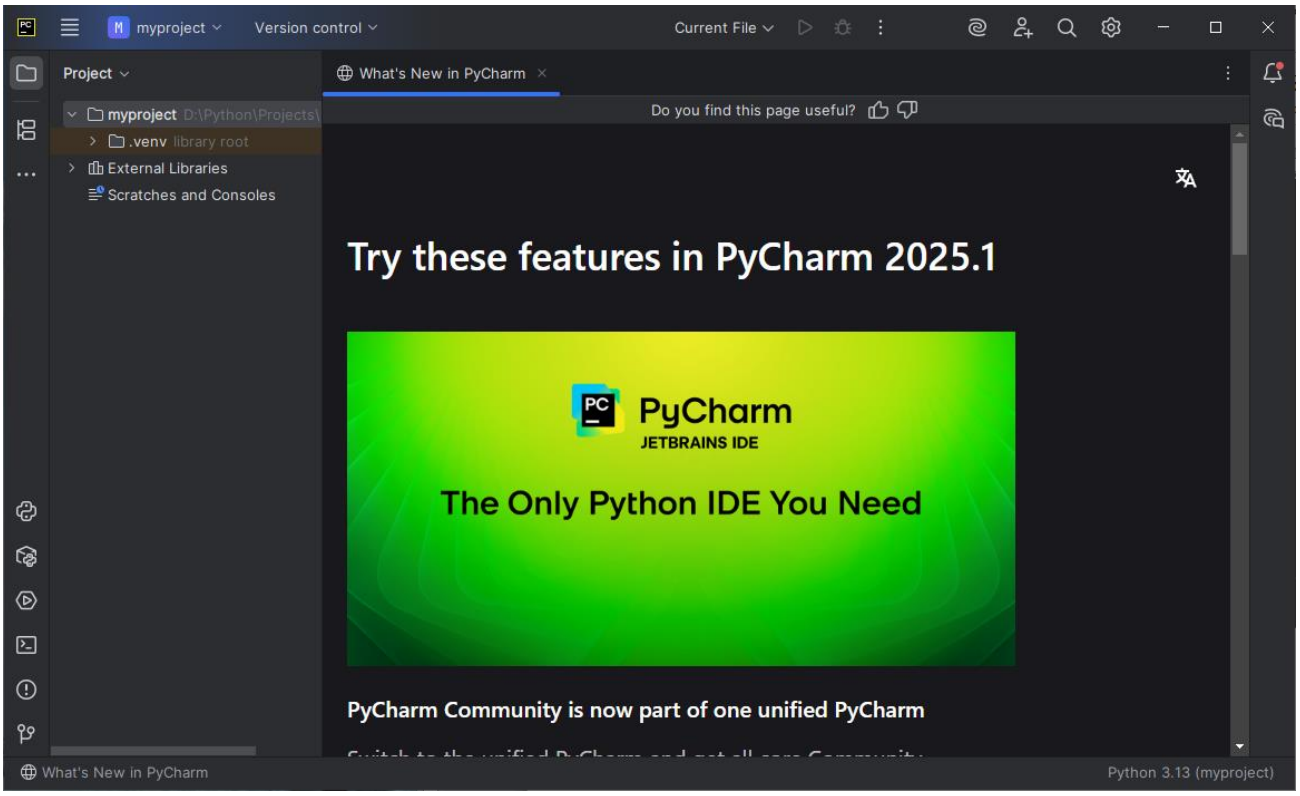
Вы, конечно же, указываете свой. Желательно без пробелов и кириллицы в его названии, чтобы избежать возможных непредвиденных ошибок.



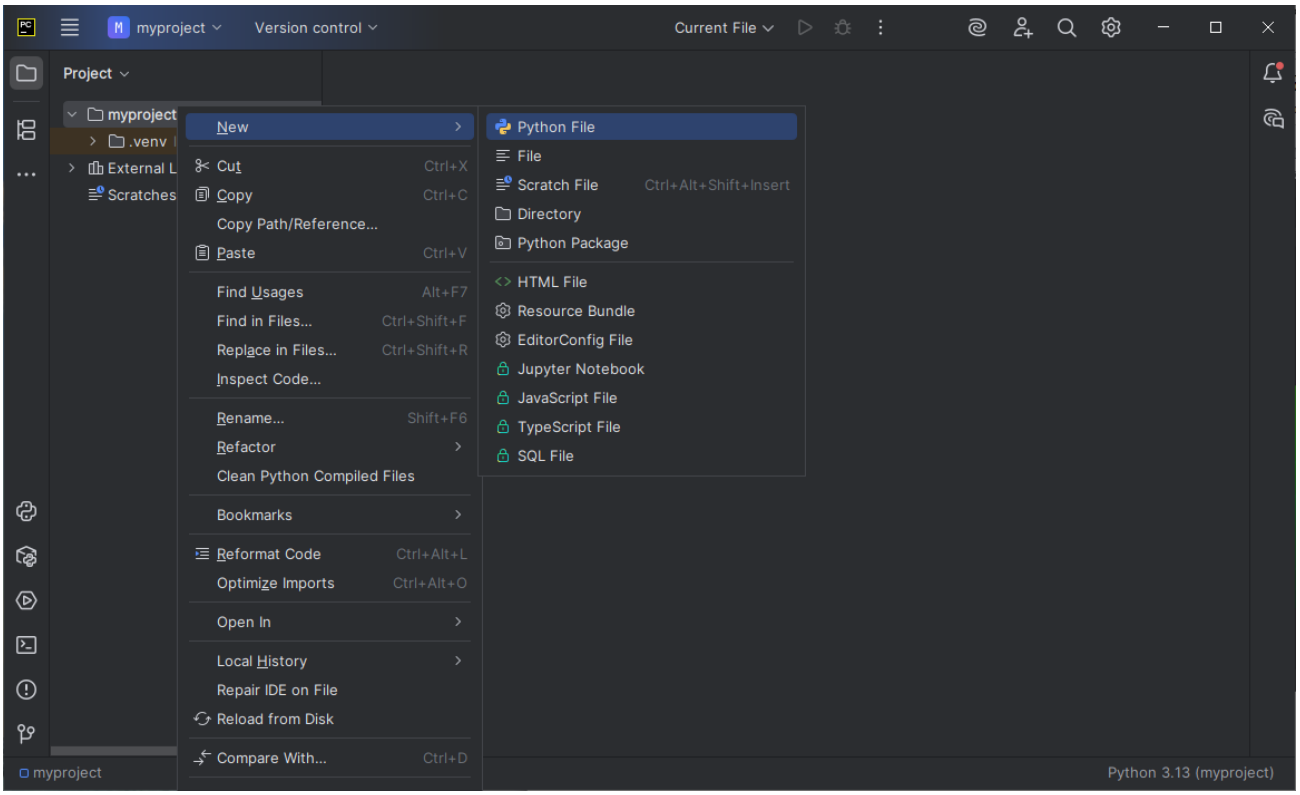
Также обратите внимание, что ниже выбран интерпретатор Python 3.13, который ранее был установлен и используется по умолчанию. У вас здесь может быть другая версия языка.

После нажатия на кнопку «Create» будет создан новый проект в отдельном

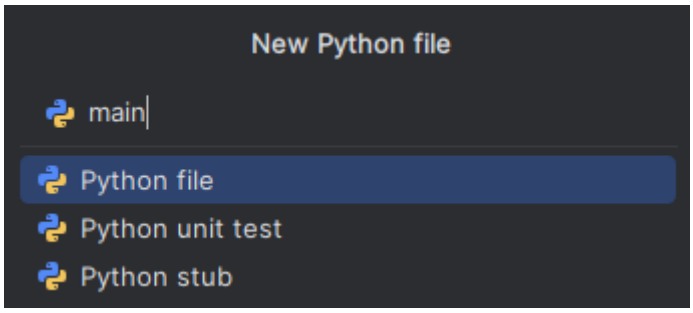
виртуальном окружении на базе выбранного интерпретатора языка Python.



Окно «What's New in PyCharm» можно закрыть. Затем щелчком правой кнопкой мыши по папке проекта «myproject» открываем всплывающее меню, в котором выберем «New» и «Python File»:



В появившемся окне следует набрать имя нового файла, например, «main», где будет располагаться текст будущей программы:

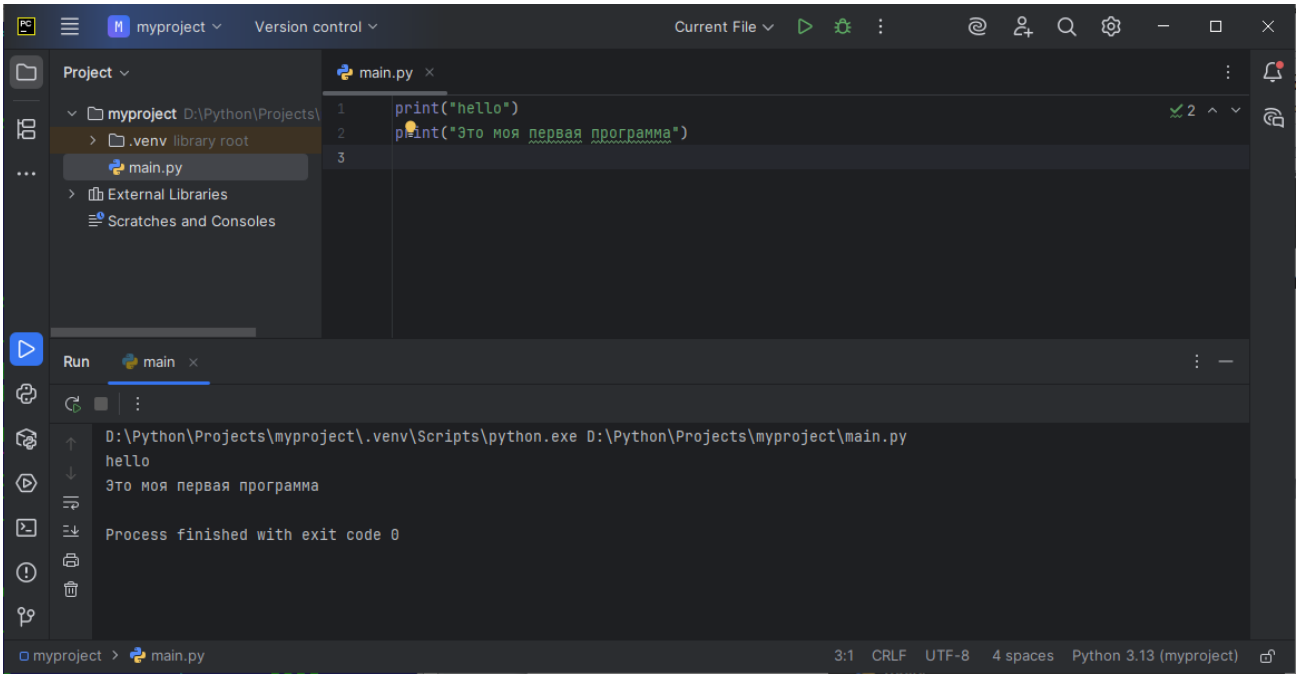


После нажатия на клавиатуре кнопки «Enter» в текущий проект будет добавлен файл «main.py».

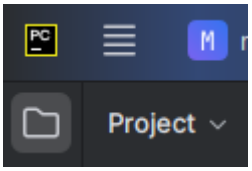
Давайте для примера пропишем следующие строчки в файле main.py:

```
print("hello")
print("Это моя первая программа")
```

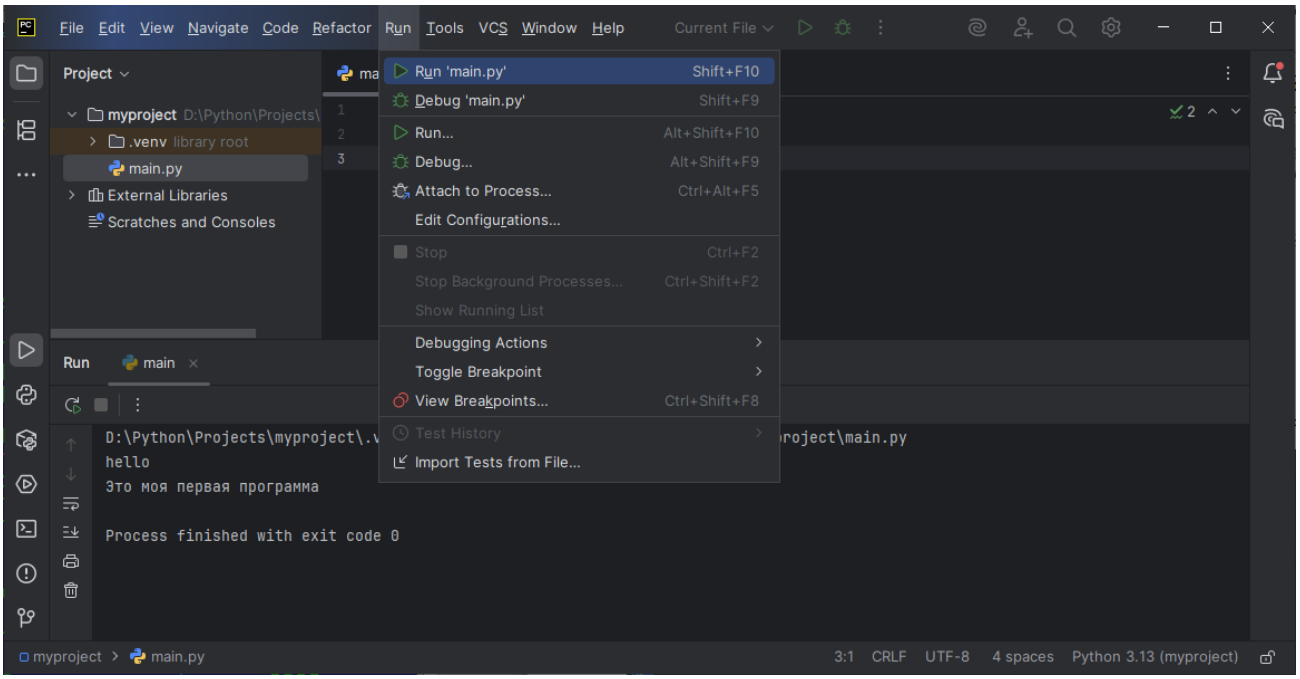
Нажимая комбинацию клавиш Shift+F10, программа должна успешно выполниться и внизу откроется консольное окно с результатом вывода двух строчек:



Также программу можно запустить через меню для этого вначале нужно кликнуть по «черточкам» рядом с иконкой программы:



И в появившемся меню выбрать пункт «Run» и далее «Run main.py»:



Но, конечно, гораздо удобнее использовать горячие клавиши:

- при первом запуске (рекомендуется): Ctrl+Shift+F10
- при повторных запусках: Shift+F10

Пока этого функционала вам будет вполне достаточно. По мере работы с PyCharm, вы будете знакомиться с его возможностями и совсем скоро, сами не заметите, как станете с ним одним целым.

ГЛАВА II. МОИ ПЕРВЫЕ ШАГИ В PYTHON

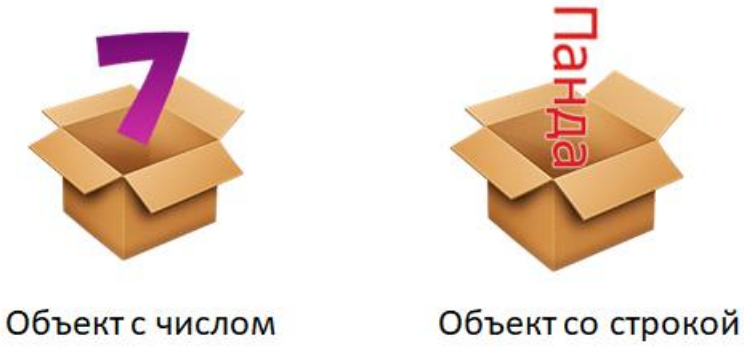
На данный момент, мы с вами установили сам язык и научились запускать программы в средах IDLE и PyCharm. Пришла пора сделать первый шаг непосредственно в программировании. Вначале давайте попробуем понять, что вообще могут делать компьютерные программы? В самой своей основе совсем немного:

- хранить данные;
- выполнять арифметические операции;
- проверять логические условия (операторы ветвления);
- реализовывать циклы.

Фактически, на комбинации этих блоков и выстраивается логика всех программ. Мы с вами последовательно познакомимся с каждым из них. И начнем с самого первого – представления и хранения данных в Python.

2.1. Переменные и оператор присваивания

Как же Python оперирует данными? Давайте представим некоторое хранилище (мы его будем называть объектом), в котором могут располагаться или числа, или строки, или, какие-либо другие типы данных. Причем в объекте может быть что-то одно: или число, или строка. Одновременно и число и строка находиться в одном хранилище не могут.



Таких хранилищ в программе может быть огромное множество. Но как нам тогда обратиться к нужному, и взять оттуда данные? Все просто. Для этого у нас должна быть ссылка на объект. По ее имени происходит обращение к хранилищу, с которым она связана. Такие ссылки называются **переменными**.



Хорошо, но как нам создать объект с некоторым содержимым и ссылкой на него? Тоже очень просто. Достаточно придумать имя переменной и через оператор

присваивания связать ее с нужным объектом, например, так:

```
a = 7
```

В результате, интерпретатор языка Python создаст ссылку с именем a и объектом с целым числом 7. Мы в этом можем легко убедиться, если выведем содержимое объекта по этой ссылке (через переменную a):

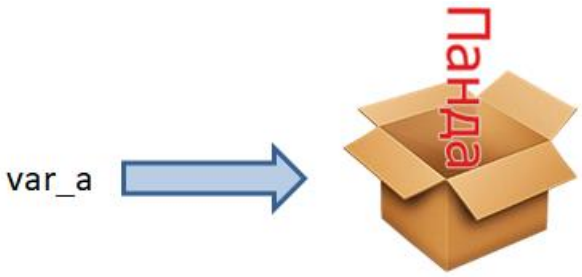
```
print(a)
```

Еще раз обратите внимание на оператор присваивания. В программировании он связывает операнд слева с операндом справа:

операнд слева = операнд справа

Например, строчка:

```
var_a = "Панда"
```



создает переменную с именем var_a, которая ссылается на объект со строкой «Панда». Обратите внимание, здесь именно **создается** переменная var_a, так как до этого она не существовала. К несуществующим переменным мы обращаться в программе не можем. Например, строчка:

```
print(x)
```

приведет к ошибке, так как переменная с именем x отсутствует. Но если ранее переменная была создана и ей снова присваивается какое-либо значение:

```
a = 6.8
```



то второй раз она не создается. Но как тогда будет работать эта строчка? Здесь формируется новый объект со значением 6.8 и существующая переменная a инициализируется этим объектом. Если на прежний объект 7 нет других ссылок, то он автоматически удаляется из памяти устройства. При этом переменная a ссылается на другой тип данных – вещественное число (до этого было

целочисленное значение). То есть, тип переменной определяется в момент присваивания ей того или иного значения. В программировании это называется **динамической типизацией**. В противовес статической типизации, когда тип переменной указывается в момент ее объявления. Например, так делается в языках C++ или Java.

Но вернемся к оператору присваивания. Давайте посмотрим, что будет, если связать одну переменную с другой:

```
b = a
```



Теперь обе переменные ссылаются на один и тот же объект. То есть, копирования данных (создание нового объекта) не происходит, копируется лишь ссылка на объект. И это важный момент, который хорошо следует запомнить и понимать! В языке Python переменные не хранят значения, а лишь ссылаются на них. Как раз, благодаря этому можно одной и той же переменной присваивать самые разные типы данных:

```
b = "Hello"
b = 0
b = -8.4
```

Это очень удобно при программировании. Но при этом всегда следует помнить, что переменная – это всего лишь ссылка на данные, а не сами данные.

Иногда, в программах можно встретить вариант **цепного присваивания** (chained assignment):

```
a = b = c = 0
```



В результате выполнения такой команды, все три переменные будут ссылаться на один и тот же объект со значением 0.

Если же мы хотим, чтобы каждая переменная ссылалась на свой отдельный объект, то можно воспользоваться **распаковкой** (иногда используют термин **множественное присваивание)**:

```
a, b = 1, 2
```



Здесь переменная a будет ссылаться на 1, а b – на 2. Используя такую команду, можно легко и просто выполнить операцию обмена значениями между двумя переменными:

```
a, b = b, a
```

2.1.1. Функция type()

Так как в программе переменные могут иметь самые разные типы данных, то, спрашивается, как можно узнать текущий тип, на который они ссылаются? Для этого в Python имеется специальная встроенная функция type, возвращающая тип данных, связанный с указанной переменной:

```
a = 5
print(type(a))
```

В консоли увидим целочисленный тип данных (int). Давайте объявим еще две переменные:

```
x = 5.8
s = "Hello"
```

и выведем их типы:

```
print(type(x), type(s))
```

Увидим float и str. Здесь у вас может возникнуть вопрос, а какие типы данных вообще существуют в Python? Конечно, можно было бы привести полный список всех типов, но на данном этапе – это избыточная, справочная информация. По мере прохождения курса, мы с вами будем знакомиться со всеми встроенными типами и постепенно у вас сложится общее представление. Такой подход гораздо эффективнее показа больших и «умных» таблиц, которые, все равно, не воспринимаются и быстро забываются.

2.1.2. Правила написания имен переменных

Давайте теперь посмотрим, как правильно прописывать имена переменных в текстах программ на Python. Есть несколько простых правил:

1. Имена следует брать существительными (отвечают на вопросы: кто, что).
2. Лучше задавать осмысленные имена, которые отражают суть данных.
3. Допустимые символы в именах: первый символ – любая буква латинского алфавита a-z, A-Z и символ подчеркивания _. В качестве второго и

последующих символов можно использовать также цифры 0-9.

Например:

```
msg = "Сообщение"
count = 0
```

Причем, обратите внимание, переменные:

```
arg = 0
Arg = 0
```

это две разные переменные, так как малая буква ‘a’ и большая ‘A’ – разные символы, а значит, имена тоже разные. Также нельзя использовать ключевые слова языка Python в качестве имен, например, писать:

```
True = 5
```

Полный их список можно посмотреть с помощью вызова в консоли функции:

```
help()
```

а, затем, набрать:

```
keywords
```

Получим:

```
False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Также не следует использовать имена стандартных функций в качестве переменных. Например, если переопределить имя функции:

```
print = 5
```

то оно теперь будет ссылаться не на функцию для печати значений, а на числовой объект 5. И если, затем, попытаться вызвать функцию:

```
print(6)
```

то возникнет ошибка, так как мы, фактически, пытаемся выполнить объект, содержащий число 5. Но Python делать этого не умеет (по крайней мере, по умолчанию). Поэтому имена стандартных функций языка Python не следует использовать в качестве имен переменных.

Но как нам узнать, является ли какое-либо имя встроенной функцией? Очень

просто. Все зарезервированные имена, будь то функции или ключевые слова, автоматически подсвечиваются интегрированными средами и подсказывают, что это имя уже имеет свою функциональность и его лучше не трогать. Со временем, вы запомните все наиболее употребительные имена языка Python и будете знать, какие имена не следует использовать для переменных.

2.2. Числа и операции над ними

Давайте теперь перейдем к первому классу данных в Python – к числам. Существуют три базовых типа для представления чисел:

- int – для целочисленных значений;
- float – для вещественных;
- complex – для комплексных.

Мы затронем первые два. Целочисленный тип int позволяет хранить целые положительные и отрицательные числа. Например:

0, 1, 2, 100, 6697959484, -1, -2, -7567658

При этом Python поддерживает работу с очень большими числами, поэтому у вас, скорее всего, не возникнет проблем с выходом за пределы диапазона.

Вещественные (дробные) числа float записываются через точку следующим образом:

6.8, -5.567, 345.546, -65467.99

Здесь тоже имеем довольно широкий диапазон значений, достаточный для большинства практических задач. Пока такого понимания чисел будет вполне достаточно.

2.2.1. Основные арифметические операции

Следующим шагом необходимо научиться выполнять арифметические операции над числами. Что это за операции? Базовыми из них являются следующие:

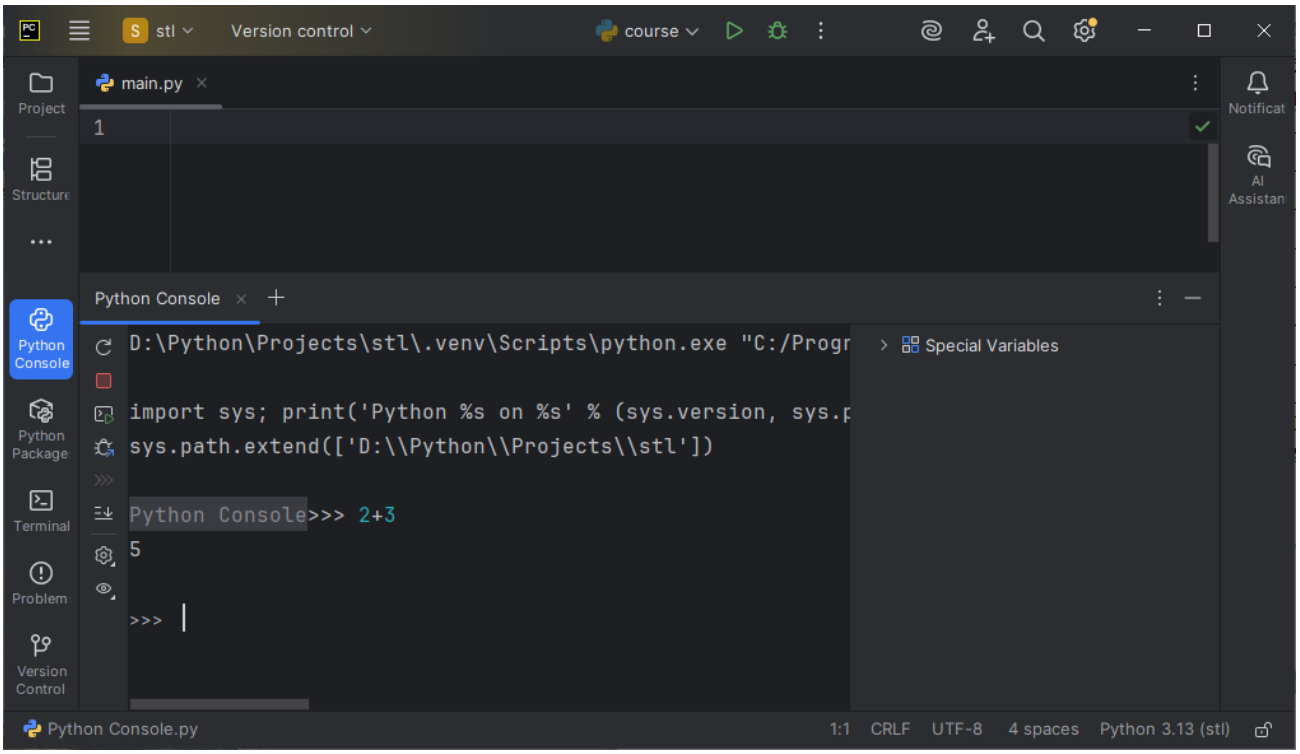
Операция	Описание	Приоритет
+	сложение	2
-	вычитание	2
*	умножение	3
/, //	деление	3
%	остаток деления	3
**	возведение в степень	4

Давайте рассмотрим их работу на конкретных примерах. В самом простом варианте можно просто сложить два целых числа:

```
2+3
```

Эту команду и все последующие удобно записывать в консоли языка Python

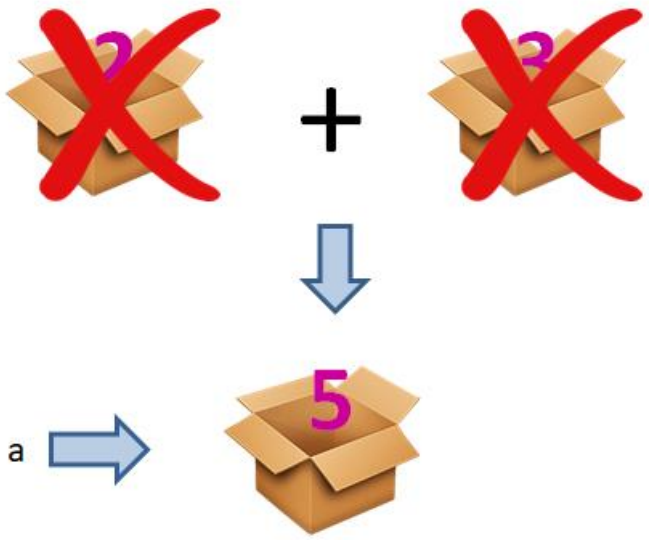
среды PyCharm, как показано на рисунке ниже:



Получим результат 5. Но этот результат у нас нигде не сохраняется. Чтобы иметь возможность делать какие-либо действия с пятеркой, ее следует сохранить через переменную, например, так:

```
a = 2+3
```

Теперь a ссылается на объект с числом 5. Давайте разберемся, как работает эта команда.



Сначала в Python создаются два объекта со значениями 2 и 3. Операция сложения берет эти значения, складывает их и формирует третий объект со значением 5. А, затем, через оператор присваивания этот объект связывается с переменной a. В конце, если на объекты 2 и 3 не ссылаются никакие другие переменные, они

автоматически удаляются из памяти сборщиком мусора.

Возможно вас удивило, что при такой простой операции сложения двух чисел выполняется столько шагов. Но в Python реализовано все именно так. И это справедливо для всех арифметических операций. Мало того, раз операция сложения возвращает объект с результатом, то можно сделать и такое сложение из трех чисел:

```
b = 2 + 3 + 4
```

И так записывается сколько угодно операций сложения в одну цепочку.

Давайте теперь сложим целое число с вещественным:

```
c = 2 + 3.5
```

Очевидно, что результат получается тоже вещественным. Отсюда можно сделать вывод, что сложение целого числа с вещественным всегда дает вещественное значение.

А вот при делении двух любых чисел мы всегда будем получать вещественное число (даже если целые числа можно разделить нацело):

```
d1 = 8 / 2  # 4.0
d2 = 3 / 6  # 0.5
```

Здесь символ ‘#’ означает комментарий в тексте программы, то есть, все, что записано после него игнорируется интерпретатором Python, и служит для чтения программистом с экрана устройства.

Если же нам нужно выполнить деление с округлением к наименьшему целому, то это делается через операцию:

```
d3 = 7 // 2
```

На выходе получаем целое значение 3, так как оно является наименьшим целым по отношению к 3,5. Обратите внимание, что при делении отрицательных чисел:

```
d3 = -7 // 2
```

получим уже целое значение -4, так как оно наименьшее по отношению к -3,5. Этот момент следует иметь в виду, применяя данную операцию деления.

Следующая операция умножения работает очевидным образом:

```
5 * 6      # 12 (целое число)
2 * 4.5    # 9.0 (вещественное число)
```

Обратите внимание, в последней операции получаем вещественное значение 9.0, а не целое 9, так как при умножении целого на вещественное всегда формируется вещественный результат.

Давайте теперь предположим, что необходимо вычислить целый остаток от

деления. Что это вообще такое? Например, если делить

$$10 : 3$$

то остаток будет равен 1. Почему так? Все просто. Число 3 трижды входит в число 10 и остается значение 10 - 3·3 = 1. Для вычисления этого значения в Python используется операция:

```
10 % 3   # значение 1
10 % 4   # значение 2
```

Я думаю, общий принцип понятен. Здесь есть только один нюанс, при использовании отрицательных чисел. Давайте рассмотрим четыре возможные ситуации:

```
9 % 5    # значение 4
-9 % 5   # значение 1
9 % -5   # значение -1
-9 % -5  # значение -4
```

Почему получаются такие значения? Первое, я думаю, понятно. Здесь 5 один раз входит в 9 и остается еще 4. При вычислении `-9 % 5` по правилам математики следует взять наименьшее целое, делящееся на 5. Это значение -10. А, далее, как и прежде, вычисляется разность между наименьшим кратным 5 и -9:

$$-9 - (-10) = 1$$

При вычислении `9 % -5`, когда делитель отрицательное число, следует выбирать наибольшее целое, кратное 5. Это значение 10. И, затем, по аналогии вычисляется разность:

$$9 - 10 = -1$$

В последнем варианте `-9 % -5` следует снова выбрать наибольшее целое (так как делитель отрицателен), получаем -5, а затем, вычислить разность:

$$-9 - (-5) = -4$$

Как видите, в целом, все просто, только нужно запомнить и знать эти правила. Справедливости ради надо сказать, что чаще всего операция «%» используется с целыми положительными числами. Другие варианты приведены больше для справки, если вам все же придется использовать отрицательные числа.

Последняя арифметическая операция – это возведение в степень:

```
2 ** 3       # возведение в куб
36 ** 0.5    # 36 в степени 1/2 (корень квадратный)
2 ** 3 ** 2  # 2^3^2 = 512
```

В последней строчке сначала 3 возводится в квадрат (получаем 9), а затем, 2 возводится в степень 9, получаем 512. То есть, операция возведения в степень выполняется справа-налево. Тогда как все остальные арифметические операции –

слева-направо.

2.2.2. Приоритеты арифметических операций

Давайте теперь посмотрим, что получится в результате выполнения команды:

```
27 ** 1/3
```

Видим значение 9. Почему так произошло? Ведь кубический корень из 27 – это 3, а не 9? Все дело в приоритете арифметических операций (проще говоря, в последовательности их выполнения). Приоритет у операции возведения в степень «**» – наибольший. Поэтому здесь сначала 27 возводится в степень 1, а затем, 27 делится на 3. Получаем искомое значение 9.

Если же нам нужно изменить порядок вычисления, то есть, приоритеты, то следует использовать круглые скобки, например, так:

```
27 ** (1/3)
```

В результате получаем значение 3. То есть, по правилам математики, сначала выполняются вычисления в круглых скобках, а затем, все остальные в порядке их приоритетов.

Еще один пример изменения приоритетов арифметических операций:

```
2 + 3 * 5    # 17
(2 + 3) * 5  # 25
```

Здесь приоритеты работают так, как нас учили на уроках математики. При этом не забывайте, что все арифметические операции выполняются слева-направо (кроме операции возведения в степень), поэтому в строчке:

```
32 / 4 * 2
```

сначала будет выполнено деление на 4, а затем, результат умножается на 2.

2.2.3. Некоторые дополнительные арифметические операции

Давайте предположим, что имеются две следующие переменные:

```
i = 5
j = 3
```

И требуется переменную `i` увеличить на 1, а `j` уменьшить на 2. Используя только текущие знания, это можно сделать так:

```
i = i + 1
j = j - 2
print(i, j)
```

Но можно и проще, используя операции:

```
i += 1
```

```
j -= 2
```

Результат будет тем же, но запись короче. Часто в подобных ситуациях на практике используют именно такие сокращенные записи операций.

То же самое можно делать и с умножением, делением и всеми другими арифметическими операциями:

```
i *= 3
j /= 4
print(i, j)
```

2.3. Математические функции и модуль math

Следующим шагом, после знакомства с базовыми арифметическими операциями, рассмотрим наиболее употребительные математические функции языка Python. Первая такая встроенная функция **abs** позволяет вычислять модуль числа (то есть, из отрицательных делает положительные):

```
abs(-5)   # 5
```

Как видите, для вызова функции нужно записать ее имя и в круглых скобках указать аргумент. Чтобы сохранить модуль того или иного числа, результат следует присвоить какой-либо переменной, например:

```
a = abs(-5.6)   # a = 5.6
```

Если же передать положительное число, то оно просто возвращается данной функцией:

```
abs(1.5)   # 1.5
```

Следующая функция **min** возвращает минимальное значение среди переданных ей чисел:

```
min(1, 2, 3, 0, -5, 10) # возвратится -5
```

А противоположная ей функция **max** возвращает максимальное значение:

```
max(1, 2, 3, 0, -5, 10) # возвратится 10
```

Число аргументов у этих функций может быть произвольным, но не менее одного. Поэтому запись вида:

```
max()
```

приведет к ошибке, т.к. не указан ни один аргумент.

Следующая функция **pow** возводит числа в указанную степень:

```
pow(6, 2)
```

Это аналог операции:

```
6 ** 2
```

Также можно указывать дробные (вещественные) значения:

```
pow(27, 0.5)   # 5.196152422706632
pow(27, 1/3)   # 3.0
```

Последняя встроенная функция, которую мы рассмотрим – это **round**. Она служит для округления чисел. Например:

```
round(0.5)  # возвратит значение 0
round(0.51) # возвратит значение 1
```

Обратите внимание, что функция round выполняет, так называемое, банковское округление чисел, а не школьное. Также у этой функции имеется второй необязательный параметр, указывающий точность округления. Если, например, записать:

```
round(7.8756, 2)  # 7.88
```

то число округляется с точностью до сотых (два знака после запятой). Если же указать отрицательное значение:

```
round(7.8756, -1)  # 10.0
```

то округление происходит до десятков. А так, округление до сотен и тысяч:

```
round(78756, -2)   # 78800
round(78756, -3)   # 79000
```

Следующая возможность при работе с функциями – это вызов одной из другой. Например, запись вида:

```
max(1, 2, abs(-3), -10)
```

вернет значение 3, так как вначале вызываются функции в аргументах, а затем, сама функция max. Причем вложенность можно делать любой глубины, например:

```
max(1, 2, abs(min(10, 5, -3)), -10)  # результат 3
```

2.3.1. Модуль math

Некоторые из вас могут заметить, что мы рассмотрели достаточно ограниченный набор математических функций. Конечно, в Python есть и другие. Чтобы ими воспользоваться, нужно импортировать специальный модуль math:

```
import math
```

Пока просто запомните эту команду. Об импорте мы еще подробнее будем говорить. Главное знать, что после выполнения этой команды, в программе появляется дополнительный набор общеупотребительных функций, которые доступны по следующему синтаксису:

math.<имя функции>

То есть, вначале прописывается имя модуля math, а затем, через точку, имя вызываемой функции. Давайте, для начала, рассмотрим наиболее употребительные из них.

Округление до наибольшего целого выполняется с помощью функции math.ceil:

```
math.ceil(5.2)  # значение 6
math.ceil(-5.2) # значение -5
```

Округление до наименьшего целого:

```
math.floor(5.99) # значение 5
math.floor(-3.3) # значение -4
```

Вычисление факториала числа:

```
math.factorial(6) # значение 1*2*3*4*5*6 = 720
```

Отбрасывание дробной части:

```
math.trunc(5.8)  # значение 5
```

Это аналог встроенной в Python функции:

```
int(5.8)
```

Далее, логарифм по основанию 2, по основанию 10 и по произвольному основанию:

```
math.log2(4)    # логарифм от 4 по основанию 2
math.log10(100) # логарифм от 100 по основанию 10
math.log(2.7)   # натуральный логарифм от 2.7
math.log(27, 3) # логарифм от 27 по основанию 3
```

Вычисление квадратного корня:

```
math.sqrt(49)  # корень из 49 равен 7.0
```

Также имеется полный набор тригонометрических функций, например:

```
math.sin(3.14/2)
math.cos(0)
```

Помимо этого есть следующие константы:

```
math.pi  # число пи
math.e   # число e
```

Пока этого набора функций нам будет достаточно. Если вам интересно узнать что еще содержит модуль math, то это можно посмотреть на странице официальной документации:

https://docs.python.org/3/library/math.html

2.4. Функции print() и input()

До сих пор мы с вами использовали функцию print для вывода информации на экран, но детально о ней еще не говорили. В действительности она обладает довольно широким функционалом, который облегчает вывод данных. И об этом пойдет речь в данном разделе. Кроме того, мы подробно рассмотрим еще одну функцию input, которая наоборот, читает данные из стандартного входного потока (обычно это клавиатура). Таким образом, функции input и print позволяют программе на Python читать внешние данные и выводить их, используя стандартные потоки ввода/вывода.

2.4.1. Знакомство с функцией print()

Мы с вами уже знаем, что в самом простом случае функцию print можно использовать следующим образом:

```
print(1) # вывод числа 1
```

или

```
print(3, 5, 7) # вывод строки: 3 5 7
```

выведет указанные значения через пробел. Также можно выводить отдельные переменные:

```
a = -6.84
print(a)
```

результаты арифметических операций:

```
print(a * 2 + 3)
```

или функций:

```
print(abs(a * 2 + 3))
```

Но у этой функции имеются два необязательных параметра, которые довольно часто используются на практике:

- sep – разделитель между данными;
- end – завершающий символ или строка.

Рассмотрим их работу на конкретных примерах. Предположим, выводятся три переменные:

```
a = -6.84
b = 7
c = 25.6
print(a, b, c) # -6.84 7 c = 25.6
```

При выводе, между числами автоматически добавляется пробел. Но этот символ можно поменять через параметр sep, например, следующим образом:

```
print(a, b, c, sep=" | ") # -6.84 | 7 | 25.6
```

Соответственно, чтобы параметр sep «сработал», необходимо хотя бы два аргумента в функции print. Если указать, например, один:

```
print(a, sep=" | ")  # -6.84
```

то здесь разделять нечего и он будет проигнорирован. А вот для двух переменных уже появится вертикальная черта:

```
print(a, b, sep=" | ") # -6.84 | 7
```

Второй параметр end задает окончание строки вывода и по умолчанию он содержит символ перевода на следующую строку:

end = '\n'

Это такой спецсимвол, о которых мы еще будем говорить. Так вот, благодаря параметру end два последовательных вызова функции print напечатают текст в столбик, т.е. каждую строчку с новой строки:

```
print("Hello")
print("World!")
```

На экране увидим:

```
Hello
World!
```

Но если в первой функции print прописать параметр end с символом пробела:

```
print("Hello", end=' ')
print("World!")
```

то оба слова будут выведены через пробел:

```
Hello World!
```

Разумеется, здесь параметр end с пробелом применяется только к первому print. У второго он уже берется по умолчанию с переносом строки.

Давайте теперь предположим, что в программе имеются две переменные (координаты точки) x и y:

```
x = 5.76
y = -8
```

И требуется вывести их в формате:

«Координаты точки: x = 5.76; y = -8»

Сделать это можно несколькими способами. Первый, самый очевидный, записать все через запятую:

```
print("Координаты точки: x = ", x, "; y = ", y)
```

Получим:

```
Координаты точки: x =  5.76 ; y =  -8
```

Но, начиная с версии Python 3.6, появилась возможность использовать специальные F-строки, о которых мы также еще будем говорить, но здесь я приведу простой пример, демонстрирующий удобство их применения. Запишем функцию print с F-строкой (символ f перед строкой) следующим образом:

```
print(f"Координаты точки: x = {x}; y = {y}")
```

Увидим результат вывода:

```
Координаты точки: x = 5.76; y = -8
```

Специальный символ f указывает языку Python, что используется F-строка. Внутри таких строк в фигурных скобок можно записывать любые операции языка Python. В данном случае просто указаны переменные `x` и `y`, значения которых подставляются в соответствующие места строки.

Благодаря такому удобству F-строки теперь практически повсеместно используются для формирования строк с требуемым содержимым. И мы в дальнейшем также будем пользоваться этим инструментом.

2.4.2. Знакомство с функцией input()

Вторая функция input служит для ввода (чтения) информации из стандартного входного потока, который, как правило, по умолчанию ассоциирован с клавиатурой. В самом простом варианте воспользоваться этой функцией можно следующим образом:

```
a = input()
```

В результате переменная `a` будет ссылаться на строку, введенную с клавиатуры. И тут первый важный момент. Функция input всегда возвращает строку, даже если вводится число. Например, команда:

```
a = input()
```

при вводе числа «-456» возвратит не целое число -456, а строку "-456". В чем разница между числом и строкой? Например, если в последствии применить к строке какую-либо функцию для работы с числами, скажем, abs:

```
b = abs(a)
```

то возникнет ошибка:

```
TypeError: bad operand type for abs(): 'str'
```

так как в качестве аргумента ожидается число, а не строка.

Решить эту проблему очень просто. Если мы знаем, что пользователь должен ввести целое число, то можно воспользоваться функцией int для преобразования строки в целочисленное значение. Например:

```
a = "54"
b = int(a)  # число 54
```

В результате b будет ссылаться на число 54, а не на строку. Поэтому программу, которая читает целочисленные значения с клавиатуры, можно записать так:

```
a = input()
a = int(a)
b = abs(a)
print(b)
```

Причем первые две строчки можно объединить в одну и записать в виде:

```
a = int(input())  # чтение строки и преобразование ее в целое число
```

Но функция int преобразовывает только целые числа. Если в строке будет хотя бы один не цифровой символ:

```
int("64.56")
```

то возникнет ошибка. То есть, в нашей программе пользователь обязательно должен вводить целые числа. Хорошо, но как тогда преобразовывать вещественные значения? Для этого используется другая функция:

```
float("64.56") # преобразование строки в вещественное число
```

Поэтому, когда на входе ожидаются вещественные данные, то следует использовать именно ее:

```
a = float(input())
b = abs(a)
print(b)
```

Давайте напишем программу для вычисления периметра прямоугольника. Пользователь будет вводить два числа (стороны прямоугольника), а мы, затем, вычислим периметр. Эту программу можно записать так:

```
a = float(input())
b = float(input())
print("Периметр:", 2 * (a + b))
```

Как видите, все достаточно просто. Сначала вводится первое число, нажимаем «Enter» и вводится второе число. Вводить два числа через пробел здесь нельзя, т.к. получится строка с двумя числами, разделенные пробелом и ее нельзя будет преобразовать функциями int или float.

Однако сейчас в этой программе пользователю совершенно непонятно, что нужно вводить. Поэтому в функции input можно дополнительно прописывать подсказки

в виде строки. Например, так:

```
a = float(input("Введите длину прямоугольника: "))
b = float(input("Введите ширину прямоугольника: "))
print("Периметр:", 2 * (a + b))
```

Забегая вперед отмечу, что в Python можно записать инструкцию (команду) для ввода значений через пробел. Это делается следующим образом:

```
a, b = map(float, input("Введите две стороны прямоугольника: ").split())
```

Здесь к каждому введенному значению применяется функция float для преобразования в вещественные числа. Пока просто запомните эту строчку, в дальнейшем она станет понятной, когда мы изучим строки и функцию map.

Аналогично можно записать эту же команду для ввода целых чисел:

```
a, b, c = map(int, input("Введите три целых числа: ").split())
print("Периметр треугольника:", a + b + c)
```

Этого начального материала пока будет достаточно для применения функций print и input в наших программах.

2.5. Логический тип Bool

Ранее мы с вами научились выполнять простые арифметические операции. Давайте теперь посмотрим, как в программах на Python можно делать логические выводы с формированием специальных значений:

- True – истина;
- False – ложь.

Предположим, требуется проверить следующее условие:

$$4 > 2$$

Для этого достаточно записать сравнение этих чисел буквально в таком виде:

```
4 > 2
```

Данная команда вернет результат: либо True, либо False. В данном случае получаем True (истина), так как 4 действительно больше 2. Если же записать:

```
4 > 7
```

то получим False, так как 4 меньше 7. То же самое можно делать и с переменными:

```
a = 5
b = 7.8
a <= b
```

Мало того, полученный результат сравнения, при необходимости, можно

сохранить в переменной, например, так:

```
res = a > b
```

получим значение False в переменной `res`.

Давайте теперь посмотрим на тип этой переменной:

```
type(res)  # <class 'bool'>
```

Это новый для нас тип данных bool (булевый тип), который может принимать только два значения:

- True – истина;
- False – ложь.

Причем эти значения, при необходимости, можно напрямую присваивать переменным, например, так:

```
res_false = False
```

И здесь, естественно, возникает вопрос, какие операции сравнения вообще существуют в Python? Они следующие:

<	сравнение на меньше
>	сравнение на больше
<=	сравнение меньше или равно
>=	сравнение больше или равно
==	сравнение на равенство
!=	сравнение на неравенство

Приведу несколько примеров их использования:

```
5 == 7 - 2
```

Здесь справа записана арифметическая операция. Так тоже можно делать. Вообще, в качестве операндов могут выступать любые инструкции языка Python: переменные, вычисления, функции и т.п.

Следующий пример:

```
8 >= 8  # True
8 > 8   # False
```

В первом случае получаем True, а во втором False, так как 8 не больше 8.

Обратите внимание, что операцию >= записывать наоборот => нельзя, синтаксически верна только первая запись. По аналогии прописываются и все остальные операции (без пробелов и других символов):

>=, !=, <=, ==

Также начинающие программисты очень часто вместо операции сравнения ==

прописывают оператор присваивания =. Для примера сравните следующие две команды:

```
x = 6   # присвоение числа 6
x == 6  # сравнение на равенство с числом 6
```

В первом случае мы присваиваем переменной x число 6, а во втором случае сравниваем переменную x со значением 6. И эти операции путать нельзя, хотя они и похожи друг на друга.

Давайте рассмотрим еще несколько полезных примеров операций сравнения. Предположим, требуется проверить переменную x на четность. Я напомню, что четные числа – это, например, числа:

2, 4, 6, -8, -10

то есть, числа, кратные 2. Как реализовать такую проверку? Часто это делают следующим образом:

```
x % 2 == 0
```

Здесь операция вычисления остатка от деления будет возвращать 0, если двойка укладывается в x без остатка. Получаем команду для проверки переменной на четность.

Обратную проверку на нечетность, можно записать в виде:

```
x % 2 != 0
```

По аналогии можно проверить, кратна ли переменная какому-либо значению, например, трем:

```
a % 3 == 0
```

и так для любого числа.

2.5.1. Операции not, and, or и их приоритеты

Давайте теперь поставим более сложную задачу и определим, попадает ли число (значение переменной) в диапазон [-2; 5]? Какие логические умозаключения здесь нужно провести, чтобы ответить на этот вопрос? Для простоты представим, что у нас имеется переменная:

```
y = 1.85
```

Очевидно, чтобы она попадала в диапазон [-2; 5], необходимо соблюдение двух условий:

y >= -2 и y <= 5

В Python это можно записать следующим образом:

```
y >= -2 and y <= 5
```

Здесь общее условие будет истинно, если истинно каждое из подусловий. Так работает операция and.

Противоположную проверку на непопадание переменной в этот же диапазон можно сделать такими проверками:

$$y < -2 \text{ или } y > 5$$

В Python такая команда прописывается следующим образом:

```
y < -2 or y > 5
```

Операция or возвращает истину, если истинно, хотя бы одно из подусловий. Видите разницу между and и or? В and общее условие истинно, если истинны оба подусловия, а в or – хотя бы одно. Причем операцию and в Python можно записывать и в более краткой форме. Например:

```
-2 <= y <= 5      # аналог y >= -2 and y <= 5
```

Какой вариант записи использовать, решать только вам, что удобнее, то и пишите.

Рассмотрим еще один пример. Допустим, требуется проверить `x` на кратность 2 или 3. Очевидно, это можно реализовать командой:

```
x % 2 == 0 or x % 3 == 0
```

А теперь инвертируем это условие для определения, что `x` не кратно 2 и не кратно 3. Это можно сделать двумя способами:

```
x % 2 != 0 and x % 3 != 0
```

или, используя операцию not (не):

```
not (x % 2 == 0 or x % 3 == 0)
```

Обратите внимание на круглые скобки. У операции not самый высокий приоритет, то есть, она выполняется в первую очередь (как умножение или деление в математике). Поэтому, для инвертирования всего составного условия это условие необходимо поместить в круглые скобки. Если команду записать в виде:

```
not x % 2 == 0 or x % 3 == 0
```

Это было бы эквивалентно условию:

```
x % 2 != 0 or x % 3 == 0
```

то есть, выполнялось бы инвертирование только первого выражения. Поэтому, при записи составных условий очень важно знать приоритеты этих операций:

or	1
and	2
not	3

Здесь чем выше число, тем выше приоритет.

2.5.2. Приведение к типу bool

В языке Python имеется встроенная функция **bool**, которая выполняет преобразование переданного входного аргумента в логический тип True или False. Общее правило преобразования такое. «Пустое» – это False (ложь), а «не пустое» – True (истина). Что значит «пустое» и «не пустое»? Давайте рассмотрим это на конкретных примерах. Любое не нулевое число – не пустое, то есть, истина:

```
bool(1)   # True
bool(-10) # True
```

А вот 0 – пустое число, дающее False:

```
bool(0)   # False
```

Похожим образом и со строками. Пустая строка:

```
bool("")  # False
```

а любая не пустая:

```
bool("a")  # True
bool(" ")  # True
```

Аналогично дело обстоит и с другими типами данных, с которыми мы еще только будем знакомиться.

ГЛАВА III. ПОСТИЖЕНИЕ СТРОК И СПИСКОВ

3.1. Строки в языке Python

Без строк не обходится практически ни одна программа. И сейчас мы с вами увидим, как в Python они задаются и обрабатываются.

Строки в программе можно прописывать или в двойных кавычках:

```
s1 = "Панда"
```

или в одинарных (апострофах):

```
s2 = 'Panda'
```

Если кавычки не ставить:

```
s2 = Panda
```

то Panda будет восприниматься как переменная.

Но в Python есть еще один способ определения многострочных строк, используя тройные кавычки (одинарные или двойные – без разницы) и в них прописывается текст, например, так:

```
text = '''Я Python бы выучил только за то,
что есть популярные курсы.
Много хороших курсов!'''
```

Если отобразить содержимое этой строки в консоли, то увидим специальный символ ‘\n’:

```
'Я Python бы выучил только за то,\nчто есть популярные курсы.\nМного хороших курсов!'
```

Этот символ отвечает за перенос на новую строку. Именно так его обрабатывает функция print. Если выполнить команду:

```
print(text)
```

то увидим следующий вывод строки:

```
Я Python бы выучил только за то,
что есть популярные курсы.
Много хороших курсов!
```

То есть, для перехода на новую строку используется спецсимвол, который записывается в виде ‘\n’. Если записать просто ‘n’ – это будет символ латинской буквы n, а при добавлении слеша он превращается в символ переноса строки.

Строка может вообще не содержать ни одного символа:

```
a = ""
```

Тогда получаем пустую строку. Но если добавить хотя бы один символ, даже если

это будет пробел:

```
a = " "
```

то имеем уже не пустую строку, а строку с символом пробела.

3.1.1. Базовые операции над строками

Давайте посмотрим, какие базовые операции можно выполнять со строками в Python. Например, требуется соединить (объединить) две строки между собой:

```
s1 = "Я люблю"
s2 = "язык Python"
```

Это можно сделать с помощью операции «+», которая, в случае со строками, выполняет их объединение (конкатенацию):

```
s3 = s1 + s2
print(s3)
```

Увидим результат:

```
Я люблюязык Python
```

Чтобы добавить пробел между словами можно записать сразу две операции конкатенации строк:

```
s3 = s1 + " " + s2
```

С помощью первой операции «+» добавляется пробел к первой строке, а вторая операция «+» добавляет вторую строку `s2`.

Однако, обратите внимание, что операция конкатенации работает только со строками. Поэтому, например, команда:

```
s3 = s1 + 5
```

приведет к ошибке, так как операнд справа является числом, а не строкой. Если нам все же необходимо объединить строку с числом, то предварительно число нужно преобразовать в строку. Сделать это можно с помощью специальной функции str:

```
s3 = s1 + str(5)
```

Функция str выполняет преобразование в строку различных типов данных, не ограничиваясь только числами. Например, можно указать булевое значение:

```
str(True)
```

и многие другие типы данных, о которых мы с вами еще будем говорить.

Следующая операция «*», применительно к строкам, выполняет их дублирование, на указанное число раз. Например, команда:

```
"ха " * 5     # результат: 'ха ха ха ха ха '
```

продублирует фрагмент «ха» пять раз. Причем здесь следует указывать только целое число. Для вещественных чисел возникнет ошибка:

```
"ха " * 5.5
```

так как продублировать строку 5,5 раз нельзя.

Следующая функция **len** возвращает длину строки (число символов в строке):

```
a = "hello"
len(a)  # 6
```

Для пустой строки возвратит значение 0:

```
len("")  # 0
```

И, как видите, этой функции можно передавать или переменную на строку, или непосредственно записывать строки:

```
len("Python")   # 6
```

Следующий оператор **in** позволяет проверять наличие подстроки в строке, например:

```
'ab' in "abracadabra"    # True
'abc' in "abracadabra"   # False
```

Следующая важная группа операций – это сравнение строк. В самом простом случае, строки можно сравнивать на равенство, используя ранее рассмотренные булевы операции. Например:

```
a = "hello"
a == "hello"  # True
```

Но сравнение:

```
a == "Hello"  # False
```

вернет False, так как большая буква H и малая h – это два разных символа.

Сравнение на неравенство реализуется операцией «!=» следующим образом:

```
a != "hello"   # False
a != "hello "  # True
```

Обратите внимание, что строка "hello" (без пробела) и строка "hello " (с пробелом) – это две разные строки и они не равны между собой.

Наконец, строки можно сравнивать на «больше» и «меньше». Например, «кот» больше, чем «кит» с точки зрения строк:

```
'кот' > 'кит'   # True
```

Почему это так? Дело в том, что здесь используется лексикографический порядок сравнения. Сначала берутся первые символы (они в нашем примере равны), затем переход ко вторым символам. Так как по алфавиту сначала идет символ ‘и’, а потом символ ‘о’, то ‘о’ больше, чем ‘и’. Как только встретились не совпадающие символы, сравнение завершается и последующие символы строк игнорируются.

Если взять равные строки:

```
'кот' > 'кот'
```

то получим False, так как ни один символ не больше соответствующего другого из второй строки. Но, добавив пробел в первую строку:

```
'кот ' > 'кот'
```

получим значение True, так как при всех прочих равных условиях больше считается более длинная строка. Наконец, если у первой строки первую букву сделать заглавной:

```
'Кот ' > 'кот'
```

то получим False. Это связано с тем, что каждый символ в компьютере связан с определенным числом (кодом), в соответствии с кодовой таблицей. Например, в таблице ASCII (см. рисунок ниже) мы видим, что сначала идут символы заглавных букв, а затем прописных. Поэтому коды больших букв меньше соответствующих кодов малых букв.

ASCII	Символ	ASCII	Символ	ASCII	Символ	ASCII	Символ	ASCII	Символ	ASCII	Символ	ASCII	Символ	ASCII	Символ	ASCII	Figu
0	NUL	32	Space	64	@	96	`	128	Ъ	160	Ў	192	А	224	а
1	SOH	33	!	65	A	97	a	129	Ѓ	161	ў	193	Б	225	б
2	STX	34	"	66	B	98	b	130	‚	162	J	194	В	226	в
3	ETX	35	#	67	C	99	c	131	ѓ	163	¤	195	Г	227	г
4	EOT	36	$	68	D	100	d	132	„	164	Ґ	196	Д	228	д
5	ENQ	37	%	69	E	101	e	133	…	165	¦	197	Е	229	е
6	ACK	38	&	70	F	102	f	134	†	166	§	198	Ж	230	ж
7	BEL	39	'	71	G	103	g	135	‡	167	Ё	199	З	231	з
8	BS	40	(	72	H	104	h	136	€	168	©	200	И	232	и
9	TAB	41	)	73	I	105	i	137	‰	169	Є	201	Й	233	й
10	LF	42	*	74	J	106	j	138	Љ	170	«	202	К	234	к
11	VT	43	+	75	K	107	k	139	‹	171	¬	203	Л	235	л
12	FF	44	,	76	L	108	l	140	Њ	172		204	М	236	м
13	CR	45	-	77	M	109	m	141	Ќ	173	®	205	Н	237	н
14	SO	46	.	78	N	110	n	142	Ћ	174	Ї	206	О	238	о
15	SI	47	/	79	O	111	o	143	Џ	175	°	207	П	239	п
16	DLE	48	0	80	P	112	p	144	ђ	176	±	208	Р	240	р
17	DC1	49	1	81	Q	113	q	145	‘	177	І	209	С	241	с
18	DC2	50	2	82	R	114	r	146	’	178	і	210	Т	242	т
19	DC3	51	3	83	S	115	s	147	“	179	ґ	211	У	243	у
20	DC4	52	4	84	T	116	t	148	”	180	µ	212	Ф	244	ф
21	NAK	53	5	85	U	117	u	149	•	181	¶	213	Х	245	х
22	SYN	54	6	86	V	118	v	150	–	182	·	214	Ц	246	ц
23	ETB	55	7	87	W	119	w	151	—	183	ё	215	Ч	247	ч
24	CAN	56	8	88	X	120	x	152	?	184	№	216	Ш	248	ш
25	EM	57	9	89	Y	121	y	153	™	185	є	217	Щ	249	щ
26	SUB	58	:	90	Z	122	z	154	љ	186	»	218	Ъ	250	ъ
27	ESC	59	;	91	[	123	~	155	›	187	j	219	Ы	251	ы
28	FS	60	<	92	\	124	DEL	156	њ	188	S	220	Ь	252	ь
29	GS	61	=	93	]	125		157	ќ	189	s	221	Э	253	э
30	RS	62	>	94	^	126		158	ћ	190	ї	222	Ю	254	ю
31	US	63	?	95	_	127		159	џ	191		223	Я	255	я

Конечно, в Python используется немного другая кодировка UTF-8, но в ней этот принцип сохраняется. Мы можем легко посмотреть код любого символа с помощью встроенной функции **ord**, например, так:

```
ord('К')  # код 1050 (в соответствии с UTF-8)
ord('к')  # код 1082 (в соответствии с UTF-8)
```

Как видите, для буквы ‘К’ код меньше, чем для ‘к’.

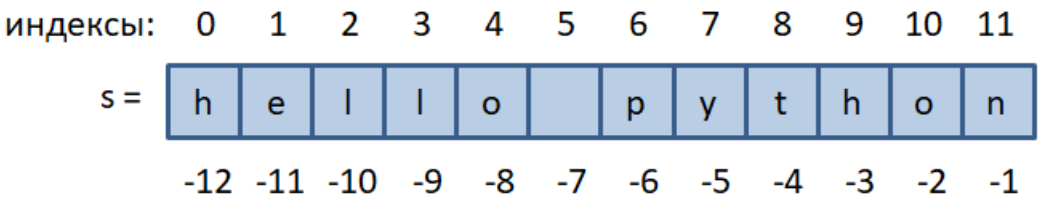
Итак, мы с вами разобрали, как задавать обычные и многострочные строки, узнали, что из себя представляет символ переноса строки, и рассмотрели базовые функции и операции со строками:

- + (конкатенация) – соединение строк;
- * (дублирование) – размножение строкового фрагмента;
- str() – функция для преобразования аргумента в строковое представление;
- len() – вычисление длины строки;
- in – операция для проверки вхождения подстроки в строку;
- операции сравнения: == != > <
- ord() – определение кода символа.

3.1.2. Индексы строк

Давайте детальнее посмотрим, что из себя представляет строка в Python, которая формируется командой:

```
s = "hello python"
```



Каждый символ строки имеет свой уникальный, порядковый номер. Эти номера называются **индексами**. Первый индекс всегда равен 0, второй на один больше и равен 1, и так до конца строки. Мы можем использовать эти индексы для обращения к отдельному символу. Для этого, прописывается имя переменной и в квадратных скобках указывается номер символа:

```
s[0]  # символ 'h'
s[1]  # символ 'e'
```

Но будьте внимательны, если указать несуществующий индекс:

```
s[12]
```

то получим ошибку выхода за пределы диапазона:

```
IndexError: string index out of range
```

В нашем примере последний индекс равен 11, а мы указали 12. Чтобы получить последний символ следовало бы записать:

```
s[11]  # символ 'n'
```

То есть, последний индекс равен длине строки минус один:

```
len(s) - 1
```

И, по идее, его так можно и записать в квадратных скобках:

```
s[len(s) - 1]
```

Особенность такой записи в том, что она будет работать для строк произвольной длины (кроме нулевой, когда символов нет). Поэтому эта команда в Python оптимизирована и записывается проще, используя только отрицательные индексы, например:

```
s[-1]  # символ 'n'
```

То есть, длину строки len(s) можно опускать и записывать отрицательные значения. Получаем способ индексации с конца строки.

Учитывая, что в Python каждая строка – это отдельный, полноценный объект, то индексацию можно прописывать непосредственно у строк, например, следующим образом:

```
"panda"[3]   # символ 'd'
```

вернет 4-й символ этой строки. Иногда такой подход используется на практике.

3.1.3. Срезы строк

В действительности, когда мы выполняем индексацию, то возвращается новая строка из одного выделенного символа. Но можно выделять сразу несколько символов, используя следующий синтаксис:

строка[start:stop)

Выделенная последовательность символов называется **срезом** строки. Например, можно записать такой срез:

```
s = "hello python"
s[1:3]   # el
```

вернет два символа с индексами 1 и 2. Последнее значение 3 не включается в срез.

В срезах можно не указывать последнее значение:

```
s[4:]  # o python
```

или первое:

```
s[:5]  # hello
```

или оба:

```
s[:]   # hello python
```

В последнем случае возвращается та же самая строка. Убедиться в этом можно, следующим образом:

```
a = s[:]
id(a)   # 2111378589744 (у вас будет другое значение)
id(s)   # 2111378589744
```

В срезах также можно использовать отрицательные индексы, например:

```
s[2:-2]   # llo pyth
```

Но если записать:

```
s[-2:2]
```

то получим пустую строку, так как эти индексы не образуют диапазон значений.

Наконец, в срезах можно дополнительно указывать еще и шаг перебора символов, согласно синтаксису:

строка[start:stop:step)

Например:

```
s[2:10:2]  # lopt
s[2::3]    # l tn
s[:5:3]    # hl
s[::2]     # hlopto
```

И использовать отрицательный шаг:

```
s[::-1]    # nohtyp olleh
```

тогда все символы будут перебираться в обратном порядке, начиная с последнего. Если же прописать:

```
s[::-2]    # nhy le
```

то будут выбираться символы с конца через один.

3.1.4. Изменение строк

При работе со строками следует иметь в виду, что они относятся к неизменяемым типам данных, то есть, существующую строку изменить нельзя. В частности, из-за этого попытка присвоить строке какой-либо символ:

```
s[0] = 'H'
```

приведет к ошибке. Чтобы изменить строку, нужно создать новую с другим содержимым, например, так:

```
s2 = 'H' + s[1:]   # новая строка 'Hello python'
```

То есть, изменение текущей строки – это всегда создание новой с измененным содержимым. И это следует иметь в виду при работе со строками в языке Python.

3.1.5. Основные методы строк

Пришло время познакомиться с основными методами строк. Что такое методы? Смотрите, когда объявляется какая-либо строка:

```
s = "python"
```

то в памяти устройства автоматически создается объект, содержащий указанные символы. Так вот, каждый такой объект связан с набором стандартных функций работы со строками. Эти функции и называются методами.



Чтобы вызвать метод для конкретной строки необходимо указать объект, поставить точку, записать имя метода и в круглых скобках список аргументов, если они необходимы:

объект.метод(аргументы)

Давайте рассмотрим все на конкретных примерах. Итак, у нас есть объект-строка, на который ссылается переменная `s`. Через эту переменную можно вызывать все методы строк. Чтобы увидеть их полный список в PyCharm достаточно прописать:

```
s.
```

Мы с вами сейчас посмотрим на работу наиболее употребительных из них. И первый такой метод:

```
s.upper()  # 'PYTHON'
```

Он возвращает новую строку (новый объект) со всеми заглавными буквами. При этом сама строка `s` остается без изменений, так как строки относятся к

неизменяемым типам данных.

Обратите внимание, что для вызова метода upper после его имени обязательно нужно ставить круглые скобки. Без них мы получим просто ссылку на объект-функцию:

```
s.upper  # <built-in method upper of str object at 0x000001F58349E8E0>
```

Выполнена она не будет. Для запуска необходимы круглые скобки в конце – это оператор для выполнения любых функций и методов.

Если требуется результат преобразования строки сохранить в какой-либо переменной, то, очевидно, сделать это можно следующим образом:

```
res = s.upper()
```

Теперь res ссылается на строку 'PYTHON'.

Противоположный метод:

```
res.lower()   # 'python'
```

наоборот, переводит все буквенные символы в нижний регистр и возвращает соответствующую строку.

Следующий метод:

string.count(sub[, start[, end]])

возвращает число повторений подстроки sub в строке string. Ему можно передавать два необязательных аргумента:

- start – индекс, с которого начинается поиск фрагмента;
- end – индекс, которым заканчивается поиск фрагмента.

В самом простом случае мы можем для строки

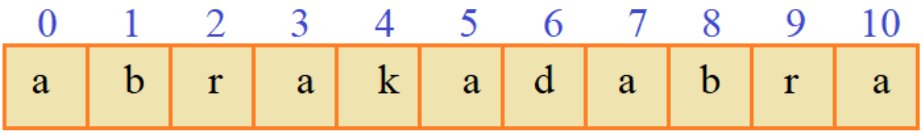
```
msg = "abrakadabra"
```

определить число повторений сочетаний «ra»:

```
msg.count("ra")
```

Получим значение 2. Именно столько подстрока «ra» встречается в строке «abrakadabra».

Теперь предположим, что поиск требуется начинать с буквы k, расположенной по индексу 4:



Тогда метод count следует вызывать со значением start=4:

```
msg.count("ra", 4)
```

Получаем значение 1. Следующим шагом укажем третий аргумент метода count – индекс, до которого будет осуществляться поиск. Предположим, что требуется дойти до 10-го индекса, имеем:

```
msg.count("ra", 4, 10)
```

Получаем значение 0. Спрашивается, почему? Мы видим, что по индексам 9 и 10 как раз записан фрагмент «ra». Но здесь, так же как и в срезах, последний индекс исключается из рассмотрения и поиск лишь доходит до 10-го, не включая его. А вот если прописать значение индекса 11:

```
msg.count("ra", 4, 11)
```

то фрагмент «ra» будет найден.

Следующий метод:

string.find(sub[, start[, end]])

возвращает индекс первого найденного вхождения подстроки sub в строке string. Аргументы start и end работают по аналогии с методом count. Например:

```
msg.find("br")
```

вернет 1, т.к. первое вхождение «br» как раз начинается с индекса 1. Пропишем теперь значение start=2:

```
msg.find("br", 2)
```

Тогда поиск начнется со второго индекса. Получим значение 8 – индекс следующего вхождения подстроки «br».

Если же указать подстроку, которой нет в строке msg:

```
msg.find("brr")
```

то метод find вернет -1.

Третий аргумент end определяет индекс, до которого осуществляется поиск и работает также как и в методе count.

Обратите внимание, что метод find выполняет поиск первого вхождения слева-направо. Если требуется осуществлять поиск в обратном направлении справа-налево, то для этого существует другой метод:

string.rfind(sub[, start[, end]])

Во всем остальном он работает так же, как и find. Например:

```
msg.rfind("br")
```

возвратит индекс 8 – первое вхождение фрагмента «br» при поиске справа в строке msg.

Наконец, третий аналогичный метод – это:

string.index(sub[, start[, end]])

Он работает абсолютно также как и find, но с одним важным отличием. Если указанная подстрока sub не находится в строке string, то метод приводит к ошибке ValueError:

```
msg.index("brr")  # ValueError
```

тогда как find возвращает -1. Спрашивается, зачем нужен такой ущербный метод index? В действительности подобные ошибки, точнее исключения, можно обрабатывать и это бывает полезно для сохранения архитектуры программы, когда неожиданные ситуации обрабатываются единым образом в блоке исключений. Но, обо всем этом речь пойдет во второй части этой книги.

Следующий метод:

string.replace(old, new, count=-1)

выполняет замену найденных фрагментов old на строку new и возвращает измененную строку. Например, в нашей строке мы можем заменить все буквы ‘a’ на ‘o’:

```
msg.replace('a', 'o')    # 'obrokodobro'
```

На выходе получим строку «obrokodobro». При этом исходная строка msg остается без изменений. Или, так:

```
msg.replace("ab", "AB")   # 'ABrakadABra'
```

Используя этот метод, можно выполнять удаление заданных фрагментов следующим образом:

```
msg.replace("ab", "")     # 'rakadra'
```

Будут удалены все найденные фрагменты «ab».

Третий необязательный аргумент задает максимальное количество замен. Например:

```
msg.replace('a', 'o', 2)   # 'obrokadabra'
```

Заменит только первые две буквы ‘a’. При значении третьего аргумента -1 количество замен неограниченно.

Следующие методы позволяют определить, из каких символов состоит та или иная строка. Например, метод:

string.isalpha()

возвращает True, если строка целиком состоит из букв, и False если имеется хотя бы один не буквенный символ. Например, команда:

```
msg.isalpha()
```

вернет True, т.к. строка msg содержит только буквенные символы. А вот для такой строки:

```
"hello world".isalpha()
```

получим False, т.к. имеется символ пробела.

Похожий набор методов:

- string.isdecimal()
- string.isdigit()
- string. isnumeric()

возвращает True, если строка целиком состоит из цифр и False в противном случае. Отличие между ними в том, какие именно символы они считают цифровыми. На данном этапе все их можно считать равноправными, но для проверки мы будем пользоваться исключительно методом isdigit, как наиболее распространенным.

Работу этого метода проще всего понять на следующих примерах:

```
"5.6".isdigit()  # False
```

Возвращает булево значение False, т.к. имеется не цифровой символ точка. А для такой строки:

```
"56".isdigit()   # True
```

получим значение True, т.к. она целиком составлена из цифровых символов.

Подобные проверки бывают полезны, например, перед преобразованием строки в целое число. Как конкретно это можно реализовать мы с вами увидим дальше при изучении условных операторов.

Следующий метод:

string.rjust(width[, fillchar = ‘ ‘])

возвращает новую строку с заданным числом символов width и при необходимости слева добавляются символы fillchar. Например, команда:

```
d="abc"
d.rjust(5)   # '  abc'
```

вернет строку « abc» с двумя добавленными слева пробелами. Этот метод можно

использовать, например, для добавления незначащих нулей перед цифрами:

```
d = "12"
d.rjust(3, '0')  # 012
```

Получим строку «012». Причем вторым аргументом можно писать только один символ. Если записать несколько, то возникнет ошибка:

```
d.rjust(3, "00")  # TypeError
```

Если ширина width будет меньше длины строки:

```
d.rjust(1)   # 'abc'
```

то возвращается исходная строка.

Аналогично работает метод:

string.ljust(width[, fillchar = ' '])

который возвращает новую строку с заданным числом символов width, но добавляет символы fillchar уже справа. Например:

```
d.ljust(10, "*")   # 'abc*******'
```

Следующий метод:

string.split(sep=None, maxsplit=-1)

возвращает коллекцию строк, на которые разбивается исходная строка string. Разбивка осуществляется по указанному сепаратору (разделителю) sep. Например:

```
"Иванов Иван Иванович".split(" ")  # ['Иванов', 'Иван', 'Иванович']
```

Строка разбивается по пробелу. На выходе формируется коллекция из строк ФИО.

Похожий результат будет и при вызове метода split без аргументов:

```
"Иванов Иван Иванович".split()   # ['Иванов', 'Иван', 'Иванович']
```

Однако есть одно важное отличие. Если в строке будут два и более подряд идущих пробелов, то результаты разбиений будут следующими:

```
"Иванов  Иван  Иванович".split(" ")# ['Иванов', '', 'Иван', '', 'Иванович']
"Иванов  Иван  Иванович".split()   # ['Иванов', 'Иван', 'Иванович']
```

То есть, `split()` учитывает от одного до нескольких подряд идущих пробелов, а `split(" ")` – строго один. Поэтому часто на практике применяется вариант вызова метода split без аргументов (первый вариант).

Предположим теперь, что перед нами задача получить список цифр, которые записаны через запятую. Причем после запятой может быть пробел, а может и не быть. Для корректного выделения всех цифр, запишем следующую программу:

```
digs = "1, 2,3, 4,5,6"
digs.replace(" ", "").split(",")    # ['1', '2', '3', '4', '5', '6']
```

Здесь сначала с помощью метода replace удаляются все пробелы из строки, а затем, вызывается метод split для получения списка цифр.

Противоположный метод:

string.join(коллекция)

возвращает строку из объединенных элементов переданной коллекции, между которыми ставится разделитель string. Например:

```
fio = ['Сергей', 'Михайлович', 'Балакирев']
", ".join(fio)   # 'Сергей, Михайлович, Балакирев'
```

получаем строку «Сергей, Михайлович, Балакирев». Или так. Изначально задана строка:

```
fio = "Иванов Иван Иванович"
```

в которой требуется пробелы заменить на запятые. Это можно выполнить командой:

```
fio2 = ",".join(fio.split())   # 'Иванов,Иван,Иванович'
```

Теперь fio2 ссылается на строку «Иванов,Иван,Иванович».

Следующий метод:

string.strip()

по умолчанию удаляет пробелы и переносы строк в начале и конце строки. Например:

```
"    hello world         \n".strip()
```

возвращает строку «hello world». Если требуется удалять какие-либо другие символы, то их можно указать при вызове метода strip, например, следующим образом:

```
'- Привет, Python!'.strip("!?.,;- ")   # 'Привет, Python'
```

Будут удаляться символы "!?.,;- ". Аналогичные методы:

string.rtrip() и string.ltrip()

удаляют символы строк только справа или только слева.

Итак, мы с вами рассмотрели следующие основные методы строк, которые следует запомнить и применять по мере необходимости в своих программах:

Название	Описание
String.upper()	Возвращает строку с заглавными буквами
String.lower()	Возвращает строку с малыми буквами
String.count(sub[, start[, end]])	Определяет число вхождений подстроки в строке
String.find(sub[, start[, end]])	Возвращает индекс первого найденного вхождения
String.rfind(sub[, start[, end]])	Возвращает индекс первого найденного вхождения при поиске справа
String.index(sub[, start[, end]])	Возвращает индекс первого найденного вхождения
String.replace(old, new, count=-1)	Заменяет подстроку old на new
String.isalpha()	Определяет: состоит ли строка целиком из буквенных символов
String.isdigit()	Определяет: состоит ли строка целиком из цифр
String.rjust(width[, fillchar = ‘ ‘])	Расширяет строку, добавляя символы слева
String.ljust(width[, fillchar = ‘ ‘])	Расширяет строку, добавляя символы справа
String.split(sep=None, maxsplit=-1)	Разбивает строку на подстроки
String.join(список)	Объединяет коллекцию в строку
String.strip()	Удаляет пробелы и переносы строк справа и слева
String.rstrip()	Удаляет пробелы и переносы строк справа
String.ltrip()	Удаляет пробелы и переносы строк слева

3.1.6. Спецсимволы и экранирование символов

Мы с вами уже видели, что строки могут содержать как обычные символы, так и специальные, например, символ перевода на новую строку:

‘\n’

В частности, когда задается многострочная строка:

```
text = """hello
python"""
```

то в нее автоматически добавляется символ перевода ‘\n’:

```
'hello\npython'
```

Причем это один символ, хотя он и выглядит как два символа: обратный слеш и n. В этом легко убедиться, вычислив длину строки:

```
len(text)  # 12
```

Получаем значение: 12 = 5 + 6 + 1 – как раз число символов в двух строках плюс один символ перевода строки. Мало того, спецсимвол ‘\n’ можно явно прописывать в любой строке, формируя многострочный текст, например, так:

```
t = "panda needs\npython"
```

Ее вывод на экран:

```
print(t)
```

даст две строки:

```
panda needs
python
```

Вообще, в строках языка Python можно прописывать следующие спецсимволы:

Обозначение	Описание
\n	Перевод строки
\\	Символ обратного слеша
\’	Символ апострофа
\"	Символ двойной кавычки
\a	Звуковой сигнал
\b	Эмуляция клавиши BackSpace
\f	Перевод формата
\r	Возврат каретки
\t	Горизонтальная табуляция (размером в 4 пробела)
\v	Вертикальная табуляция
\0	Символ Null (не признак конца строки)
\xhh	Символ с шестнадцатиричным кодом hh
\ooo	Символ с восьмиричным кодом ooo
\N{id}	Идентификатор из кодовой таблицы Unicode
\uhhhh	16-битный символ Unicode в шестнадцатиричной форме
\Uhhhhhhhh	32-битный символ Unicode в шестнадцатиричной форме
\другое	Не является экранированной последовательностью

Все их запоминать совсем не обязательно, на практике используются, в основном следующие:

\n, \\, \’, \", \t

Значительно реже другие варианты. И, обратите внимание, перед каждым спецсимволом записан символ обратного слеша. Это, своего рода, маркер начала спецсимвола. И если после слеша идет одно из обозначений таблицы, то оно будет восприниматься как некая управляющая последовательность.

Давайте, для примера, добавим в строку символ табуляции:

```
t = "\tpanda needs\npython"
```

Теперь функция print интерпретирует его, как особый горизонтальный отступ:

```
print(t)
```

Увидим на экране:

```
     panda needs
python
```

Если же убрать букву t:

```
t = "\panda needs\npython"
```

то при печати увидим просто обратный слеш:

```
\panda needs
python
```

В действительности, здесь сработала последняя строчка таблицы: когда не подходит ни одна последовательность, то просто печатается обратный слеш. Но нужно быть осторожным. Предположим, что слово needs следует заключить в обратные слеши:

t = "panda \needs\ python"

При выводе на экран получим:

```
panda
eeds\ python
```

Вместо первого слеша получаем спецсимвол переноса строки. Поэтому для добавления символа обратного слеша в строку, нужно записывать два обратных слеша подряд:

```
t = "panda \\needs\\ python"
```

В этом случае они автоматически заменяются на один символ слеша. При выводе мы это видим:

```
panda \needs\ python
```

Такой подход называется **экранированием**, когда символы с двойным назначением записываются с обратным слешем перед ними. Например, часто обратный слеш присутствует в файловых путях в ОС Windows:

D:\Python\Projects\stepik\tex1.py

Чтобы правильно записать такой маршрут обратные слеши следует экранировать:

```
path = "D:\\Python\\Projects\\stepik\\tex1.py"
```

Соответственно, при печати:

```
print(path)
```

будем видеть верный маршрут:

```
D:\Python\Projects\stepik\tex1.py
```

Если бы обратные слеши не были экранированы, то получили бы неверный путь к файлу. Этот момент следует хорошо запомнить.

Кроме обратного слеша экранировать также следует и кавычки. Например, мы хотим сформировать строку:

```
s = "Марка вина "Ягодка""  # SyntaxError
```

Внутри этой строки имеются кавычки. Но эти же самые кавычки определяют начало и конец строки в Python. Поэтому такая запись приведет к синтаксической ошибке. Чтобы все работало корректно, нужно выполнить экранирование кавычек:

```
s = "Марка вина \"Ягодка\""
```

Или, в данном случае, можно было бы использовать одинарные кавычки для определения строки, а внутри нее записать двойные:

```
s = 'Марка вина "Ягодка"'
```

Но на практике рекомендуется всегда выполнять экранирование таких символов, чтобы избежать случайных ошибок. Например, в этой строке уже нельзя просто так записать одинарные кавычки:

```
s = 'Марка вина 'Ягодка''  # SyntaxError
```

Снова получим синтаксическую ошибку. Их нужно экранировать:

```
s = 'Марка вина \'Ягодка\''
```

Однако если мы знаем, что в строке нет специальных символов и все символы следует воспринимать буквально так, как они записаны, то для ее определения можно использовать, так называемые, сырые (raw) строки. Например, если взять строку с путем к файлу:

```
path = "D:\\Python\\Projects\\stepik\\tex1.py"
```

то при ее отображении видим по одному слешу:

```
D:\Python\Projects\stepik\tex1.py
```

Но, если определить эту же строку, как сырую, добавив букву r перед ней:

```
path = r"D:\\Python\\Projects\\stepik\\tex1.py"
```

то при печати увидим по два слеша, именно так, как прописали:

```
D:\\Python\\Projects\\stepik\\tex1.py
```

Поэтому в таких строках можно убрать спецопределения и записывать все содержимое буквально:

```
path = r"D:\Python\Projects\stepik\tex1.py"
```

3.2. Форматирование строк. F-строки

Мы с вами уже знаем, как создавать строки и объединять их между собой. Предположим, есть две переменные:

```
age = 18
name = "Сергей"
```

и на их основе формируется следующее сообщение:

```
"Меня зовут "+name+", мне "+str(age)+" и я люблю язык Python."
```

На выходе получаем:

```
'Меня зовут Сергей, мне 18 и я люблю язык Python.'
```

3.2.1. Метод format()

Но это не самый удобный и распространенный способ формирования строк по заданному шаблону. До версии Python 3.6 удобнее было применять метод строки:

str.format(*args)

который позволяет подставлять в шаблон значения из args. Например, им можно воспользоваться следующим образом:

```
"Меня зовут {0}, мне {1} и я люблю язык Python.".format(name, age)
```

Смотрите, здесь в фигурных скобках указываются индексы переменных, значения которых будут подставлены в соответствующие места строки (шаблона). В частности, индекс 0 – первая переменная, указанная в методе format, т.е. name, а индекс 1 – вторая переменная age. На выходе метод format возвращает новую строку с подставленными в нее значениями. Сохраним эту новую строку в переменной msg:

```
msg = "Меня зовут {0}, мне {1} и я люблю язык Python.".format(name, age)
```

И выведем ее на экран:

```
print(msg)
```

Увидим все тот же результат:

```
'Меня зовут Сергей, мне 18 и я люблю язык Python.'
```

Причем индексы переменных в строке шаблона можно указывать несколько раз и в любом порядке. Например, так:

```
r = "Меня зовут {0}, мне {1} и {0} любит язык Python.".format(name, age)
```

```
print(r)
```

Получим результат:

```
Меня зовут Сергей, мне 18 и Сергей любит язык Python.
```

Как видите, это удобнее обычной конкатенации строк. Но эту запись можно сделать еще понятнее, используя именованные параметры. Для этого у каждой переменной можно прописывать ее имя (ключ), имена которых мы придумываем сами. Например:

```
msg = "Меня зовут {fio}, мне {old} и {fio} любит язык
Python.".format(fio=name, old=age)
```

Получаем более понятный, читаемый текст программы. Конечно, если указать неверное имя, например, так:

```
msg = "Меня зовут {name}, мне {old} и {name} любит язык
Python.".format(fio=name, old=age)
```

то возникнет ошибка, говорящая, что ключа name не существует. То есть, в фигурных скобках указывается именно ключ, а не имя переменной. Кроме того, благодаря использованию ключей, мы можем записывать переменные в методе format в любом порядке и менять их местами без опасения изменения результирующей строки:

```
msg = "Меня зовут {fio}, мне {old} и {fio} любит язык
Python.".format(old=age, fio=name)
```

3.2.2. F-строки

Начиная с версии Python 3.6, появился новый, гораздо более удобный способ форматирования строк – это, так называемые, F-строки (formatted string literals, «форматированные строковые литералы»). Они оказались настолько удачным нововведением, что теперь повсеместно используются в программах на Python, вытесняя все больше и больше предыдущий метод format.

Итак, чтобы указать Python воспринимать строку, как F-строку, перед ее литералом ставится символ ‘f’ (либо ‘F’), а внутри строки в фигурных скобках указывается произвольное Python-выражение. Например:

```
age = 18
name = "Сергей"

msg = f"Меня зовут {name}, мне {age} и я люблю язык Python."
print(msg)
```

После выполнения этой программы увидим результат:

```
Меня зовут Сергей, мне 18 и я люблю язык Python.
```

Ключевое отличие от конкатенации строк и метода format состоит в том, что F-строка вычисляется в момент выполнения, а не является статическим шаблоном.

Внутри {} можно поместить почти любое выражение языка: вызов функции, арифметические вычисления, срезы и многое другое, в том числе и еще одну F-строку (правда, с некоторыми ограничениями). Все это превращает F-строки в полноценный мини-язык внутри литерала. Например, можно записать следующую F-строку:

```
msg = f"Меня зовут {name.upper()}, мне {age*2} и я люблю язык Python."
```

Увидим при выводе на экран:

```
Меня зовут СЕРГЕЙ, мне 36 и я люблю язык Python.
```

Или даже не прописывать переменные, а использовать обычные числовые операции:

```
msg = f"Меня зовут {len(name)}, мне {14*2} и я люблю язык Python."
```

Получим:

```
Меня зовут 6, мне 28 и я люблю язык Python.
```

Как вы понимаете, символы {} внутри F-строк играют синтаксическую роль, поэтому определения вида:

```
f1 = f"строка }"       # SyntaxError
f2 = f"строка {"       # SyntaxError
```

будут приводить к синтаксическим ошибкам. Пустые фигурные скобки также недопустимы:

```
templ = f"строка {}"   # SyntaxError
```

так как внутри них предполагается Python-выражение.

Если же нам нужно просто вывести фигурные скобки внутри F-строки, то их следует дублировать. Например:

```
s1 = f"Фигурные скобки: {{ или }}"
print(s1)
```

При выводе на экран увидим:

```
Фигурные скобки: { или }
```

Как уже было сказано, внутри фигурных скобок можно прописывать практически любые Python-команды, в том числе и обычные строки. Однако здесь есть один важный нюанс. В качестве примера рассмотрим следующую программу:

```
width, height = 5, 10
msg = f"{'Периметр'}: {width} x {height} = {2 * (width + height)}"
print(msg)
```

Здесь внутри фигурных скобок записана строка 'Периметр'. Обратите внимание на кавычки. Они одинарные. Тогда как у F-строки они двойные. Различие кавычек

при отображении строк внутри F-строки было строгим требованием до версии Python 3.12. И запись вида (с одинаковыми кавычками):

```
msg = f"{"Периметр"}: {width} x {height} = {2 * (width + height)}"
```

приводила к синтаксической ошибке. Но, начиная с версии 3.12 и выше, это ограничение снято.

3.2.3. Спецификации форматирования F-строк

Давайте теперь представим, что нам нужно отобразить некоторое вещественное значение с точностью до сотых, например, число пи. Если записать программу в виде:

```
import math

msg = f"Число пи: {math.pi}"
print(msg)   # Число пи: 3.141592653589793
```

то в F-строку будет подставлено значение 3.141592653589793. Как сделать так, чтобы значение округлялось до сотых? Для этого в F-строках существует определенный набор спецификаторов форматирования. В частности, для решения нашей задачи F-строку следует записать следующим образом:

```
msg = f"Число пи: {math.pi:.2f}"
```

То есть, после выражения math.pi ставится двоеточие, а затем прописывается спецификатор «.2f». Как его понимать и интерпретировать? Здесь нужно запомнить следующее:

- Символ точка означает точность. Для вещественных величин – это число знаков после запятой, а для строк – максимальное количество выводимых символов.
- Число после точки – величина точности, т.е. либо указание количества знаков после запятой, либо указание максимальной длины строки.
- Символ f – означает вещественный тип данных. В общем случае можно указывать следующие символы типов:

d	целое число в десятичной системе
b, o, x, X	двоичное, восьмеричное, шестнадцатеричное (нижний и верхний регистр)
e, E	экспоненциальная запись числа
f, F	число с фиксированной точкой (заданной точностью)
g, G	«умный» выбор между f и e
%	процент (умножает на 100 и добавляет знак)
c	символ по значению кода (в соответствии с текущей кодовой таблицей)
s	строка (значение по умолчанию)

Примеры использования различных символов:

```
f"{255:x}"                          # 'ff'
f"{255:#X}"                         # '0XFF'
f"{'очень длинная строка':.10}"     # 'очень длин'
f"{0.756:.1%}"                      # '75.6%'
f"{42:c}"                           # '*'
f"{144.2:e}"                        # '1.442000e+02'
```

Обратите внимание, что символ точка (точность) прописывать не обязательно. В этом случае число интерпретируется, как ширина выводимых данных:

```
f"{13:5d}"       # '   13'
f"{-22:5d}"      # '  -22'
f"{1024:5d}"     # ' 1024'
f"{16384:5d}"    # '16384'
f"{-65536:5d}"   # '-65536'
```

Последний пример показывает, что если ширина меньше длины выводимой строки, то строка выводится целиком, игнорируя значение ширины. Также из этих примеров мы видим, что все данные выравниваются по правому краю с символом-заполнителем пробелом. Все это можно менять. В частности, выравнивание можно задавать следующими символами:

<	выравнивание влево
>	выравнивание право (по умолчанию)
^	выравнивание по центру
=	знак числа выравнивается отдельно

Например:

```
f"{13:<10d}"       # '13        '
f"{-22.7:^10.1f}"  # '  -22.7   '
f"{-65536:=10d}"   # '-    65536'
```

И указывать символ-заполнитель:

```
f"{1024:*^10d}"                # '***1024***'
f"{1:0>2d}:{5:0>2d}:{45:02d}"  # '01:05:45'
```

В последней F-строке в качестве заполняющего символа указано число 0. Его удобно использовать для формирования незначащих нулей при выводе чисел с фиксированной шириной, например, часов.

В общем виде синтаксис спецификации F-строк можно записать следующим выражением:

[[fill]align][sign][#][0][width][,][.precision][type]

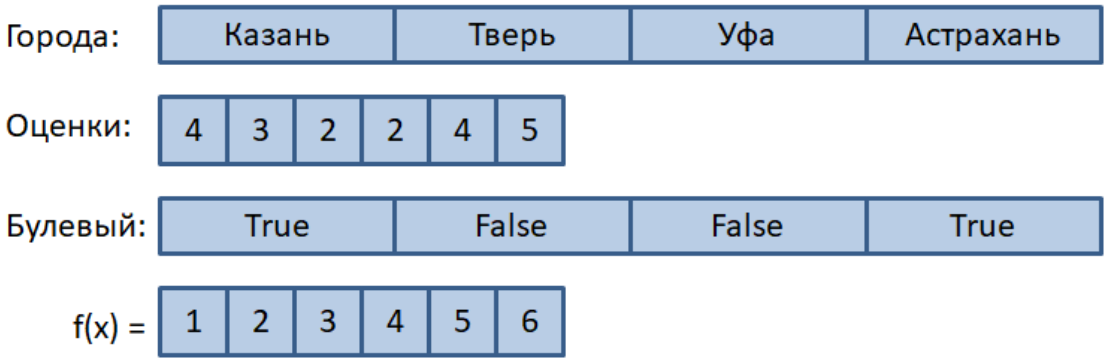
Здесь квадратные скобки означают необязательность того или иного параметра. Более подробно о нем можно почитать на странице официальной документации:

https://docs.python.org/3/library/string.html#formatspec

А нам, для начала, приведенной информации будет вполне достаточно.

3.3. Списки и операции над ними

Представьте, что нам в программе нужно хранить и обрабатывать, например, список городов, или список оценок студента, или список булевых значений, или значения функции. Во всех этих случаях требуется оперировать набором каких-либо данных. В Python для таких целей можно воспользоваться специальным объектом, который называется **список**.



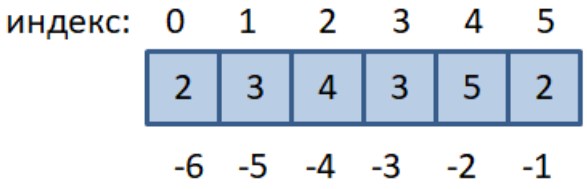
Задать список в программе очень просто: ставятся квадратные скобки и внутри них через запятую перечисляются его элементы:

```
cities = ["Москва", "Тверь", "Вологда"]
```

или

```
marks = [2, 3, 4, 3, 5, 2]
```

Давайте посмотрим, как можно оперировать данными списка, например, чтобы вычислить средний балл по оценкам. Для этого нам потребуется обращаться к отдельным элементам списка. Но как это сделать? Учитывая, что список – это **упорядоченная** коллекция, то все его элементы имеют свой порядковый номер (индекс), начиная с нулевого и заканчивая последним:



Используя эти индексы и синтаксис:

список[индекс]

мы можем обращаться к отдельным элементам. Например:

```
marks[0]  # 2
marks[2]  # 4
```

Следовательно, для вычисления среднего балла среди шести оценок можно записать команду:

```
avg = (marks[0] + marks[1] + marks[2] + marks[3] + marks[4] + marks[5]) / 6
```

Это простейший пример того, как можно хранить и использовать данные списка. Причем индексирование здесь работает так же, как и со строками. Если указать несуществующий индекс:

```
marks[10]  # IndexError
```

то получим ошибку IndexError. А чтобы обратиться к последнему элементу, можно использовать отрицательный индекс:

```
marks[-1]  # 2
```

То есть, здесь, так же как и в строках, можно использовать положительные и отрицательные индексы.

3.3.1. Список – изменяемый тип данных

В отличие от строк списки в Python относятся к изменяемым типам данных. То есть, мы можем изменить ранее хранимое значение. Например, студен пересдал первую двойку на тройку и нужно внести это изменение в список marks. Сделать это можно через оператор присваивания:

```
marks[0] = 3
```

Теперь первое значение равно 3. Как вы помните, со строками такая команда приводила к ошибке, так как строки – это неизменяемый тип. Но со списками это вполне допустимо и в этом их кардинальное отличие от строк. Список – динамическая структура данных, который может меняться в процессе работы программы. Мало того, списку, состоящему из чисел, мы легко можем присвоить любой другой тип данных, например, строку:

```
marks[1] = "удовл."
```

Вообще списки могут содержать самые разные типы данных, например:

```
lst = ["Москва", 1320, 5.8, True, "Тверь", False]
```

в том числе и другие, вложенные списки:

```
lst2 = [1, 2.5, [-1, -2, -3], 4]
```

О вложенных списках мы еще будем говорить.

Если нам нужно создать пустой список, то достаточно записать квадратные скобки без элементов:

```
a = []
```

Или воспользоваться специальной функцией:

```
b = list()
```

которая создает новый пустой список. Если же ей в качестве аргумента указать другой список:

```
a = list([True, False])   # a = [True, False]
```

то будет создан новый список с тем же самым содержимым. Также мы можем передать ей строку:

```
list("Python")   # ['P', 'y', 't', 'h', 'o', 'n']
```

тогда получим список, состоящий из отдельных символов этой строки. Вообще на вход функции list можно передавать любой перебираемый объект, на основе которого формируется новый список. Такие перебираемые объекты еще называются **итерируемыми**, но о них речь пойдет дальше.

3.3.2. Функции работы со списками

Язык Python содержит несколько удобных встроенных функций для работы со списками:

- len() – определение числа элементов в списке (длина списка);
- max() – для нахождения максимального значения;
- min() – для нахождения минимального значения;
- sum() – для вычисления суммы;
- sorted() – для сортировки коллекции.

Для начала воспользуемся уже знакомой нам функцией len для определения длины списка:

```
len(marks)   # 6
```

Соответственно, для пустого списка:

```
len([])   # 0
```

она возвращает 0.

Следующие две функции min и max нам уже знакомы. Их также можно применять со списками. Пусть для примера имеется список со значениями дневных температур города Москвы:

```
t = [23.5, 25.6, 27.3, 26.0, 30.4, 29.5]
```

Чтобы найти максимальное и минимальное значения, достаточно вызвать функции:

```
max(t)  # 23.5
min(t)   # 30.4
```

Для подсчета суммы всех значений используется функция sum:

```
sum(t)   # 162.3
```

Соответственно, вычислить среднюю температуру можно командой:

```
sum(t)/len(t)  # 27.05
```

Наконец, последняя функция sorted по умолчанию возвращает новый список с отсортированными значениями по неубыванию (или, как часто говорят, по возрастанию):

```
sorted(t)   # [23.5, 25.6, 26.0, 27.3, 29.5, 30.4]
```

Обратите внимание, эта функция не меняет прежний список t, она именно возвращает новый список с отсортированными значениями. Поэтому для сохранения результата работы его нужно присвоить какой-либо переменной, например, так:

```
t_sort = sorted(t)
```

Если же нам нужно отсортировать список по невозрастанию (убыванию), то дополнительно прописывается аргумент `reverse=True`:

```
sorted(t, reverse=True)   # [30.4, 29.5, 27.3, 26.0, 25.6, 23.5]
```

Функции min, max и sorted работают не только с числовыми типами, но и вообще с любыми, где допустимы операторы сравнения больше и меньше. Например, создадим список из символов (по сути, строк):

```
s = list("python")   # ['p', 'y', 't', 'h', 'o', 'n']
```

И применим все те же самые функции. Получим:

```
max(s)     # 'y'
min(s)     # 'h'
sorted(s)  # ['h', 'n', 'o', 'p', 't', 'y']
```

3.3.3. Базовые операции со списками

При работе со списками часто используются следующие операции:

- + – соединение двух списков в один;
- * – дублирование значений списка;
- in – проверка вхождения элемента в список;
- del – удаление элемента списка.

Например, команда:

```
[1, 2, 3] + [4, 5]   # [1, 2, 3, 4, 5]
```

создаст новый список со значениями обоих списков. В этом легко убедиться, записав следующие строчки:

```
a = [1, 2, 3]
```

```
b = a + [4, 5]
```

После их выполнения список `a` останется неизменным, а список `b` будет содержать элементы обоих списков.

Однако, обратите внимание, что следующая команда работать не будет:

```
[1, 2, 3] + 4   # TypeError
```

так как операция «+» объединяет именно списки между собой. Указывать число или какой-либо другой тип данных недопустимо. В данном случае правильно будет запись вида:

```
[1, 2, 3] + [4]
```

или, так:

```
[1, 2, 3] + [True]
```

Как видите, использовать в списках можно самые разные типы данных.

Следующая операция «*» выполняет дублирование элементов списка на указанное число раз. Например, команда:

```
["Я", "люблю", "Python"] * 3
```

сформирует на выходе результат:

```
['Я', 'люблю', 'Python', 'Я', 'люблю', 'Python', 'Я', 'люблю', 'Python']
```

В операции «*» можно прописывать только целое число (или переменную, ссылающуюся на целое значение). Использовать дробные значения недопустимо:

```
["Я", "люблю", "Python"] * 3.5   # TypeError
```

На практике можно комбинировать операции и записывать, например, такие конструкции:

```
["Я"] + ["люблю"] * 3 + ["Python"]
```

Получим список:

```
['Я', 'люблю', 'люблю', 'люблю', 'Python']
```

Все достаточно просто и наглядно. Этим и замечателен язык Python. В нем многое реализуется простыми и понятными методами, в отличие от многих других языков программирования.

Следующий оператор **in** позволяет определять вхождение некоторого значения в список. Используется он следующим образом:

```
lst = ["Москва", 1320, 5.8, True, "Тверь", False]
1320 in lst  # True
120 in lst   # False
```

Если же булевый результат нужно сохранить в какой-либо переменной, то эту команду можно записать так:

```
result = "Москва" in lst   # True
```

Вместо чисел или строк можно использовать список, чтобы узнать является ли он элементом другого списка:

```
[1, 2] in lst   # False
```

В данном случае получим False, но если его добавить:

```
lst = lst + [[1, 2]]
```

то команда:

```
[1, 2] in lst   # True
```

вернет True. И так со всеми типами данных.

Последний оператор **del** выполняет удаление элемента списка по его индексу, например, так:

```
del lst[2]   # ['Москва', 1320, True, 'Тверь', False, [1, 2]]
```

при этом индексы оставшихся элементов по прежнему идут по порядку от 0 и до конца списка.

Обратите внимание, что del – это оператор, а не операция. Оператор – это завершенная, самостоятельная конструкция языка Python, которая не может входить в состав другого оператора. В частности, из-за этого команда:

```
res = del lst[2]   # SyntaxError
```

приведет к синтаксической ошибке. Здесь оператор del удаляет элемент из списка lst, меняя его содержимое. А вот операции, например, такие как сложение, умножение и многие другие, могут объединяться в рамках одного оператора, формируя более сложные конструкции. Например:

```
per = 2 * (width + height)  # вычисление периметра
```

Здесь записаны сразу две операции «*» и «+». В этом ключевое различие между понятиями операция и оператор.

3.3.4. Срезы списков

О срезах мы с вами уже говорили, когда рассматривали строки и с их помощью выделяли наборы символов из строк. Со списками все работает аналогично: срезы позволяют выделять наборы элементов. Например, имеется список:

```
lst = ["Москва", "Уфа", "Тверь", "Казань"]
```

и требуется выбрать из него 2-й и 3-й элементы. Это можно сделать, так:

```
result = lst[1:3]    # ['Уфа', 'Тверь']
```

Давайте разберемся, как это работает. Синтаксис для срезов имеет вид:

список[start:stop]

В данном случае, мы указываем стартовый индекс 1 (второй элемент) и конечный 3 (последний элемент, до которого выделяются элементы среза, не включая его). На выходе получаем новый список, состоящий из двух элементов:

```
['Уфа', 'Тверь']
```

Обратите внимание, что result – это именно новый, независимый от lst список, содержащий выделенные элементы. В этом легко убедиться, изменив первый элемент списка result:

```
result[0] = "Воронеж"
```

И это никак не влияет на значения исходного списка lst.

Какие вариации записи срезов существуют в Python? Смотрите, в них можно указывать, например, только начальный индекс:

```
lst[2:]   # ['Тверь', 'Казань']
```

тогда будут выбраны все элементы, начиная с 3-го (индекс 2), и до конца списка. Если же указать только конечный индекс:

```
lst[:3]   # ['Москва', 'Уфа', 'Тверь']
```

то возвращаются все элементы с первого и до четвертого (не включая последний). Этот пример еще раз показывает, что последний элемент среза всегда исключается. Если же не указывать ни первый, ни последний индексы, то получаем копию исходного списка:

```
cities = lst[:]   # копия списка lst
```

В этом так же легко убедиться, сравнивая их идентификаторы (id) между собой:

```
id(lst)        # 2512811179200 (у вас будет другое число)
id(cities)     # 2512811161984
```

Различие идентификаторов означает разные объекты, в данном случае, разные списки в памяти устройства.

Еще один вариант создания копии списка – это использование функции list, о которой мы ранее уже говорили:

```
c = list(lst)  # создание копии списка lst
```

Какой именно вариант использовать для копирования списка, решает сам программист, в зависимости от удобства и поставленной задачи. Чаще всего используется синтаксис срезов, т.к. это более короткая запись.

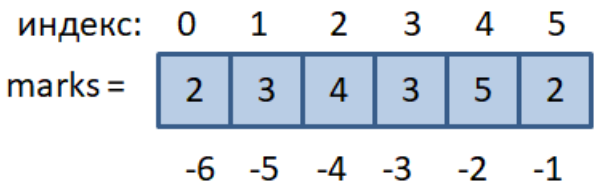
А что будет, если мы один список присвоим другому? Следующим образом:

```
d = lst
```

будет ли здесь создаваться копия списка? Вспоминая занятие по переменным, мы говорили, что переменные – это ссылки на объекты и оператор присваивания лишь копирует ссылку, но не сам объект. Поэтому переменные d и lst будут ссылаться на один и тот же список, их id будут равны между собой:

```
id(d)    # 2512811179200 (у вас будет другое число)
id(lst)  # 2512811179200
```

Помимо положительной индексации в списках можно использовать и отрицательные индексы.



Эти индексы тоже можно записывать в срезах. Например, создадим список из оценок (целых чисел):

```
marks = [2, 3, 4, 3, 5, 2]
```

и запишем следующие срезы:

```
marks[2:-1]   # [4, 3, 5]
marks[-3:-1]  # [3, 5]
```

Смотрите, здесь -1 означает последний элемент списка, а -3 – третий элемент с конца списка. Получаем на выходе соответствующие срезы.

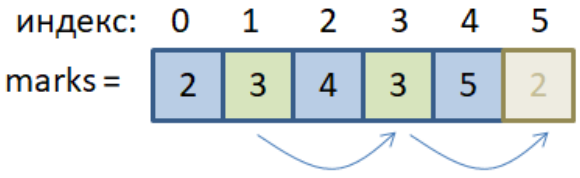
Дополнительно в срезах можно указывать шаг перебора элементов, согласно синтаксису:

список[start:stop:step]

Например:

```
marks[1:5:2]   # [3, 3]
```

Здесь выделяются элементы, начиная со 2-го (индекс 1), и до 6-го (не включая его) с шагом 2, т.е. через элемент. Получаем на выходе два значения 3 и 3:



или без указания левой и/или правой границ:

```
marks[:5:2]  # [2, 4, 5]
marks[1::2]  # [3, 3, 2]
marks[::2]   # [2, 4, 5]
```

Если шаг – отрицательное число, то перебор элементов осуществляется с конца списка, например:

```
marks[::-1]  # [2, 5, 3, 4, 3, 2] (все элементы в обратном порядке)
marks[::-2]  # [2, 3, 3] (то же самое, но через элемент)
```

Вообще, механизм срезов работает абсолютно так же как и со строками, только здесь они применяются к спискам. Но, учитывая, что списки относятся к изменяемым типам данных, то со срезами можно выполнять одну дополнительную операцию – изменение группы элементов. Например, для списка marks требуется 3-ю и 4-ю оценки представить в виде строк. Это можно сделать следующим образом:

```
marks[2:4] = ["хорошо", "удовлетв."]
```

Теперь коллекция содержит данные:

```
[2, 3, 'хорошо', 'удовлетв.', 5, 2]
```

Видите, как легко и просто это можно сделать. Или, другой пример, перебрать все элементы через один и присвоить им 0:

```
marks[::2] = [0, 0, 0]
```

Правда, такая конструкция содержит потенциальную ошибку. Если увеличить размер списка:

```
marks += [3]
```

и повторить операцию:

```
marks[::2] = [0, 0, 0]
```

то получим ошибку, так как число замен здесь уже четыре, а не три. Поэтому, в таких присваиваниях лучше явно указывать границы срезов:

```
marks[:5:2] = [0, 0, 0]
```

Либо не указывать вовсе:

```
marks[:] = [0, 0, 0]     # marks = [0, 0, 0]
```

Тогда произойдет полная замена элементов списка marks присваиваемыми значениями с изменением их количества (будет ровно три нуля). Этот прием удобно использовать, когда нужно заменить все элементы текущего списка значениями из другой коллекции.

Также для группового присваивания можно использовать следующий синтаксис:

```
marks[2:4] = 10, 20
```

3.3.5. Сравнение списков

Списки можно сравнивать между собой с помощью операций:

>, <, ==, !=

Используются они очевидным образом. Например:

```
[1, 2, 3] == [1, 2, 3]    # True  (сравнение на равенство)
[1, 2, 3] != [1, 2, 3]    # False (сравнение на неравенство)
[1, 2, 3] > [1, 2, 3]     # False (сравнение на больше)
```

Последняя операция вернет False, т.к. списки равны, но если записать разные списки:

```
[10, 2, 3] > [1, 2, 3]    # True
[1, 2, 30] > [1, 2, 3, 4] # True
[1, 2, 3, 4] > [1, 2, 3]  # True
```

то первый список будет больше второго. Почему так? Сравнение на больше или меньше выполняется по тому же принципу, что и у строк. Элементы списков сравниваются попарно (от первой пары и до последней) и если элемент из пары первого списка больше соответствующего элемента второго списка, то первый список больше второго. Если же все соответствующие элементы равны между собой, то большим считается более длинный список.

Аналогично выполняется сравнение на меньше:

```
[1, 20, 3] < [1, 2, 3]   # False
[1, 2, 3] < [1, 2, 3, 4] # True
[1, 2, 3] < [1, 2]       # False
```

Все эти сравнения работают только с данными, которые можно между собой сравнивать на больше и меньше. Например, следующая команда отработает без ошибок:

```
[1, 2, "abc"] > [1, 2, "abc"]  # False
```

А если взять два таких списка:

```
[1, 2, 3] > [1, 2, "abc"]
```

то произойдет ошибка, т.к. число 3 не может быть сравнено на больше со строкой «abc».

3.3.6. Методы списков

Список в Python – это один из типов объектов. И каждый такой объект имеет свой набор методов. Ранее мы с вами уже рассматривали методы строк. Здесь же речь пойдет об основных методах списков.

Давайте предположим, что имеется список, содержащий целые числа:

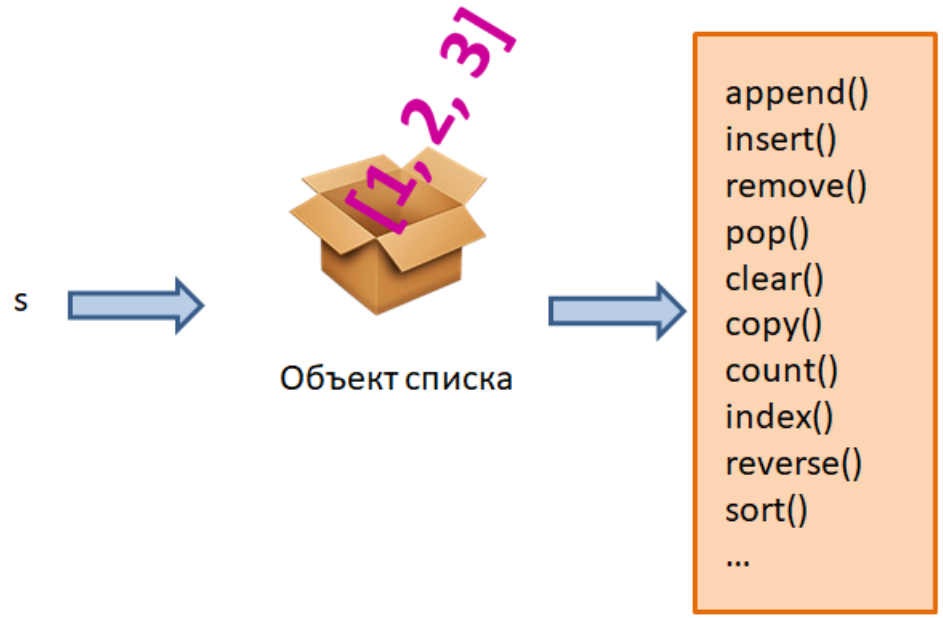
```
a = [1, -54, 3, 23, 43, -45, 0]
```

и требуется в конец этого списка добавить еще одно значение. Это можно сделать с помощью метода:

```
a.append(100)
```

Получим результат:

```
[1, -54, 3, 23, 43, -45, 0, 100]
```



Обратите внимание, что метод **append** ничего не возвращает, а меняет непосредственно сам список. Поэтому писать конструкцию вида:

```
a = a.append(100)  # a = None
```

не следует, так как это приведет к тому, что переменная `a` станет равна None и перестанет ссылаться на список.

Учитывая, что список может содержать самые разные данные, то в методе append можно прописывать не только числа, но, например, и строку:

```
a.append("hello")  # [1, -54, 3, 23, 43, -45, 0, 100, 'hello']
```

тогда в конец списка будет добавлен этот элемент. Или, булево значение:

```
a.append(True)
```

Или еще один список:

```
a.append([1,2,3])
```

Получим итоговый результат:

```
[1, -54, 3, 23, 43, -45, 0, 100, 'hello', True, [1, 2, 3]]
```

Обратите внимание, что список [1, 2, 3] добавляется в виде одного отдельного

элемента, как вложенный список.

В методе append допустимо указывать только какое-либо одно добавляемое значение. Например, следующая команда работать не будет:

```
a.append(1, 2)  # TypeError
```

Если нужно добавить в конец списка сразу несколько значений, то это можно сделать так:

```
a = a + [1, 2]  # [1, -54, …, 'hello', True, [1, 2, 3], 1, 2]
```

Или, используя метод **extend**:

```
a.extend([1, 2]) # аналогично команде a = a + [1, 2]
```

Для вставки нового элемента в произвольную позицию списка используется метод **insert**. Например, команда:

```
a = [1, 2, 3, 4, 5, 6]
a.insert(3, -1000)      # [1, 2, 3, -1000, 4, 5, 6]
```

вставит значение -1000 по индексу 3. При указании индекса за пределами списка, вставка будет выполняться в конец:

```
a.insert(30, -100)   # [1, 2, 3, -1000, 4, 5, 6, -100]
```

Следующий метод **remove** удаляет элемент по значению:

```
a = [1, 2, 3, 4, True, 'hello', 7]
a.remove(True)     # [2, 3, 4, True, 'hello', 7]
a.remove('hello')  # [2, 3, 4, True, 7]
```

Обратите внимание, что при удалении булевого значения True было удалено первое целочисленное значение 1. Дело в том, что в Python True связано с целочисленным значением 1 и некоторые методы считают их идентичными. Это следует учитывать, при написании программ. По аналогии, булево значение False связано с целочисленным значением 0. Эти два элемента False/True могут интерпретироваться, как целочисленные значения 0 и 1.

Если же в методе remove указывается несуществующий элемент, то возникает ошибка:

```
a.remove('hello2')  # ValueError
```

Еще один похожий метод **pop** по умолчанию удаляет последний элемент списка и возвращает его значение. Например:

```
a = [1, 2, 3, 4, True, 'hello', 7]
end = a.pop()
```

В результате список будет иметь вид:

```
[1, 2, 3, 4, True, 'hello']
```

a переменная end ссылаться на значение 7. Дополнительно в методе pop можно указывать индекс удаляемого элемента, например:

```
res = a.pop(3)  # res = 4;  a = [1, 2, 3, True, 'hello']
```

Чтобы очистить весь список, то есть, удалить все элементы можно воспользоваться методом **clear**:

```
a.clear()  # a = []
```

Список a становится пустым.

Следующий метод **copy** возвращает копию списка. Например:

```
a = [1, -54, 3, 23, 43, -45, 0]
c = a.copy()
```

Здесь список c является копией списка a. Это эквивалентно команде:

```
c = list(a)
```

Обратите внимание, что оба подхода создают копию только указанного списка. Если в нем будут присутствовать вложенные списки, то их копии создаваться не будут. Это касается всех вложенных в список изменяемых типов данных. Поэтому метод copy создает **поверхностную** копию списка. Отдельно отмечу, что глубокая копия может быть создана функцией deepcopy стандартного модуля copy, например, следующим образом:

```
from copy import deepcopy

data = [1, 2, [5, 6], True, "abc"]
data_cpy = deepcopy(data)  # создание глубокой (полной копии списка)
```

Функция deepcopy рекурсивно перебирает все элементы списка и создает копии каждого изменяемого объекта данных. Подробно о глубоком копировании разных типов данных смотрите в курсе по стандартной библиотеке языка Python:

https://stepik.org/a/259466

Следующий метод **count** позволяет найти число элементов с указанным значением. Например:

```
marks = [2, 3, 4, 3, 5, 2]

res1 = marks.count(3)   # 2
res2 = marks.count(5)   # 1
res3 = marks.count(7)   # 0
```

Если же нам нужен индекс определенного значения, то для этого можно воспользоваться методом **index** следующим образом:

```
indx = marks.index(3)   # 1
indx = marks.index(5)   # 4
```

Обратите внимание, что метод index возвращает индекс первого найденного элемента в списке. Однако дополнительно можно указать стартовое значение для поиска, например:

```
marks.index(3, 2)    # 3
```

В этом случае поиск значения 3 будет начинаться с индекса 2. Третьим аргументом можно указывать граничный индекс для поиска элемента. Например:

```
marks.index(3, 2, 4) # 3
```

Выполняется поиск числа 3 со 2-го индекса по 4-й, не включая его.

Если элемент не находится:

```
marks.index(7, 2, 4) # ValueError
```

то возникает ошибка. Чтобы такая ошибка не возникала, предварительно можно проверить существование элемента в интервале поиска:

```
7 in marks[2:4]   # False
```

И при значении True, затем, уже определять индекс этого элемента.

Следующий метод **reverse**:

```
marks.reverse()  # [2, 5, 3, 4, 3, 2]
```

меняет порядок следования элементов на обратный. Причем делает это непосредственно для указанного списка и ничего не возвращает. Поэтому записывать эту команду в виде:

```
marks = marks.reverse()  # marks = None
```

не следует, т.к. это приведет лишь к изменению значения переменной marks равной None.

Последний метод, который мы здесь рассмотрим – это метод **sort** для сортировки элементов списка. Делается это следующим образом:

```
marks.sort()   # [2, 2, 3, 3, 4, 5]
```

В результате список marks будет отсортирован по возрастанию (не убыванию) значений элементов. Обратите внимание, что метод sort меняет сам список и ничего не возвращает.

Если же требуется сортировка по убыванию (не возрастанию), то дополнительно в методе sort прописывается аргумент reverse со значением True:

```
marks.sort(reverse=True)  # [5, 4, 3, 3, 2, 2]
```

Метод sort работает не только с числами, но и с любыми типами данных, которые можно сравнивать на больше и меньше. Например, можно выполнять сортировку

списка строк:

```
lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]
lst.sort()   # ['Казань', 'Москва', 'Санкт-Петербург', 'Тверь']
```

Здесь при сортировке используется лексикографическое сравнение, о котором мы говорили, когда рассматривали строки.

Ранее была рассмотрена аналогичная функция sorted, которая также выполняет сортировку различных упорядоченных коллекций. И ее вполне можно применить к спискам. Различие между ними в том, что метод sort меняет непосредственно сам список, а функция sorted возвращает новый отсортированный список, не меняя начальный. Во всем остальном их функционал совпадает.

Итак, мы с вами рассмотрели следующие методы списков, которые следует запомнить и применять в своих программах по мере необходимости:

Метод	Описание
append()	Добавляет элемент в конец списка
extend()	Добавляет несколько элементов в конец списка
insert()	Вставляет элемент в указанное место списка
remove()	Удаляет элемент по значению
pop()	Удаляет последний элемент, либо элемент с указанным индексом
clear()	Очищает список (удаляет все элементы)
copy()	Возвращает копию списка
count()	Возвращает число элементов с указанным значением
index()	Возвращает индекс первого найденного элемента
reverse()	Меняет порядок следования элементов на обратный
sort()	Сортирует элементы списка

3.3.6. Вложенные списки

Мы уже знаем, что список в качестве своих элементов может содержать другие списки. Такая конструкция называется **вложенным списком** и повсеместно используется в программах на Python. Давайте детальнее посмотрим, как формируются такие вложения и как с ними работать. Но вначале ответим на вопрос зачем они нужны и что дают?

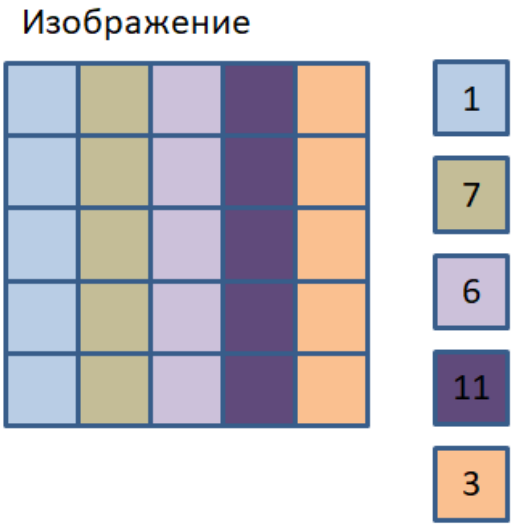
Предположим, что в программе необходимо хранить изображение. Для примера оно представлено на рисунке с размерами 5 на 5 пикселей. Каждый цвет описывается (кодируется) своим уникальным числом. Условно они здесь обозначены числами 1, 7, 6, 11 и 3. Так как изображение – это двумерная таблица чисел, то для ее представления в программе можно воспользоваться двумерным списком 5x5 элементов. Сформировать его можно следующим образом. Одна строка приведенного на рисунке изображения, очевидно, может быть описана одномерным списком:

```
line = [1, 7, 6, 11, 3]
```

Так как нам нужно хранить пять таких строк, то приходим к определению следующего двумерного вложенного списка:

```
img = [[1, 7, 6, 11, 3], [1, 7, 6, 11, 3],
       [1, 7, 6, 11, 3], [1, 7, 6, 11, 3],
       [1, 7, 6, 11, 3]]
```

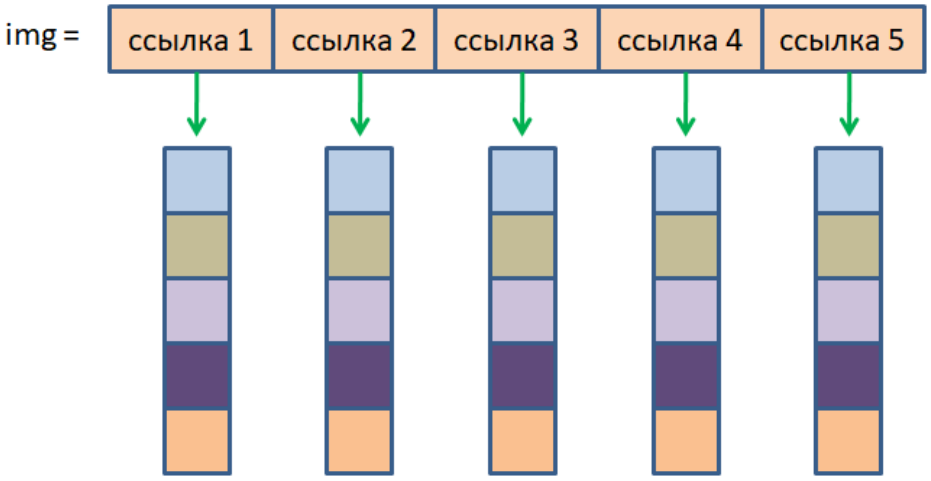
Здесь внутри первого списка записаны пять вложенных списков. Получаем двумерный список.



Учитывая, что все вложенные списки содержат одинаковые значения, то его можно было бы сформировать, используя одномерный список line, следующим образом:

```
img = [line[:], line[:], line[:], line[:], line[:]]
```

В итоге имеем такой же список с независимыми строками.



Последняя запись хорошо показывает структуру представления многомерных данных на уровне списков. Первый, главный список хранит ссылки на вложенные списки. А вложенные списки уже хранят ссылки на соответствующие числа,

представляющие тот или иной цвет. Поэтому, если взять первый элемент главного списка:

```
line1 = img[0]  # [1, 7, 6, 11, 3]
```

то мы получим список, представляющий первую строку (или, первый столбец в зависимости от интерпретации программистом этих данных). А раз это так, то можно записать еще одни квадратные скобки и из этого вложенного списка взять, допустим, второй элемент:

```
pixel = img[0][1]   # 7
```

Или же можно заменить, например, вторую строку на новую, состоящую из нулей:

```
img[1] = [0, 0, 0, 0, 0]
```

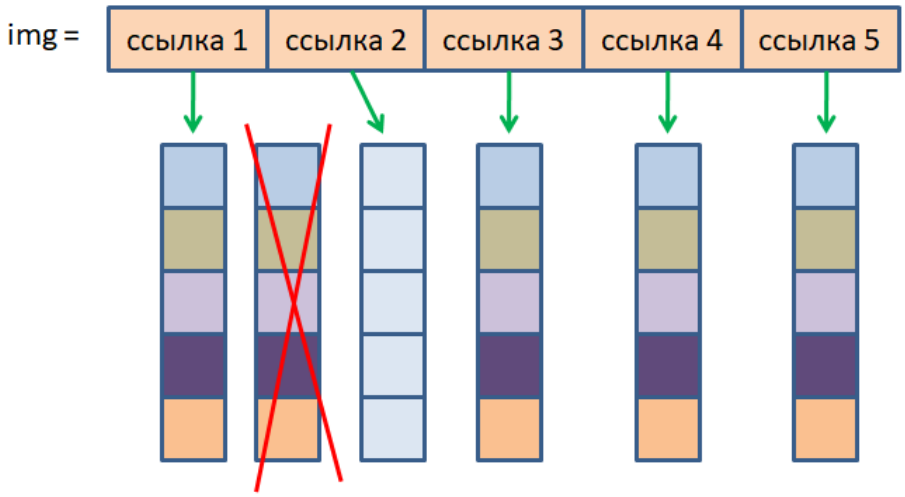
или, то же самое, но в более краткой форме:

```
img[1] = [0] * 5
```

В результате получаем двумерный список вида:

```
[[1, 7, 6, 11, 3], [0, 0, 0, 0, 0], [1, 7, 6, 11, 3],
[1, 7, 6, 11, 3], [1, 7, 6, 11, 3]]
```

Как в деталях это работает? Вначале был сформирован новый объект – список из нулей, с которым была связана вторая ссылка главного списка, а прежний список был автоматически удален сборщиком мусора.



Если бы мы хотели изменить значения уже существующего вложенного списка, то следовало бы обратиться к его элементам, например, через механизм срезов:

```
img[1][:] = [0] * 5
```

и присвоить его элементам новые числовые значения. Видимый результат был бы тем же самым.

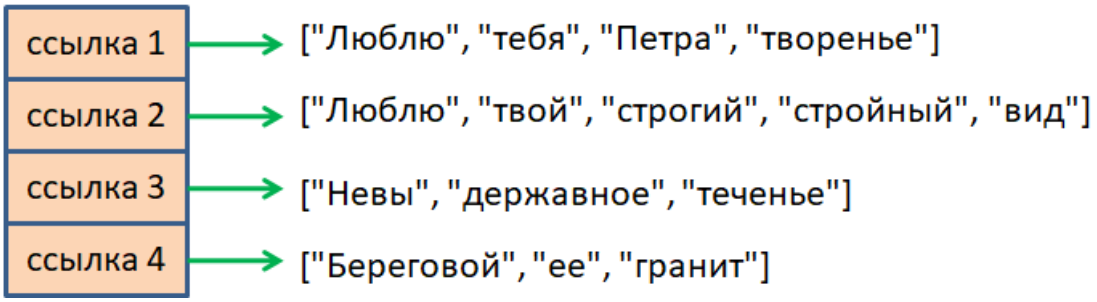
В качестве другого примера с помощью вложенных списков представим строки

известного стихотворения на уровне отдельных слов:

```
Люблю тебя, Петра творенье,
Люблю твой строгий, стройный вид,
Невы державное теченье,
Береговой ее гранит,
```

Здесь в каждой строке разное количество слов, но для вложенных списков – это не проблема. Они могут иметь разное число элементов, имеем:

```
t = [["Люблю", "тебя", "Петра", "творенье"],
     ["Люблю", "твой", "строгий", "стройный", "вид"],
     ["Невы", "державное", "теченье"],
     ["Береговой", "ее", "гранит"]
]
```



В результате получаем следующую структуру наших данных. Основной список содержит ссылки на вложенные списки с разным количеством элементов (слов в строке). Здесь так же для доступа к первой строке достаточно указать первый индекс:

```
line1 = t[0]    # ['Люблю', 'тебя', 'Петра', 'творенье']
```

А для доступа к отдельному слову этой строки – еще один, второй индекс:

```
word = t[0][2]  # 'Петра'
```

Если же нужно изменить какое-либо слово, то это можно сделать следующим образом:

```
t[0][2] = "Питон"
```

В итоге, первая строка примет вид:

```
['Люблю', 'тебя', 'Питон', 'творенье']
```

Мало того, мы можем добавить новую строку, используя известный метод:

```
t.append(["Твоих", "оград", "узор", "чугунный"])
```

Получим:

```
[['Люблю', 'тебя', 'Петра', 'творенье'],
['Люблю', 'твой', 'строгий', 'стройный', 'вид'],
['Невы', 'державное', 'теченье'], ['Береговой', 'ее', 'гранит'],
```

```
['Твоих', 'оград', 'узор', 'чугунный']]
```

Или удалить какой-либо вложенный список, например, второй:

```
del t[1]
```

Имеем:

```
[['Люблю', 'тебя', 'Петра', 'творенье'],
['Невы', 'державное', 'теченье'],
['Береговой', 'ее', 'гранит'],
['Твоих', 'оград', 'узор', 'чугунный']]
```

И так далее, то есть, с вложенными списками можно выполнять все те же операции, что и с обычными данными.

Конечно, уровень вложенности списков может быть произвольным. Например, таким:

```
A = [[[True, False], [1, 2, 3]], ["матрица", "вектор"]]
```

В итоге, первый элемент этого списка:

```
table = A[0]   # [[True, False], [1, 2, 3]]
```

будет двумерным вложенным списком. Соответственно, для доступа к его элементам можно использовать два и три индекса, например:

```
res1 = A[0][1]     # [1, 2, 3]
res2 = A[0][1][1]  # 2
```

Здесь сначала берется первый элемент A[0], затем второй вложенный список A[0][1] и из него выбираем второй элемент A[0][1][1].

А команда:

```
item2 = A[1]   # ['матрица', 'вектор']
```

возвратит обычный одномерный список. Для доступа к отдельным строкам здесь достаточно два индекса, например:

```
w = A[1][0]    # 'матрица'
```

Вот так в Python можно задавать многомерные списки разной структуры и разного уровня вложенности.

ГЛАВА IV. УСЛОВНЫЙ ОПЕРАТОР if. КОНСТРУКЦИЯ if-elif-else

До сих пор все наши программы содержали команды, которые выполнялись строго по порядку одна за другой. Однако часто требуется выполнять определенные блоки команд в зависимости от некоторых условий. В программировании это называется **ветвлением**. А реализация таких ветвлений в Python выполняется с помощью оператора if, о котором пойдет речь в этой главе.

4.1. Условный оператор if

Оператор if называют условным оператором и реже оператором ветвления. Что он делает и для чего нужен? Давайте вспомним, что на одном из прошлых занятий, мы с вами рассматривали операции сравнения, например, проверяли, что четыре больше двух:

```
4 > 2   # True
```

И прописывали аналогичные сравнения с переменными:

```
a = 5
a < 0   # False
```

Но не говорили, как дальше можно использовать эти операции в программе. Пришло время восполнить этот пробел. Так вот, оператор if позволяет выполнять группу команд, при истинности записанного в нем условия. Давайте рассмотрим его работу на конкретном примере.

Предположим, нужно написать программу вычисления модуля числа (отрицательное число делать положительным). Ее можно реализовать следующим образом:

```
x = int(input("Введите целое число:"))

if x < 0:
    x = -x  # команда, выполняющаяся по условию

print(x)    # команда за пределами условного оператора if
```

В этой программе записан оператор if, после которого указано условие «x < 0». Оно будет принимать значение True для отрицательных `x`, и False для неотрицательных `x`. Так вот, оператор if выполняет команды, записанные в нем, только в том случае, если условие принимает значение True. В нашем примере – это строчка:

```
    x = -x
```

А вот следующая строчка:

```
print(x)
```

стоит уже вне этого условия и будет выполняться всегда.

Как Python «понимает», что находится в блоке условного оператора? Все дело в форматировании текста программы. Когда ставится двоеточие после условия, то открывается новый блок, где можно прописывать множество операторов. И все конструкции языка Python, которые имеют отступ (обычно четыре пробела) относительно оператора if, будут располагаться внутри этого блока. Именно поэтому строчка `x = -x` выполняется только при истинности условия `x < 0`, а функция `print(x)` стоит уже после оператора if и выполняется всегда.

После запуска полученной программы можно убедиться, что при вводе чисел 5 или -5 на экране будет отображаться значение 5 – модуль введенного числа.

Если функцию print записать с начальными пробелами, т.е. поместить внутрь условного оператора:

```
x = int(input("Введите целое число:"))

if x < 0:
    x = -x   # команда, выполняющаяся по условию
    print(x) # команда, выполняющаяся по условию
```

то она будет срабатывать только при вводе отрицательных чисел. Это происходит из-за того, что функция print теперь находится внутри блока оператора if и выполняется только при истинности условия. Этот пример показывает, что в Python форматирование текста программы имеет ключевое значение. Благодаря этому программы становятся более наглядными и читаемыми, а также заставляют начинающих программистов правильно подходить к их оформлению. Это большой плюс данного языка программирования.

Для большей ясности рассмотрим еще пару примеров. Предположим, требуется поменять значения двух переменных, но только в том случае, если a меньше b:

```
a = float(input("a: "))
b = float(input("b: "))

if a < b:
    a, b = b, a

print(a, b)
```

Также в операторе if можно прописывать и составные условия, например, для проверки попадания числа в заданный диапазон:

```
x = int(input())

if x >= -4 and x <= 10:
    print("x в диапазоне [-4; 10]")
```

О составных условиях мы с вами уже ранее говорили, поэтому здесь все должно быть понятно. Напомню только, что приведенное выше условие допускается записывать и в таком виде:

```
x = int(input())

if -4 <= x <= 10:
    print("x в диапазоне [-4; 10]")
```

Как видите, в качестве условия оператора if можно записывать любые конструкции, которые можно интерпретировать как истину (True) и ложь (False). Даже, просто указав числовое значение:

```
x = int(input())

if x:
    print("x = True")
```

Это условие будет истинно для любых не нулевых значений x, т.к. не ноль – это истина (True). И, конечно же, в условиях можно явно прописывать булево значение, например:

```
if True:
    print("True")
```

Разумеется это всего лишь пример для лучшего понимания условных конструкций внутри оператора if. Практический смысл записывать булевы значения True/False возникает крайне редко.

Также в операторе if допускается использовать различные операции проверки для списков, которые мы уже с вами проходили. Например, пусть имеется список с оценками студента:

```
marks = [3, 3, 4, 2, 4]
```

и требуется узнать, будет ли он отчислен (имеет хотя бы одну двойку). Запишем для этого следующую проверку:

```
if 2 in marks:
    print("студент будет отчислен")
```

4.2. Конструкция if-else условного оператора if

В последнем приведенном примере нам бы хотелось дополнительно выдать сообщение, что студент успешно сдал сессию, если не имеет ни одной двойки. Это лучше всего реализовать с помощью необязательного блока else, который можно прописывать после оператора if, например, так:

```
marks = [3, 3, 4, 2, 4]

if 2 in marks:
    print("студент будет отчислен")
else:
    print("студент успешно сдал сессию")
```

Если в этой программе срабатывает первое условие в if, то блок else не выполняется. И, наоборот, если условие в if не срабатывает (ложно), то блок else

выполняется. Здесь название оператора else можно перевести как «иначе» и получается либо выполнение первого блока, либо второго, но не обоих вместе. То есть, здесь определены два взаимоисключающих случая. Действительно, студент не может быть одновременно отчисленным и успешно сдавшим сессию (будем полагать, что по собственному желанию он не уходит).

Другой подобный пример:

```
x = int(input())

if x < 0:
    print("x - отрицательное число")
else:
    print("x - неотрицательное число")
```

Очевидно, что x не может быть одновременно отрицательным и неотрицательным, поэтому для проверки такого условия можно использовать оператор else. То есть, для реализации любых взаимоисключающих условий можно (и целесообразно) использовать конструкцию if-else.

Еще один пример проверки на четность введенного числа:

```
x = int(input())

if x % 2 == 0:
    print("x - четное число")
else:
    print("x - нечетное число")
```

Здесь также число не может быть одновременно четным и нечетным, поэтому вполне логично воспользоваться конструкцией if-else.

4.3. Вложенные условия. Конструкция if-elif-else

Внутри условного оператора if можно прописывать любые конструкции языка Python, в том числе и другие условия. Давайте посмотрим на примерах, как это делается и как это работает.



Предположим, что требуется написать программу, в которой для четных чисел нужно дополнительно проверить состоит ли оно из одной цифры или содержит

несколько цифр. Такую программу можно реализовать с использованием вложенных условий следующим образом:

```
x = int(input())

if x % 2 == 0:
    if 0 <= x <= 9:
        print("x – четное число из одной цифры")
    else:
        print("x - четное число из нескольких цифр")
```

В первом условии проверяется, что x – четное число, а во вложенном, что это цифра или несколько цифр. Обратите внимание здесь на оператор else. Он относится строго ко второму (вложенному) условию, так как записан на одном уровне с ним. А вот, если бы мы его прописали без отступов:

```
if x % 2 == 0:
    if 0 <= x <= 9:
        print("x – четное число из одной цифры")
else:
    print("x - четное число из нескольких цифр")
```

то этот блок стал бы относиться к первому условию.

Вернем блок else на прежнюю позицию, и добавим еще один такой блок для первого условия, получим:

```
x = int(input())

if x % 2 == 0:
    if 0 <= x <= 9:
        print("x – четное число из одной цифры")
    else:
        print("x - четное число из нескольких цифр")
else:
    print("x - нечетное число")
```

Теперь обрабатываются все возможные исходы для числа x.

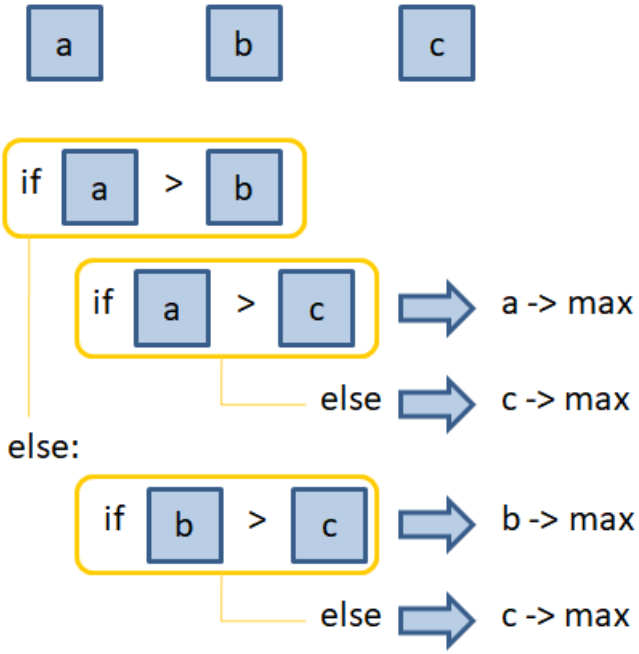
Следующий пример, который мы рассмотрим, это поиск наибольшего значения среди трех введенных чисел a, b и c:

```
a = int(input("a: "))
b = int(input("b: "))
c = int(input("c: "))

if a > b:
    if a > c:
        print("a -> max")
    else:
        print("c -> max")
else:
    if b > c:
```

```
        print("b -> max")
    else:
        print("c -> max")
```

Логика работы программы подробно расписана на рисунке ниже. Предлагаю вам самостоятельно в ней разобраться. Здесь ничего сложного нет.



Приведенные примеры хорошо демонстрируют ценность и порядок работы вложенных условий. Конечно, степень вложенности может быть любой, но в практике программирования лучше избегать больших вложений. Нормальным считается не более трех. Если у вас выходит больше, то, скорее всего, это признак неверно спроектированной архитектуры программы.

4.3.1. Оператор elif

Помимо блока else условный оператор if может содержать еще один или несколько блоков elif. Что это за блок и зачем он нужен? Давайте, для примера, представим, что в программе пользователь выбирает один из пунктов меню:

1. Курс по Python
2. Курс по C++
3. Курс по Java
4. Курс по JavaScript

Используя текущие знания, мы можем реализовать это так:

```
item = int(input())

if item == 1:
    print(“Выбран курс по Python”)
else:
    if item == 2:
        print(“Выбран курс по C++”)
    else:
```

```
        if item == 3:
            print(“Выбран курс по Java”)
        else:
            if item == 4:
                print(“Выбран курс по JavaScript”)
            else:
                print(“Неверный пункт”)
```

Однако можно сделать проще и нагляднее, используя блок условного оператора:

elif = else if

То есть, все конструкции с else if предыдущего примера мы можем заменить на блоки elif следующим образом:

```
if item == 1:
    print("Выбран курс по Python")
elif item == 2:
    print("Выбран курс по C++")
elif item == 3:
    print("Выбран курс по Java")
elif item == 4:
    print("Выбран курс по JavaScript")
else:
    print("Неверный пункт")
```

Поэтому, когда в программе нужно реализовать выбор какого-то одного значения из множества доступных, хорошо подходит блок операторов:

if – elif – else

Причем они должны записываться именно в таком порядке. После if можно пропускать любой из блоков (или оба), но порядок следования всегда должен быть именно таким.

Конечно, помимо проверок на равенство после elif допускается прописывать любые виды условий. Например, с клавиатуры вводится целое положительное x (до трехзначного) и для него требуется определить количество десятков:

```
x = int(input())

if x < 0:
    print("x должно быть положительным")
elif 0 <= x <= 9:
    print("x - цифра")
elif 10 <= x <= 99:
    print("x - двузначное число")
elif 100 <= x <= 999:
    print("x - трехзначное число")
else:
    print("x содержит более трех цифр")
```

Вот принцип работы условного оператора if и конструкций if-else и if-elif-else,

которые постоянно используются в программах на языке Python, реализуя заданную логику.

4.4. Тернарная условная операция

В версии Python 2.5 появилась еще одна конструкция ветвления под названием тернарная условная операция. Обратите внимание, что это операция, а не оператор, то есть, она может входить в состав других операторов.

В самом простом варианте синтаксис тернарной операции можно записать в виде:

<значение 1> if <условие> else <значение 2>

Здесь возвращается «значение 1», если условие истинно, а иначе – «значение 2».

Давайте рассмотрим работу этой операции на конкретных примерах. Предположим, в программе объявлены две переменные:

```
a = 12
b = 7
```

и требуется определить максимальное среди них. Используя имеющиеся знания, это можно было бы сделать так:

```
if a > b:
    res = a
else:
    res = b

print(res)
```

Но тернарная операция приводит к более элегантной команде:

```
res = a if a > b else b
```

В результате переменная `res` будет принимать значение `a`, если `a` больше `b`, а иначе значение `b`. Видите, какая простая, понятная и компактная запись в итоге получилась! Это одно из удобств данной операции. Она здесь записана с оператором присваивания и образует с ним одну единую команду. С условным оператором if такой подход был бы невозможен, т.к. два оператора нельзя объединять в одну синтаксическую единицу. То есть, если бы мы попытались записать что-то вроде:

```
d = if a > b: ...
```

то попросту возникла бы синтаксическая ошибка, так делать нельзя.

Из последнего примера использования тернарной операции видно, что она автоматически возвращает вычисленный результат. В приведенном примере – это переменная `a` или переменная `b`. Тогда как обычный условный оператор if предназначен для выполнения блока кода по определенному условию и сам по себе не возвращает никаких значений.

Следующее важное отличие тернарной операции в том, что в ней не может быть внутренних блоков с набором нескольких операторов. Вместо переменных `a` и `b` можно прописывать только одну какую-либо конструкцию, которая также является операцией. Часто это некое конкретное значение, или результат работы операций, например, арифметических:

```
res = a + 2 if a > b else b - 5
```

Также допустимо использовать какие-либо функции:

```
a = -12
b = -7
res = abs(a) if a > b else abs(b)
```

То есть, здесь может быть любая операция языка Python, но только одна. По идее, можно даже записать функцию print:

```
res = print(a) if a > b else print(b)
```

Но тогда переменная res будет принимать значение None, так как функция print ничего не возвращает (точнее, значение None).

Рассмотрим еще один пример. Пусть имеется некая строка и способ преобразования этой строки:

```
s = 'python'
t = 'upper'
```

Пусть 'upper' означает преобразование строки s в верхний регистр. Сделаем это через тернарный оператор следующим образом:

```
res = s.upper() if t == 'upper' else s
```

Видите, как элементарно и просто реализуется эта идея? В этом большой плюс тернарной операции. Вообще, ее можно рассматривать, как некий активный объект, который возвращает определенное значение в зависимости от условия. Благодаря этому, ее не редко прописывают внутри различных конструкций, например, так:

```
lst = [1, 2, a if a < b else b, 4, 5]
s = "a - " + ("четное" if a % 2 == 0 else "нечетное") + " число"
```

Обратите внимание на круглые скобки во втором примере. Они необходимы, чтобы следующая операция «+» применялась к результирующей строке, а не к тернарной операции. Если круглые скобки убрать, то последний фрагмент «число» будет добавляться к строке «нечетное» при срабатывании else.

Еще один пример использования тернарной операции в виде аргумента функции:

```
res = max(1, 5, a if a > 0 else b, 4, 5)
```

Все это показывает гибкость использования данной условной операции. Конечно, она не может собой заменить оператор if, т.к. всегда следует помнить об

ограничениях тернарной операции: внутри нее можно записывать только одну конструкцию и эта конструкция должна быть так же операцией, а не оператором.

4.5. Вложенная тернарная операция

Так как внутри тернарной операции можно использовать любые другие операции, то никто не мешает вложить одну тернарную операцию в другую. Получаем конструкцию вида:

<r_1> if <c_1> else <r_2> if <c_2> else <r_3>

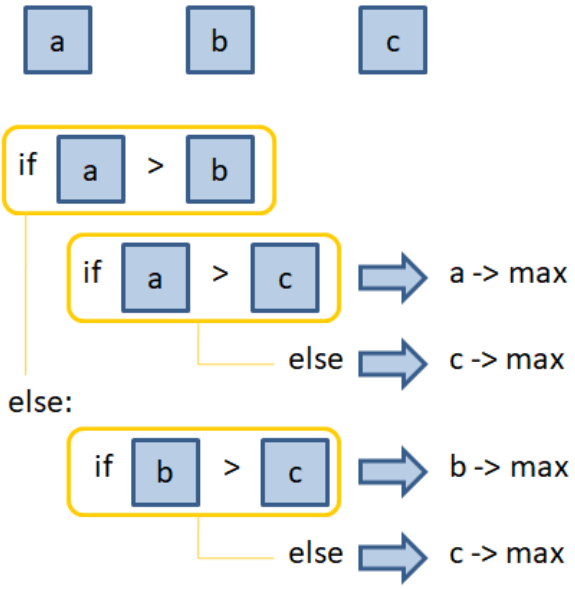
Или еще более разнообразную:

<r_1> if <c_1> else <r_2> if <c_2> else <r_3> if <c_3> else <r_4>

Конечно, использовать их в программах стоит крайне аккуратно и внимательно. Лучше совсем избегать из-за сложного восприятия и понимания таких длинных однострочных записей.

Давайте посмотрим на примере, как реализуются вложения в тернарной операции. Мы с вами выше записывали вложенные условия для определения максимального из трех чисел. Давайте сделаем то же самое, но через тернарные операции. Получим:

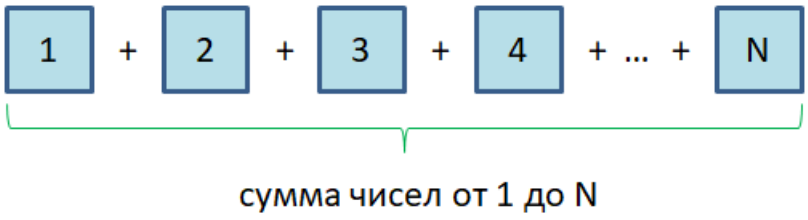
```
a = 2
b = 3
c = -4
d = (a if a > c else c) if a > b else (b if b > c else c)
```



Думаю, общий принцип работы здесь понятен. Сначала проверяется условие `a > b`, затем, в зависимости от результата проверки, выполняется или левая тернарная операция, или правая. Причем, опять же, здесь обязательно вложенные тернарные операции следует заключать в круглые скобки, чтобы тернарная конструкция if-else воспринималась как единое целое внутри вложений.

ГЛАВА V. ЦИКЛЫ, ИТЕРАТОРЫ И ГЕНЕРАТОРЫ СПИСКОВ

В этой главе мы с вами познакомимся с еще одним ключевым элементом компьютерных программ – циклами. Что это такое и зачем они нужны? Давайте представим, что требуется вычислить сумму целых чисел от 1 до N. Причем N может быть сколько угодно большой: тысяча, миллион и так далее. Понятно, что мы не можем здесь просто записать операторы сложения чисел друг за другом для вычисления длинной последовательности. Тем более, что на момент написания программы, N может быть неизвестна. В таких ситуациях нам на помощь, как раз, и приходят циклы.



Также мы с вами рассмотрим внутренний механизм перебора элементов коллекций, основанный на механизме итераторов, и познакомимся с удобным способом формирования списков – генераторами списков или, более точный термин, списковыми включениями (list comprehensions).

5.1. Оператор цикла while

Циклы в программах позволяют реализовывать некие повторяющиеся действия. Например, предположим, что маленькие панды прыгают с горки в течение часа, пока мама-панда не позовет всех к столу кушать.

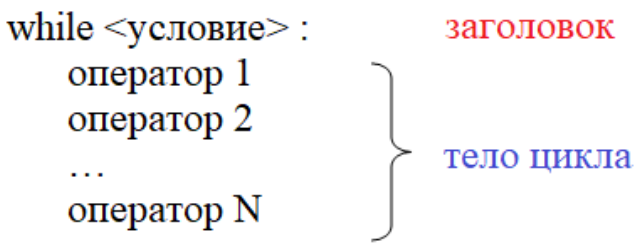


На уровне текста это можно записать, так:

```
цикл (пока не прошел час):
      прыгаем с горки
```

То есть, пока истинно условие, цикл работает, как только условие становится ложным (прошел час), цикл завершается. Ровно так работает цикл **while**, о котором мы сейчас с вами будем говорить.

В самом простом варианте цикл while имеет следующий синтаксис:



Вначале записывается ключевое слово while, затем, условие работы цикла, и ставится двоеточие для определения блока операторов, работающих внутри этого цикла. Такой блок еще называют **телом цикла**, а ключевое слово while с условием – **заголовком цикла**.

Обратите внимание на форматирование при описании цикла while. Здесь так же, как и в условных операторах, набор операторов внутри тела цикла должен иметь отступы относительно ключевого слова while (обычно четыре пробела).

Давайте вернемся к задаче вычисления суммы чисел от 1 до N и посмотрим, как здесь нам поможет цикл while. Вначале определим три вспомогательные переменные: N – значение последнего слагаемого; s – для хранения вычисленной суммы (начальное значение 0); i – значение текущего слагаемого (начинается с 1):

```
N = 1000
s = 0
i = 1
```

Так как сумму нужно вычислять, пока слагаемое i не достигнет значения N, то условие цикла можно определить, следующим образом:

```
while i <= N:
```

А внутри цикла пропишем следующие команды:

```
    s += i
    i += 1
```

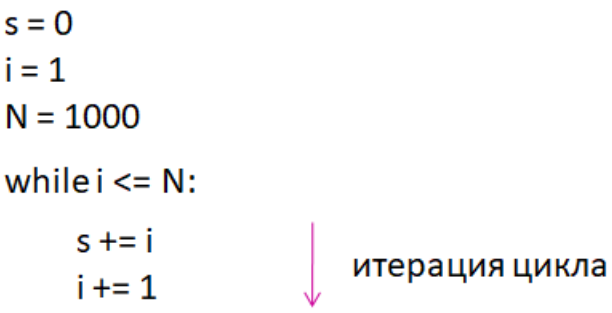
Получаем реализацию цикла:

```
N = 1000
s = 0
i = 1

while i <= N:
    s += i
    i += 1
```

Порядок работы программы следующий. Вначале i равна 1 и эта единица прибавляется к сумме s (команда s += i). После этого i увеличивается на 1 (команда i += 1) и становится равной 2. Затем, выполняется проверка условия цикла (i <= N). На данном этапе оно истинно, поэтому снова переходим внутрь

тела цикла, и к s прибавляется уже значение 2, а i опять увеличивается на 1 и оно становится равным 3. И так до тех пор, пока i не станет больше N. К этому моменту будут просуммированы все числа от 1 до N с результатом в переменной s. Вот принцип работы циклов, причем во всех языках программирования, не только в Python.



Здесь следует запомнить, что однократное выполнение тела цикла в программировании называют **итерацией**. Мы будем часто использовать этот термин в дальнейшем.

В приведенной выше программе было использовано довольно простое условие цикла «i <= N». Спрашивается, какие вообще условия можно прописывать в циклах? В действительности, все те же самые, что и в условных операторах. В том числе и составные. В качестве примера, давайте будем вычислять сумму s пока не дойдем до последнего слагаемого N, но не более 50-ти. Так как цикл работает, пока истинно условие, то такое условие можно записать в виде:

```
N = 1000
s = 0
i = 1

while i <= N and i <= 50:
    s += i
    i += 1
```

Здесь цикл будет работать, пока оба подусловия (i <= N) и (i <= 50) будут истинны. Как только хотя бы одно из подусловий оказывается ложным, то все составное условие становится ложным и цикл завершает свою работу. В результате i достигнет или N, или 50. Вот это нужно хорошо усвоить: в циклах прописываются условия их работы, а не завершения.

Используя этот же цикл, мы легко можем поменять условие задачи и вычислить сумму чисел, которые меняются через один:

1, 3, 5, 7, …

Для этого достаточно счетчик i увеличивать не на 1, а сразу на два:

```
while i <= N and i <= 50:
    s += i
    i += 2
```

При этом условие цикла можно оставить прежним. Он завершится, как только i превысит N или 50. Здесь уже нет гарантии, что последнее слагаемое i будет строго равно N или 50-ти, но это и не требуется, так как цикл завершится при превышении одного из этих значений.

Конечно, внутри тела цикла можно записывать любые операторы, в том числе и функцию print. Давайте, например, отобразим в консоли цифры от 0 до 9:

```
i = 0

while i < 10:
    print(i)
    i += 1
```

Здесь прописано условие (i < 10), а не (i <= 9). Хотя, можно использовать любой вариант. Все зависит от предпочтений самого программиста.

Если нам нужно реализовать убывающую последовательность чисел, например:

-1, -2, -3, …, -N

то это делается аналогично, только с уменьшением счетчика:

```
N = -10
i = -1

while i >= N:
    print(i)
    i -= 1
```

Вообще, счетчик в цикле можно менять произвольным образом, например, умножая на два:

```
    i *= 2
```

или деля на три:

```
    i //= 3
```

Здесь нет никаких ограничений. Все зависит от логики реализации поставленной задачи.

Следующим примером рассмотрим задачу, когда пользователь вводит пароль, до тех пор, пока не введет верно. Это можно сделать следующим образом:

```
pass_true = 'password'
ps = ''

while ps != pass_true:
    ps = input("Введите пароль: ")

print("Вход в систему осуществлен")
```

Первыми двумя строчками задаются истинный пароль pass_true и тот, который

вводит пользователь ps. Затем, идет цикл, который будет работать до тех пор, пока истинный пароль не совпадет с введенным. Обратите внимание на условие цикла: «пока пароли не совпадают». Опять же, всегда следует помнить, что мы прописываем условие работы, а не остановки цикла. Поэтому здесь нужно проверять на несовпадение паролей и запрашивать у пользователя пароль, пока они не совпадут.

Еще один пример использования условного оператора if внутри цикла while. Предположим, нужно вывести на экран все числа кратные 3, которые встречаются при переборе целых значений от 1 до N. Программу можно записать следующим образом:

```
N = 20
i = 1

while i <= N:
    if i % 3 == 0:
        print(i)

    i += 1
```

Как видите, все достаточно просто и очевидно. Здесь главное не забывать о правильном форматировании текста программы: функция print находится внутри условия if, поэтому перед ней необходимы отступы. А сам блок условия имеет отступы относительно оператора while. Так определяются вложенные блоки операторов в языке Python.

5.2. Операторы break, continue и else

Внутри тела циклов можно использовать операторы break и continue. Суть этих операторов в следующем:

- break – выполняет досрочное завершение цикла;
- continue – пропускает одну итерацию цикла.

5.2.1. Оператор break

Вначале рассмотрим работу оператора break. В программе ниже условие цикла записано со значением True:

```
print("запуск цикла")

i = 0
while True:
    i += 1

print("завершение цикла")
```

В результате такой цикл будет работать «вечно», он сам никогда не остановится и строчку «завершение цикла» мы не увидим. Программу в этом случае можно только прервать (например, комбинацией клавиш Ctrl+C) и полностью

остановить. Однако если в теле цикла прописать оператор break, то он прервется, как только встретится:

```
while True:
    i += 1
    break
```

Здесь будет запущен цикл, выполнится один раз оператор `i += 1` и следующая команда break прервет дальнейшее выполнение цикла while.

Конечно, это искусственный пример демонстрирующий работу оператора break, и писать программы с «вечными» циклами следует с большой осторожностью. В целом, нужно их избегать и корректно прописывать условия циклов и тогда не потребуется применять оператор break. Но бывают ситуации, когда все же требуется досрочно прервать работу цикла. Например, имеется список с числами:

```
d = [1, 5, 3, 6, 0, -4]
```

И нужно определить, есть ли среди них хотя бы одно четное значение. Здесь удобно воспользоваться оператором break, так как только встретится первое четное значение, дальнейшую проверку можно не проводить:

```
d = [1, 5, 3, 6, 0, -4]
flFind = False
i = 0

while i < len(d):
    flFind = d[i] % 2 == 0
    if flFind:
        break
    i += 1

print(flFind)
```

Конечно, эту же программу можно было бы записать и без break, используя составное условие:

```
i = 0
while i < len(d) and d[i] % 2 != 0:
    i += 1

flFind = i != len(d)
```

Но, на мой взгляд, первый вариант с break красивее и понятнее, чем второй. Хотя, все дело вкуса – кому, что нравится. Вот принцип работы оператора break. Он прерывает работу любого оператора цикла, как только выполнение программы доходит до него.

5.2.2. Оператор continue

Следующий оператор continue позволяет пропускать одну итерацию тела цикла. Давайте, опять же, на конкретном примере посмотрим, где и как его

целесообразно применять.

Предположим, что у пользователя в цикле запрашивается ввод с клавиатуры целых чисел и требуется просуммировать все нечетные значения. При этом цикл завершается, когда пользователь вводит число 0. Реализовать эту программу можно следующим образом:

```
s = 0
d = 1

while d != 0:
    d = int(input("Введите целое число: "))
    if d % 2 == 0:
        continue

    s += d
    print("s = " + str(s))
```

Здесь на каждой итерации цикла проверяется на четность текущее значение d. Если условие оказывается истинным, то последующий подсчет суммы и вывод ее в консоль не выполняется, пропускается. При этом цикл продолжает свою работу и переходит к следующей итерации, пока пользователь не введет число 0. Благодаря оператору continue основная логика работы программы прописывается непосредственно в теле цикла, а не во вложенном блоке условного оператора. То есть, если программу переписать без continue, то она бы выглядела так:

```
s = 0
d = 1

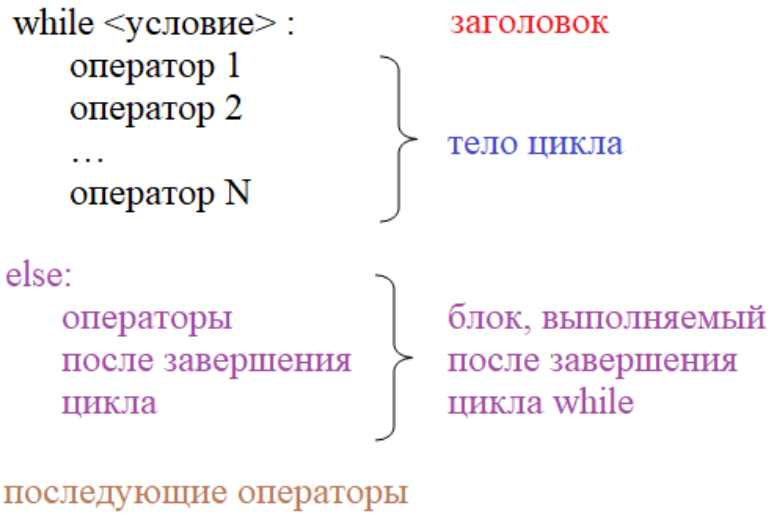
while d != 0:
    d = int(input("Введите целое число: "))
    if d % 2 != 0:
        s += d
        print("s = " + str(s))
```

Здесь получается вложенный блок операторов, реализующий часть логики работы программы. Конечно, на производительность это никак не влияет, но, все же такие вложения лучше избегать, так программа становится менее читабельной и удобной для дальнейшего редактирования.

5.2.3. Оператор else

Совместно с операторами циклов можно прописывать блок else. Мы уже говорили об этом операторе, когда рассматривали условия. Циклы тоже имеют подобный блок и мы сейчас увидим, как он работает.

Синтаксис этого оператора следующий. После тела цикла можно прописывать необязательный блок else, в котором указываются операторы, выполняющиеся после завершения цикла:



И здесь возникает вопрос. А чем блок else отличается от блока операторов сразу идущих после блока while? Ведь когда цикл while завершается так и так осуществляется переход к последующим операторам! Все верно. Однако блок else будет срабатывать не всегда при завершении цикла, а только при его **штатном** завершении. А штатное завершение – это когда условие цикла стало равным False и оператор while прекратил свою работу по указанному в нем условию. Только в этом случае осуществляется переход к блоку else.

Давайте рассмотрим это на конкретном примере. Предположим, нужно вычислять сумму вида:

$$S = 1/2 + 1/3 + 1/4 + 1/10 + \ldots + 1/0$$

И если в этой сумме встречается деление на ноль, то вычисления следует прервать. Реализуем эту программу следующим образом:

```
S=0
i=-10

while i < 100:
    if i == 0:
        break

    S += 1/i
    i += 1
else:
    print("Сумма вычислена корректно")

print(S)
```

Смотрите, если при вычислении суммы встречается деление на 0 (`i == 0`), то срабатывает оператор break и цикл досрочно прерывается, то есть, завершается в нештатном режиме. В этом случае блок else пропускается и мы не видим сообщения, что сумма вычислена корректно. Если же все проходит штатно и цикл завершается по записанному условию, то в консоли появляется сообщение «Сумма вычислена корректно». Вот так, например, применение блока else

помогает оценить корректность вычисления суммы.

5.3. Оператор цикла for. Функции range() и enumerate()

Рассмотренный ранее оператор while позволяет создавать циклы, работающие по заданному условию. Однако в Python имеется еще один оператор цикла for, который удобно применять для перебора элементов любых итерируемых (перебираемых) коллекций. Он имеет следующий синтаксис:

```
for <переменная> in <итерируемый объект>:
      оператор 1
      оператор 2
       …
      оператор N
```

Подробно об итерируемых объектах мы с вами еще будем говорить, а пока достаточно знать, что это объекты, состоящие из множества элементов, которые можно перебирать. Например, списки или строки. Давайте посмотрим, как оператор цикла for может быть применен к таким типам данных.

Пусть имеется список:

```
d = [1, 2, 3, 4, 5]
```

и требуется перебрать все его элементы. Через оператор цикла for это делается следующим образом:

```
for x in d:
    print(x, end=" ")
```

Видим на экране значения:

```
1 2 3 4 5
```

Или вместо списка можно перебирать символы строки:

```
for x in "python":
    print(x, end=" ")
```

Получим вывод:

```
p y t h o n
```

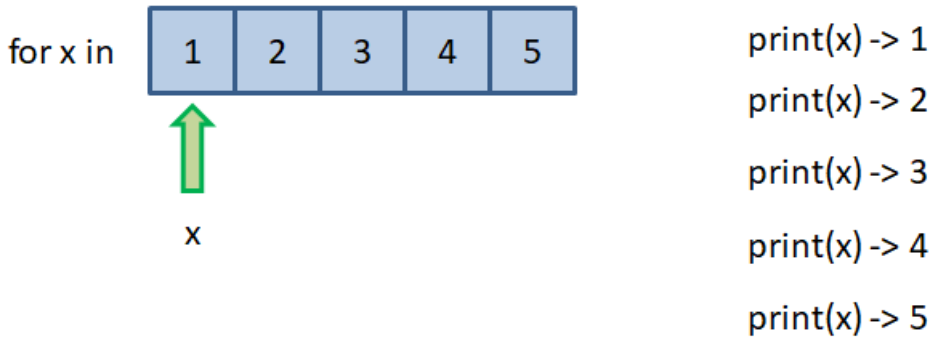
Давайте теперь посмотрим, как все это в деталях работает. Вернемся к примеру со списком:

```
d = [1, 2, 3, 4, 5]

for x in d:
    print(x, end=" ")
```

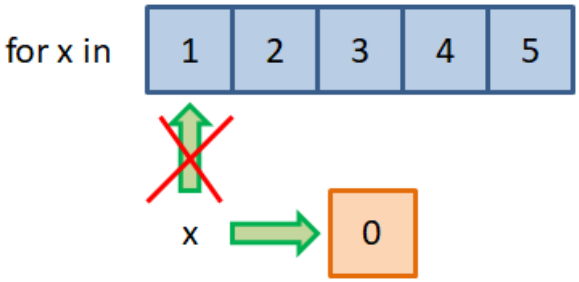
На первой итерации переменная x ссылается на первый элемент со значением 1. Соответственно, функция print выводит это значение в консоль. На следующей итерации переменная x будет ссылаться уже на второй элемент и функция print

выведет значение 2. И так до тех пор, пока не достигнем конца списка.



Обратите внимание, что переменная x в цикле for лишь ссылается на элемент списка, а не является самим элементом. Это значит, что если бы мы захотели изменить какое-либо значение списка, используя переменную x, например, так:

```
for x in [1, 2, 3, 4, 5]:
    x = 0
```



то эффекта никакого не было бы. Здесь x просто станет ссылаться на другой объект со значением 0, но элементы списка от этого никак не поменяются. То есть, в такой реализации оператора цикла for мы можем лишь перебирать значения элементов, читая их значения, например, с целью вычисления произведения:

```
d = [5, 2, 4, 3, 1]

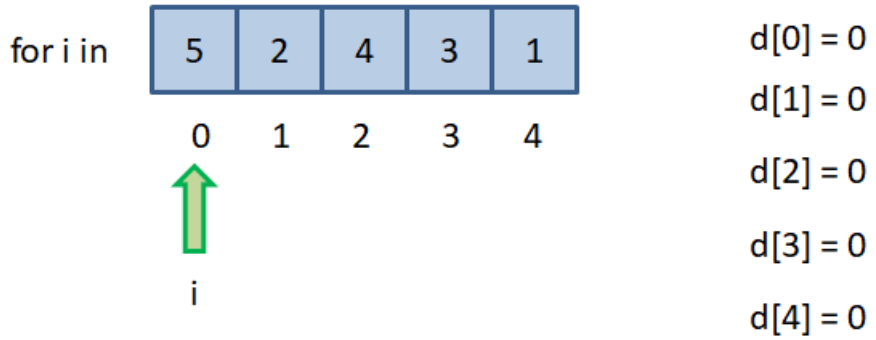
p = 1
for x in d:
    p *= x
```

Но как тогда менять значения элементов списка, используя цикл for? Для этого к элементам списка нужно обращаться по индексу. То есть, цикл должен перебирать не элементы списка, а его индексы, например, так:

```
d = [5, 2, 4, 3, 1]

for i in [0, 1, 2, 3, 4]:
    d[i] = 0

print(d)
```



В этом случае на каждой итерации цикла будет происходить обращение сначала к первому элементу списка с присвоением нуля, затем ко второму элементу, с тем же присвоением нуля, и так для всех его элементов. В итоге будут изменяться элементы самого списка.

5.3.1. Функция range()

Однако описывать индексы через еще один список далеко не лучшая практика. Для подобных целей в Python существует специальная функция range, которая генерирует арифметическую последовательность чисел, и имеет следующий синтаксис:

range(start, stop, step)

Например, для генерации последовательности от 0 до 4 функцию range можно записать в виде:

```
range(5)
range(0, 5)
range(0, 5, 1)
```

Здесь всюду последнее значение 5 не включается в диапазон. Чтобы увидеть сгенерированные числа, преобразуем их в список с помощью известной нам функции list:

```
list(range(5))        # [0, 1, 2, 3, 4]
list(range(0, 5))     # [0, 1, 2, 3, 4]
list(range(0, 5, 1))  # [0, 1, 2, 3, 4]
```

Если же записать:

```
list(range(0))        # []
```

то получим пустой список, так как значения начинаются с нуля и заканчиваются нулем, но ноль в интервал не входит.

То же самое произойдет и при указании любого отрицательного значения, например:

```
list(range(-5))       # []
list(range(0, -5))    # []
```

Но как тогда формировать последовательность из отрицательных значений?

Очень просто, в качестве стартового значения нужно указать число меньше конечного значения:

```
list(range(-10, -5))     # [-10, -9, -8, -7, -6]
```

или дополнительно с указанием шага:

```
list(range(-10, -5, 2)   # [-10, -8, -6]
```

Но если записать отрицательный шаг:

```
list(range(-10, -5, -2)) # []
```

то снова увидим пустой список. Почему так получилось? Дело в том, что теперь движение начинается от -10 в меньшую сторону, и значение -5 становится недостижимым. В этом случае функция range не выдает никаких значений. Чтобы исправить ситуацию в качестве конечного значения нужно записать число меньше -10, например, -20:

```
list(range(-10, -20, -2))  # [-10, -12, -14, -16, -18]
```

Вот так генерируются последовательности в обратном порядке. То же самое можно проделать и в положительной области:

```
list(range(5, 0, -1))      # [5, 4, 3, 2, 1]
```

Обратите внимание, мы начинаем с 5 и заканчиваем 1. Здесь конечное значение 0 не включается в диапазон. Если же нам нужно дойти до нуля, то в данном случае следует указать -1 в качестве конечного значения:

```
list(range(5, -1, -1))     # [5, 4, 3, 2, 1, 0]
```

Это всегда следует помнить при работе с функцией range: конечное значение не включается в диапазон и все генерируемые значения являются целочисленными. С вещественными числами функция range не работает.

Теперь, когда мы с вами узнали, как работает функция range, перепишем предыдущую программу перебора элементов списка по индексам следующим образом:

```
d = [5, 2, 4, 3, 1]

for i in range(5):
    d[i] = 0
print(d)
```

Обратите внимание, здесь не нужно функцию range превращать в список. Оператор цикла for умеет перебирать любые итерируемые объекты, а range, как раз, возвращает такой объект. Поэтому переменная i будет принимать значения от 0 до 4 включительно. После выполнения этой программы увидим, что все элементы списка стали равны нулю:

```
[0, 0, 0, 0, 0]
```

Однако прописывать явно число 5 в функции range не лучшее решение. Заменим его вызовом функции определения длины списка len следующим образом:

```
for i in range(len(d)):
    d[i] = 0
```

Получаем более универсальное решение, работающее со списком любой длины.

Конечно, функцию range в цикле for используют не только для удобного формирования индексов, но и для формирования требуемых числовых последовательностей. Например, если требуется вычислить сумму ряда:

$$S = 1/2 + 1/3 + 1/4 + \ldots + 1/1000$$

то это можно реализовать так:

```
S = 0

for i in range(2, 1001):
    S += 1 / i

print(S)
```

Здесь функция range генерирует на каждой итерации очередное значение знаменателя дроби, и затем оно используется для подсчета суммы s.

5.3.2. Примеры работы оператора цикла for. Функция enumerate()

Давайте рассмотрим несколько примеров использования цикла for для решения различных задач. Первая классическая задача – это вычисление факториала от неотрицательного целого числа n в пределах от 0 до 100. Пусть пользователь вводит число n, а программа выдает значение его факториала:

```
n = int(input("Введите неотрицательное цело число не больше 100: "))

if n < 0 or n > 100:
    print("Неверно введено натуральное число")
else:
    p = 1
    for i in range(1, n+1):
        p *= i

    print(f"Факториал {n}! = {p}")
```

Обратите внимание на запись функции range. Чтобы она выдавала последовательность целых чисел от 1 до n включительно, необходимо записать n+1, так как последнее значение не включается в диапазон.

Следующий пример. Отображение символов «*» в виде ёлочки. Сделать это можно следующим образом:

```
for i in range(1, 7):
    print('*' * i)
```

```
*
**
***
****
*****
******
```

Здесь счетчик i будет принимать значения 1, 2, 3, 4, 5 и 6. Соответственно, функция print выводит строку с дублированием символов «*» i раз. В итоге, в первой строке будет одна звездочка, во второй – две и так до шести звездочек. Видите, как просто можно реализовать такую программу, используя оператор цикла, и операцию «*» работы со строками.

Следующее задание заключается в объединении всех слов списка в одно предложение. Предположим, что имеется список:

```
words = ["Python", "дай", "мне", "силы",
         "пройти", "этот", "курс", "до", "конца"
]
```

Объединение этих слов будем выполнять через пробел следующим образом:

```
s = ''
for w in words:
    s += ' ' + w

print(repr(s))
```

После запуска программы увидим результат:

```
' Python дай мне силы пройти этот курс до конца'
```

Здесь образуется пробел вначале строки. Убрать его можно двумя способами. Либо вызвать метод строки:

```
s = s.lstrip()
```

Либо сделать так, чтобы он не появлялся:

```
s = ''
flFirst = True
for w in words:
    s += ('' if flFirst else ' ') + w
    flFirst = False
```

На мой взгляд, здесь лучше воспользоваться методом lstrip, т.к. программа в этом случае выглядит более читаемой, а скорость работы в обоих случаях будет примерно одинакова.

Конечно, внимательный читатель сейчас с упреком заявляет, что все это можно сделать гораздо проще с помощью уже знакомого нам строкового метода join:

```
print(" ".join(words))
```

И был бы абсолютно прав! Те из вас, кто думал так же, проявили себя, как настоящие питонисты!

Следующая задача. В списке чисел:

```
digs = [4, 3, 100, -53, -30, 1, 34, -8]
```

все двузначные числа нужно заменить нулями. Используя текущие знания, программу можно записать следующим образом:

```
for i in range(len(digs)):
    if 10 <= abs(digs[i]) <= 99:
        digs[i] = 0

print(digs)
```

Обратите внимание, что в цикле перебираются индексы элементов списка, а не сами элементы. Благодаря этому в цикле получаем возможность изменять значение каждого i-го элемента.

В этой задаче нам потребовалось в цикле работать и с индексом элемента и с его значением. Для подобных ситуаций в Python существует специальная функция:

индекс, значение = enumerate(объект)

которая возвращает пару (индекс, значение). Используя эту функцию, можно переписать программу следующим образом:

```
for i, d in enumerate(digs):
    if 10 <= abs(d) <= 99:
        digs[i] = 0
```

Такой вариант выглядит более читаемым. Во всем остальном эта программа подобна предыдущей.

Еще один пример, который мы с вами рассмотрим, – это преобразование кириллицы в латиницу. Такая задача иногда возникает при создании сайтов.

Вначале объявим список замен для соответствующих русских букв (по алфавиту от а до я):

```
t = ['a', 'b', 'v', 'g', 'd', 'e', 'zh',
     'z', 'i', 'y', 'k', 'l', 'm', 'n', 'o', 'p',
     'r', 's', 't', 'u', 'f', 'h', 'c', 'ch', 'sh',
     'shch', '', 'y', '', 'e', 'yu', 'ya'
]
```

Затем определим кодовое значение для первой буквы этого списка:

```
start_index = ord('a')
```

Далее, запишем две переменные:

```
title = "Программирование на Python - лучший курс"
slug = ''
```

Переменная title содержит заголовок, который требуется преобразовать в латиницу, а переменная slug будет содержать результат этого преобразования.

Само преобразование реализуем с помощью цикла, перебирая каждый символ исходной строки. Предварительно, строку title переведем в нижний регистр, чтобы работать только с малыми буквами. Получим:

```
for s in title.lower():
    if 'a' <= s <= 'я':
        slug += t[ord(s) - start_index]
    elif s == 'ё':
        slug += 'yo'
    elif s in ' !?:,.':
        slug += '-'
    else:
        slug += s
```

Если запустить программу в таком виде и вывести на экран переменную slug, то увидим следующий результат:

```
programmirovanie-na-python---luchshiy-kurs
```

Получаем несколько подряд идущих символов дефиса. Было бы правильно заменять такие последовательности на один символ. Сделаем это с использованием цикла while:

```
while slug.count('--'):
    slug = slug.replace('--', '-')

print(slug)
```

Теперь итоговая строка принимает вид:

```
programmirovanie-na-python-luchshiy-kurs
```

Конечно, это не самая оптимальная реализация данной задачи. Лучше ее делать с использованием, так называемых, регулярных выражений. Но, используя только текущие знания и, как пример, она очень хорошо показывает порядок работы операторов циклов for и while.

5.3.3. Операторы break, continue и else в цикле for

Внутри оператора цикла for можно так же, как и в цикле while, прописывать операторы break, continue и добавлять блок else. Порядок их работы абсолютно тот же самый:

- оператор break прерывает работу цикла for;
- оператор continue пропускает одну итерацию цикла for;
- блок else содержит операторы, выполняющиеся после штатного завершения цикла for.

Давайте рассмотрим несколько примеров использования ключевых слов break, continue и else внутри for. Начнем с оператора break.

Пусть имеется список из чисел, в котором требуется найти первое значение больше семи. Программу можно реализовать следующим образом:

```
numbers = [1, 3, 5, 7, 9, 11, 9, 13, -5]

for num in numbers:
    if num > 7:
        print(f"Найдено первое число больше 7: {num}")
        break
    print(f"Проверяем: {num}")
```

После запуска программы, получаем результат ее работы:

```
Проверяем: 1
Проверяем: 3
Проверяем: 5
Проверяем: 7
Найдено первое число больше 7: 9
```

Как видите, при срабатывании оператора break, оператор цикла for полностью завершает свою работу и управление передается следующему оператору, стоящему после цикла (в данном случае программа просто завершается).

Давайте модифицируем эту программу и будем дополнительно выводить строку «Искомый элемент не найден!» в случае его отсутствия в списке. Это можно сделать путем добавления блока else после цикла for:

```
numbers = [1, 3, 5, 7, 9, 11, 9, 13, -5]

for num in numbers:
    if num > 7:
        print(f"Найдено первое число больше 7: {num}")
        break
    print(f"Проверяем: {num}")
else:
    print("Искомый элемент не найден!")
```

Если сейчас запустить программу, то результат вывода не изменится, т.к. цикл for завершается оператором break, то есть, не штатно. А вот если изменить список numbers, убрав из него все числа больше 7:

```
numbers = [1, 3, 5, 7]
```

то увидим следующий результат вывода:

```
Проверяем: 1
Проверяем: 3
Проверяем: 5
Проверяем: 7
Искомый элемент не найден!
```

То есть, функция print, записанная в блоке else, выполняется только при штатном завершении работы цикла for.

Следующий пример показывает возможность использования оператора continue внутри тела цикла for. Пусть имеется список со следующими значениями:

```
data = [10, -11, 20, 6.5, None, -10, 0, 7, None, None]
```

Требуется пропускать значения None и вычислять сумму только неотрицательных чисел. Реализовать это можно следующим образом:

```
s = 0
for x in data:
    if x is None or x < 0:
        continue

    s += x

print(s)
```

Здесь записано условие, в котором проверяется, что текущее значение элемента списка data равно либо None, либо меньше 0. Если это так, то выполняется оператор continue, который пропускает все ниже стоящие операторы в теле цикла на текущей итерации. После чего цикл продолжается с переходом к следующему значению списка.

Конечно, все эти операторы break, continue и else можно использовать совместно в рамках одного цикла. Покажем это на примере вывода слов от 5 символов и более, но при обнаружении слова более 10 символов цикл будет прерываться:

```
words = ["кот", "добрые", "дом", "циклы", "окно", "программирование"]

for word in words:
    if len(word) < 5:
        continue  # пропускаем короткие слова

    if len(word) > 10:
        break

    print(word)
else:
    print("В списке были обработаны все слова.")
```

После запуска этой программы увидим результат:

Здесь оператор continue пропускает слова меньше 5 символов, оператор break прерывает цикл for при обнаружении слова более 10 символов, а блок else служит для вывода строки:

```
"В списке были обработаны все слова."
```

в случае штатного завершения работы цикла for.

Конечно, один и тот же алгоритм (задачу) можно программно реализовать самыми разными способами. В одной программе могут использоваться операторы break и/или continue, в другой обходятся без них. В конечном итоге, нужно стремиться, чтобы итоговый текст программы получался понятным, читаемым, а сама программа использовала как можно меньше памяти и выполнялась как можно быстрее. Это основные критерии написания программ, которые на данный момент следует учитывать. А какие операторы будут использованы – это уже на усмотрение самого программиста. Составление программ – творческий процесс.

5.4. Итератор и итерируемые объекты. Функции iter() и next()

Мы с вами уже знаем, что в языке Python существуют объекты, содержащие последовательность некоторых элементов, например, строки и списки. С одной стороны – это совершенно разные объекты, разных типов. Но с другой их элементы можно перебирать единым образом с помощью цикла for, например:

```
s = "Программирование"
lst = [1, 2, 3, 'one', 'two', True, [5, 6]]

for x in s:
    print(x)

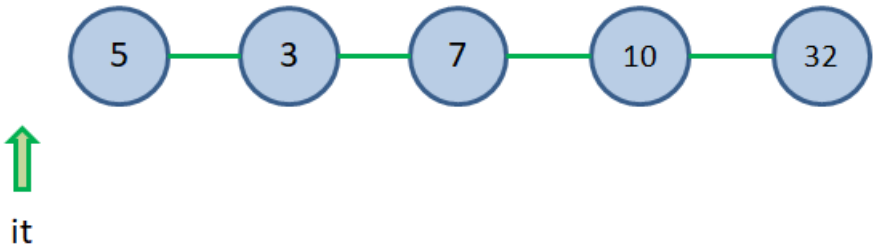
for x in lst:
    print(x)
```

Каким же образом цикл for «знает», как обрабатывать эти разнотипные объекты и выделять из них последовательно элементы? Дело в том, что делает он это с помощью универсального механизма перебора элементов под названием **итератор**. Каждый итерируемый объект (строка, список и все остальные коллекции) реализуют этот механизм и через него предоставляют последовательный доступ к своим элементам. Используя объект итератора, можно один раз пройтись по всем элементам коллекции от начала и до конца. Давайте посмотрим, как этот осуществляется на языке Python.

Чтобы получить доступ к объекту итератора, например, для списка, нужно вызвать специальную функцию iter и в ней первым аргументом прописать итерируемый объект, например, так:

```
d = [5, 3, 7, 10, 32]
```

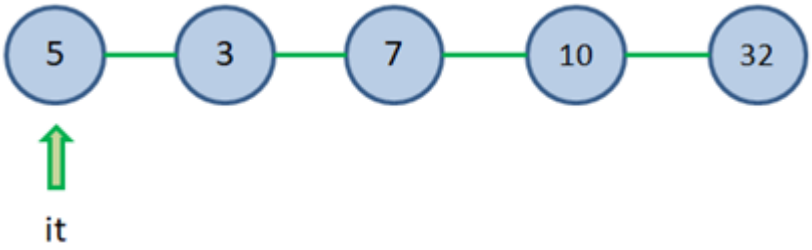
```
it = iter(d)            # it – объект итератора
```



Используя итератор it, можно осуществить однократный перебор всех элементов списка d. Делается это с помощью последовательного вызова функции next следующим образом:

```
res = next(it)  # res = 5
```

Функция next осуществляет переход к следующему элементу (при первом вызове переходим к первому элементу) и возвращает значение этого элемента. В нашем примере – это значение 5. Соответственно состояние итератора it также меняется. Теперь он указывает на первый элемент списка.



Если вызвать функцию next еще раз:

```
res = next(it)  # res = 3
```

то итератор it перемещается на следующий (второй) элемент списка и возвратит его значение 3. И так можно дойти до конца нашего списка:

```
res = next(it)  # res = 7
res = next(it)  # res = 10
res = next(it)  # res = 32
```

Когда итератор it находится на последнем элементе коллекции и для него вызывается функция next:

```
res = next(it)  # StopIteration
```

то возникает исключение (ошибка) StopIteration. Такие ошибки можно программно обрабатывать с помощью механизма обработки исключений. Цикл for именно так и делает, чтобы определить конец перебираемых данных в коллекции. (Об обработке исключений речь еще впереди.)

Когда итератор дошел до конца коллекции, его уже нельзя вернуть назад и пройти все элементы заново. Для этого придется создавать новый итератор с помощью функции iter:

```
it = iter(d)
```

и снова, с самого начала перебирать все элементы:

```
res = next(it)  # res = 3
```

Так работает механизм перебора элементов любого итерируемого объекта. Причем он универсален и не зависит от типа объекта: это может быть и список и строка и любой другой перебираемый объект. Например, можно взять строку:

```
s = "python"
```

Создать для нее итератор:

```
it_s = iter(s)
```

и той же функцией next перебирать ее символы:

```
next(it_s)  # 'p'
next(it_s)  # 'y'
```

Так итератор дает возможность универсальным образом перебирать элементы любого итерируемого объекта, который поддерживает этот механизм. Также следует иметь в виду, что доступ к элементам через итератор и по индексу:

```
res = s[2]  # 't'
```

– это совершенно разные способы обращения к элементам. К тому же не у всех итерируемых объектов есть возможность указания индексов.

Помимо строк и списков к итерируемым объектам также относится ранее рассмотренная функция:

```
r = range(5)
```

Для нее тоже можно сформировать итератор:

```
it = iter(r)
```

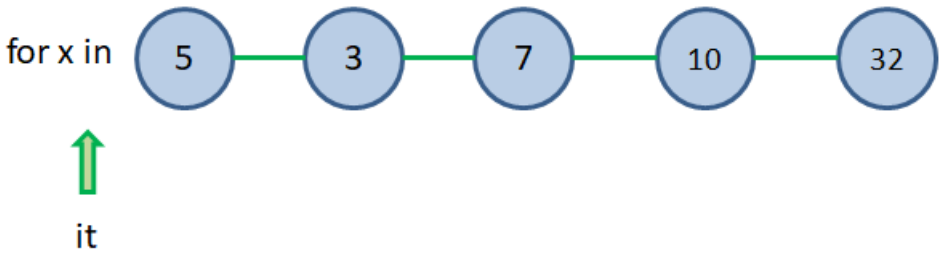
и перебирать значения функцией next:

```
next(it)   # 0
next(it)   # 1
```

Еще раз подчеркнем ценность механизма итераторов. На первый взгляд кажется, что мы и так можем обращаться к элементам строк, списков, используя индексы. Однако если в программе требуется перебирать итерируемые объекты самых разных типов (иногда наперед неизвестных), то единственный универсальный и безопасный способ сделать это – использовать итераторы. Например, как я уже говорил, так происходит в операторе цикла for. В нем можно перебирать любой итерируемый объект и цикл for должен «уметь» это делать вне зависимости от типа указанной коллекции. Поэтому внутри цикла for формируется объект итератора, через который, затем, осуществляется последовательный перебор

элементов, пока не возникнет исключение StopIteration. Именно так в цикле for осуществляется перебор списков:

```
for x in [5, 3, 7, 10, 32]:
    print(x)
```



строк:

```
for x in "python":
    print(x)
```

и других итерируемых объектов:

```
for x in range(1, 6):
    print(x)
```

А что будет, если применить функцию iter для неитерируемого объекта? Например, числа:

```
iter(5)   # TypeError
```

В этом случае возникнет ошибка, говорящая, что объект типа int не является итерируемым, то есть, не поддерживает итератор и перебор элементов. Действительно, число – это одно конкретное, не перебираемое значение, поэтому оно относится к неитерируемым объектам. Соответственно, в цикле for мы не можем указывать такие элементы:

```
for x in 5:
    print(x)
```

Возникнет все та же ошибка TypeError. Это следует иметь в виду, при использовании цикла for.

5.5. Вложенные циклы

Рассмотренные операторы циклов while и for могут содержать любые операторы внутри себя, в том числе и дополнительные операторы циклов. Такая конструкция, когда один цикл вложен в другой, называется вложенными циклами.

Давайте вначале разберемся, как работают такие конструкции. Для простоты воспользуемся двумя циклами for (один вложен в другой). Первый (внешний) цикл for будет перебирать целые числа от 1 до 3, а второй – целые числа от 1 до 5:

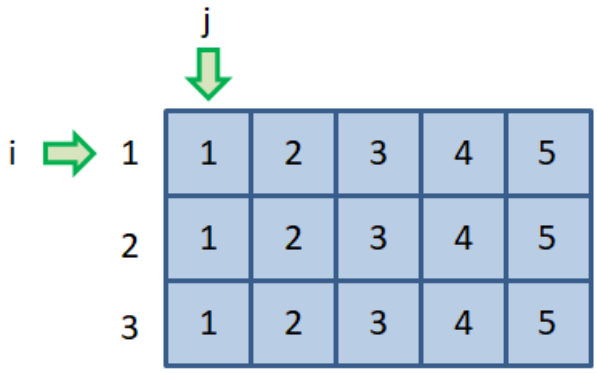
```
for i in range(1, 4):
    for j in range(1, 6):
```

```
        print(f"i = {i}, j = {j}", end=' ')

    print()
```

После запуска программы на экране увидим следующую таблицу чисел:

```
i = 1, j = 1 i = 1, j = 2 i = 1, j = 3 i = 1, j = 4 i = 1, j = 5
i = 2, j = 1 i = 2, j = 2 i = 2, j = 3 i = 2, j = 4 i = 2, j = 5
i = 3, j = 1 i = 3, j = 2 i = 3, j = 3 i = 3, j = 4 i = 3, j = 5
```



Почему получились именно такие значения? Вначале у нас счетчик i принимает значение 1, а счетчик j пробегает числа от 1 до 5, в итоге на экран выводится первая строка. После завершения вложенного цикла, срабатывает функция print и курсор переходит на новую строку. После этого переходим ко второй итерации первого цикла. Теперь i = 2. Счетчик j снова проходит значения от 1 до 5 вложенного цикла и выводится вторая строка. На следующей итерации первого (внешнего) цикла переменная i = 3, а переменная j проходит значения от 1 до 5 и осуществляется вывод третьей строки. То есть, вложенный цикл for трижды запускался заново и каждый раз j меняется от 1 до 5. Это и есть принцип работы вложенного цикла – на каждой итерации внешнего цикла он отрабатывает снова и снова, пока не завершится первый (внешний) цикл for.

Возможно здесь возникает вопрос, для чего нужны подобные вложенные конструкции из циклов? Давайте представим, что имеется вложенный (двумерный) список:

```
a = [[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]
```

При переборе его элементов с помощью одного цикла for:

```
for row in a:
    print(row, type(row))
```

переменная row будет ссылаться сначала на первый вложенный список, затем, на второй и потом на третий:

```
[1, 2, 3, 4] <class 'list'>
[2, 3, 4, 5] <class 'list'>
[3, 4, 5, 6] <class 'list'>
```

Но, так как row ссылается на список, то есть, на итерируемый объект, то нам

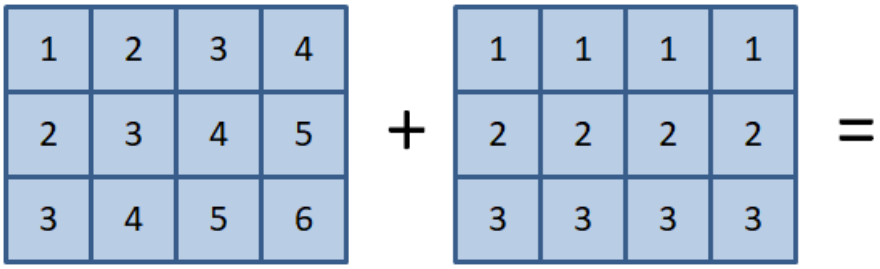
ничто не мешает перебрать его элементы с помощью вложенного цикла for:

```
for row in a:
    for x in row:
        print(x, type(x), end=' ')
    print()
```

Как видите, теперь в консоль выводятся числа типа int, то есть, мы обращаемся непосредственно к элементам этого двумерного списка:

```
1 <class 'int'> 2 <class 'int'> 3 <class 'int'> 4 <class 'int'>
2 <class 'int'> 3 <class 'int'> 4 <class 'int'> 5 <class 'int'>
3 <class 'int'> 4 <class 'int'> 5 <class 'int'> 6 <class 'int'>
```

В качестве более практичного примера использования вложенных циклов можно рассмотреть задачу суммирования двух матриц:



Пусть каждая матрица представлена своим двумерным списком:

```
a = [[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]
b = [[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3]]
```

Тогда вычислить их сумму можно следующим образом:

```
c = []  # результат суммирования
for i, row in enumerate(a):
    r = []
    for j, x in enumerate(row):
        r.append(x + b[i][j])

    c.append(r)

print(c)
```

Обратите внимание, здесь в программе используется уже знакомая нам функция enumerate, которая возвращает индекс и значение текущего элемента. Это удобно для реализации данной программы. Внутри первого цикла на каждой итерации создается новый пустой список `r`, который наполняется значениями с помощью метода append. Затем, полученная строка (список `r`) добавляется в основной список `c`. Так формируется и вычисляется сумма значений элементов двух одинаковых по размерам списков `a` и `b`.

Другой пример. Пусть имеется текст, представленный в виде списка:

```
t = ["- Скажи-ка,  дядя, ведь не даром",
```

```
    "Я Python выучил с   каналом",
    "Балакирев что    раздавал?",
    "Ведь были  ж заданья боевые,",
    "Да, говорят,  еще какие!",
    "Недаром помнит    вся Россия",
    "Как мы рубили   их тогда!"]
```

Здесь в строках присутствуют два и более пробелов. Наша задача удалить их и оставить только один. Сделаем это с помощью вложенных циклов. В первом цикле for будем перебирать строки – элементы списка, а во втором (вложенном) цикле while удалять лишние пробелы:

```
for i, line in enumerate(t):
    while line.count('  '):
        line = line.replace('  ', ' ')

    t[i] = line

print(t)
```

В качестве условия вложенного цикла while используется вызов метода count, который подсчитывает число фрагментов из двух пробелов подряд. Как только метод count вернет 0, – это будет означать False и цикл while завершится. Преобразованная строка line становится новым i-м элементом списка t. Функция print выводит преобразованный список t на экран.

Следующий пример. Пусть вначале в программе формируется вложенный список размером M x N элементов, состоящий из всех нулей. Причем M, N вводятся с клавиатуры:

```
M, N = list(map(int, input("Введите M и N: ").split()))

zeros = []
for i in range(M):
    zeros.append([0]*N)

print(zeros)
```

После выполнения этой программы на экране увидим:

```
Введите M и N: 3 4
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Ставится задача заменить все элементы полученного списка на единицы. Сделать это можно различными способами. Один из них – использование вложенных циклов:

```
for i in range(M):
    for j in range(N):
        zeros[i][j] = 1
```

Справедливости ради, надо сказать, что ту же самую задачу можно реализовать и

так:

```
for i in range(M):
    zeros[i] = [1] * N
```

Этот пример показывает, что один и тот же алгоритм можно выполнить самыми разными способами. Искусство программирования состоит в применении наиболее эффективных подходов для решения поставленных задач.

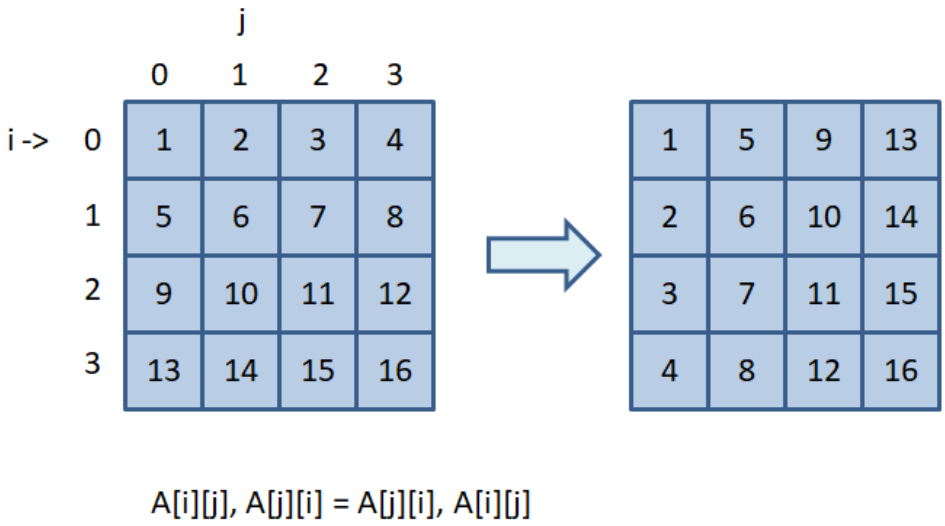
И еще один, последний пример вложенных циклов. Пусть имеется квадратный список размером 4 x 4:

```
A = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]
```

Необходимо поменять строки на столбцы и получить новое представление этого же списка. Для этого достаточно поменять местами элементы, стоящие выше главной диагонали с соответствующими элементами, стоящими ниже главной диагонали. Получим следующую реализацию:

```
for i in range(len(A)):
    for j in range(i+1, len(A)):
        A[i][j], A[j][i] = A[j][i], A[i][j]

for r in A:
    for x in r:
        print(x, end='\t')
    print()
```



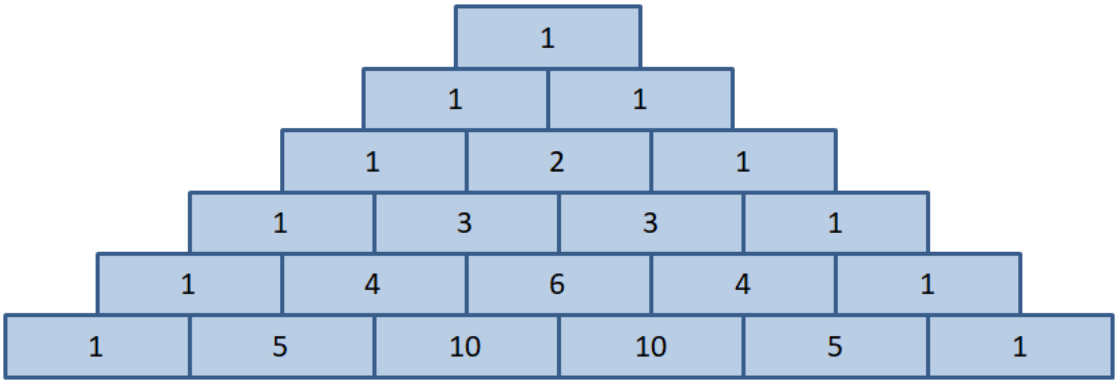
A[i][j], A[j][i] = A[j][i], A[i][j]

В математике такое преобразование называется транспонированием матрицы. Попробуйте самостоятельно разобраться, как работает приведенная программа.

В заключение отмечу, что уровень вложенности операторов циклов может быть больше двух. Однако на практике слишком большого вложения следует избегать, так как программа становится менее читаемой и гибкой для дальнейшего изменения. Лучше ограничиваться тремя подряд идущими вложениями операторов циклов, не более того.

5.5.1. Треугольник Паскаля

Хорошим примером использования вложенных циклов является известная задача построения треугольника Паскаля. Этот треугольник состоит из особых чисел, которые вычисляются по правилу: по краям каждой строки стоят единицы, а каждое из остальных чисел равно сумме двух стоящих над ним из предыдущей строки. Именно в таком виде его привел французский математик Блез Паскаль в «Трактате об арифметическом треугольнике». Популярность этого треугольника возникла из-за того, что эти числа довольно часто встречаются в самых разных задачах математики: алгебре, комбинаторике, теории вероятностей и многих других. Мы сейчас не будем вдаваться в их анализ, так как перед нами стоит несколько иная задача – написать программу формирования такого треугольника.



Первый вопрос, на который обычно требуется ответить при разработке новой программы, как хранить данные? В данном случае, как хранить эти числа? Мы это сделаем с помощью неравномерного вложенного списка. Смотрите, весь треугольник можно представить как набор вложенных списков постоянно увеличивающейся длины:

```
P = [ [1],
      [1, 1],
    [1, 2, 1],
        …
]
```

Далее, в программе мы запишем два вложенных цикла for со счетчиками: i – для перебора строк треугольника Паскаля; j – для перебора элементов формируемой строки. Причем j будет меняться в диапазоне [0; i]. То есть, для первой строки принимает только одно значение 0, для второй – два значения [0; 1], для третьей – три [0; 1; 2] и так далее. А формировать значения строк будем следующим образом. Для каждой новой (i-й) строки создадим список длиной i+1, состоящий из всех единиц:

```
row = [1] * (i+1)
```

Затем, все крайние элементы с индексами:

$$j == 0 \text{ or } j == i$$

трогать не станем, так как там уже стоят единицы. А все другие значения будут

формироваться на основе суммы двух предыдущих:

```
row[j] = P[i-1][j-1] + P[i-1][j]
```

В результате получаем следующую программу:

```
N = 7
P = []

for i in range(0, N):
    row = [1] * (i + 1)
    for j in range(i + 1):
        if j != 0 and j != i:
            row[j] = P[i-1][j-1] + P[i-1][j]

    P.append(row)

for r in P:
    print(r)
```

Здесь вначале задается общее число строк N=7 для треугольника Паскаля, затем коллекция P=[], которая будет хранить значения треугольника Паскаля и, далее, идут вложенные циклы. Первый цикл со счетчиком i перебирает строки, а второй со счетчиком j перебирает элементы текущей строки row. Внутри вложенного цикла делается проверка: если индекс j ссылается не на крайние элементы строки, то вычисляются их значения на основе суммы двух чисел из предыдущей строки. После этого сформированная строка добавляется в список P. Эта операция повтряется N раз. В конце программы выводится полученный результат:

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
```

Действительно, у нас сформировался треугольник Паскаля, состоящий из семи строк. Этот пример показывает, как с помощью вложенных циклов можно реализовать эту относительно несложную задачку.

Опять же, все программы, что приводятся в данной книге, не являются истиной в последней инстанции. У вас могут получаться другие программные реализации той же самой задачи. Здесь главное помнить, что лучшая реализация будет та, которая расходует меньше памяти и работает быстрее остальных. Это два самых главных критерия построения любых программ. И, разумеется, программы должны приводить к корректным результатам, иначе их можно считать неработоспособными.

5.6. Генераторы списков (List comprehension)

Ранее в программах нам не раз приходилось формировать различные списки, непосредственно прописывая в них значения элементов, например:

```
lst = [1, 2, 10, -5]
```

Либо, размножая его элементы:

```
res = ['abs', 'sum', 'round'] * 3
```

Однако существует еще один довольно распространенный способ формирования значений элементов списков под названием **списковые включения** (list comprehension), или, по-простому, **генераторы списков**.

Особенность list comprehension заключается в генерации каждого нового элемента списка по некоторому заданному алгоритму. Давайте рассмотрим этот подход на конкретных примерах. Предположим, нам нужно сформировать список из квадратов целых чисел в диапазоне от 0 до N. Используя текущие знания, это можно было бы сделать следующим образом:

```
N = 6
a = [0] * N

for i in range(N):
    a[i] = i ** 2
```

Однако все то же самое можно реализовать буквально в одну строчку с помощью list comprehension:

```
a = [x ** 2 for x in range(N)]
```

Давайте подробно разберемся в синтаксисе приведенной конструкции. Вначале в квадратных скобках указывается способ вычисления текущего значения формируемого списка, а именно, команда «`x ** 2`». Далее следует цикл for, в котором формируются значения в виде переменной `x`. В частности, она будет пробегать диапазон целых значений от 0 до N-1 включительно. Причем эта временная переменная `x` будет существовать только внутри генератора списка, и недоступна за его пределами.

Вот общий принцип записи простого спискового включения. Вначале указывается команда вычисления текущего значения списка, а затем, через ключевое слово for, определяется способ изменения параметра `x`. Причем здесь можно прописывать только for, ключевое слово while указывать недопустимо. Получаем следующий синтаксис для list comprehension:

[<способ вычисления значения> for <переменная> in <итерируемый объект>]

В качестве примера запишем list comprehension для формирования последовательности, состоящей из остатков от деления на 4 чисел в диапазоне от

0 до N-1:

```
a = [x % 4 for x in range(N)]
```

Как видите, для вычисления текущего значения формируемого списка можно использовать любые доступные операции языка Python.

Еще несколько примеров. С помощью операции определения четного и нечетного значений сформируем список из булевых величин True и False:

```
a = [x % 2 == 0 for x in range(N)]
```

Или же можно сформировать значения некоторой линейной функции:

```
a = [0.5*x+1 for x in range(N)]
```

И так далее. Кроме того, для вычисления текущего значения можно вызывать функции. Пусть, например, пользователь вводит несколько целых чисел через пробел:

```
d_inp = input("Целые числа через пробел: ")
```

Тогда с помощью генератора списков мы легко можем преобразовать их в набор целых чисел:

```
a = [int(d) for d in d_inp.split()]
```

Здесь конструкция `d_inp.split()` возвращает список строк из чисел, а функция `int(d)` преобразовывает каждую строку в целое число.

Забегая вперед, отмечу, что операцию применения некоторой функции ко всем элементам итерируемой последовательности удобно выполнять с помощью функции map. Следующая команда будет давать тот же результат преобразования списка числовых строк в список целых чисел:

```
d_inp = list(map(int, input("Целые числа через пробел: ").split()))
```

Конечно, о функции map мы с вами еще будем подробно говорить. Этот пример лишь еще раз показывает, что одну и ту же задачу в Python можно реализовать самыми разными способами.

Вообще, в list comprehension можно использовать любые итерируемые объекты, в том числе и строки. Например, вполне допустима следующая запись:

```
a = [d for d in "python"]
```

Получим список из отдельных символов. Или, можно каждый символ превратить в его код:

```
a = [ord(d) for d in "python"]
```

Получим список из соответствующих кодов. Если же взять список из слов:

```
t = ["Я", "6", "Python", "выучил", "только", "за", "то", "что", "есть",
```

```
"он", "на", "этом", "канале"]
```

то генератор списка:

```
a = [d for d in t]
```

сформирует точно такой же список. То есть, строки здесь воспринимаются уже как отдельные элементы и d ссылается на каждую из них по порядку.

Если же в последнее выражение добавить функцию len:

```
a = [len(d) for d in t]
```

то получим список из длин соответствующих строк.

5.6.1. Условия в генераторах списков

В списковых включениях дополнительно можно прописывать условный оператор. Например, требуется сформировать список с числами строго меньше нуля. Записать такой генератор списка можно следующим образом:

```
a = [x for x in range(-5, 5) if x < 0]
```

Смотрите, здесь после оператора for записан необязательный оператор if. В результате, в список будут попадать только те элементы, для которых истинно условие. В нашем примере – это отрицательные значения x.

Другой пример. Нужно сформировать список только из нечетных чисел из диапазона [-5; 5), имеем:

```
a = [x for x in range(-5, 5) if x % 2 != 0]
```

Здесь также можно прописывать составные условия, например:

```
a = [x for x in range(-6, 7) if x % 2 == 0 and x % 3 == 0]
```

В результате будут выбираться числа, которые одновременно кратны 2 и 3. Или, такой пример. Нужно выбрать из списка городов только те, длина которых меньше семи:

```
cities = ["Москва", "Тверь", "Рязань", "Ярославль", "Владимир"]
a = [city for city in cities if len(city) < 7]
```

То есть, с помощью list comprehension достаточно просто формируются различные списки по некоторому заданному алгоритму. Причем, обратите внимание, так как при вычислении каждого текущего элемента списка можно прописывать любую операцию языка Python, то никто не запрещает использовать тернарную условную операцию. Давайте посмотрим, что она дает.

Пусть имеется список чисел:

```
d = [4, 3, -5, 0, 2, 11, 122, -8, 9]
```

Требуется преобразовать его в последовательность со словами «четное» и

«нечетное». Сделать это можно следующим образом:

```
a = ["четное" if x % 2 == 0 else "нечетное" for x in d]
```

Благодаря тому, что тернарная условная операция – это именно операция, а не оператор, то ее можно использовать в составе другого оператора, в частности, генератора списка. На выходе получаем список из следующих слов:

```
['четное', 'нечетное', 'нечетное', 'четное', 'четное', 'нечетное',
'четное', 'четное', 'нечетное']
```

Конечно, в полученную конструкцию можно добавить еще какое-либо условие, например, выбирать только положительные числа, получим следующую запись:

```
a = ["четное" if x % 2 == 0 else "нечетное"
     for x in d
     if x > 0
]
```

Обратите внимание, как записано списковое включение. Каждый элемент синтаксиса list comprehension допускается записывать с новой строки. Это улучшает восприятие текста программы, поэтому так рекомендуется делать для длинных конструкций.

5.6.2. Вложенные циклы в list comprehension

До сих пор мы с вами рассматривали примеры формирования списков согласно синтаксису:

```
[<способ формирования значения> for <счетчик> in <итерируемый объект>]
```

или

```
[<способ формирования значения>
   for <счетчик> in <итерируемый объект>
   if <условие>
]
```

Однако в Python допускается прописывать любое количество циклов for внутри генераторов списков. Давайте рассмотрим простейший пример такой реализации:

```
a = [(i, j) for i in range(3) for j in range(4)]
print(a)
```

После запуска программы увидим следующий результат вывода:

```
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0),
(2, 1), (2, 2), (2, 3)]
```

Для большей наглядности перепишем этот же генератор в несколько строк:

```
a = [(i, j)
     for i in range(3)
     for j in range(4)
```

```
]
```

Отсюда хорошо видно, что второй цикл вложен в первый. То есть, сначала при i = 0 отрабатывает целиком внутренний цикл, пробегая j от 0 до 3. Затем, осуществляется переход к первому циклу, i увеличивается на 1, и внутренний (вложенный) цикл повторяется заново. В результате, получаем искомые пары чисел.

Дополнительно у каждого цикла можно прописывать условия, например, так:

```
a = [(i, j)
     for i in range(3) if i % 3 == 0
     for j in range(4)
]
```

Или у обоих вместе:

```
a = [(i, j)
     for i in range(3) if i % 3 == 0
     for j in range(4) if j % 2 != 0
]
```

Используя этот подход, мы можем, например, сформировать таблицу умножения:

```
a = [f"{i}*{j} = {i*j}"
     for i in range(1, 4)
     for j in range(1, 4)
]
```

После вывода результата на экран командой:

```
for x in a:
    print(*x)
```

увидим:

```
1 * 1   =   1
1 * 2   =   2
1 * 3   =   3
2 * 1   =   2
…
```

Или можно преобразовать двумерный список в одномерный:

```
matrix = [[0, 1, 2, 3],
          [10, 11, 12, 13],
          [20, 21, 22, 23]]

a = [x
     for row in matrix
     for x in row
]
```

Обратите внимание, здесь во втором цикле используется переменная row из

первого цикла. Это вполне допустимо, т.к. второй цикл вложен в первый и в нем доступны все переменные, объявленные выше.

5.6.3. Вложенные генераторы списков

Давайте еще раз посмотрим на исходное определение генератора списка:

[<операция> for <счетчик> in <итерируемый объект>]

Как я уже отмечал, в качестве <операции> можно записывать любую конструкцию языка Python. А раз так, то ничего не мешает указать еще один генератор списка:

```
[ [генератор списка]
   for <переменная> in <итерируемый объект>
]
```

В результате получим вложенный list comprehension. Давайте посмотрим на примере, как это работает.

В самом простом варианте вложенные списковые включения можно записать следующим образом:

```
M, N = 3, 4
matrix = [[a for a in range(M)] for b in range(N)]

print(matrix)
```

На выходе получаем двумерный список вида:

```
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

Результат вполне очевиден. Здесь сначала отрабатывает первый (внешний) генератор списка (`for b in range(N)`) и переменная b стартует со значения 0. Затем, происходит формирование текущего элемента списка и выполнение переходит к вложенному генератору (`a for a in range(M)`), который создает список [0, 1, 2]. Этот список помещается как первый элемент основного списка. Далее, снова отрабатывает первый генератор и b принимает значение 1. После этого переходим к вложенному генератору, который возвращает такой же список [0, 1, 2]. И так пока не закончится работа первого (основного) генератора. В итоге получаем список, в который вложены четыре других списка.

Где может пригодиться такой подход? Например, для изменения значений двумерного списка. Давайте предположим, что имеется список:

```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

и требуется возвести все его значения в квадрат. Это можно сделать через вложенный генератор следующим образом:

```
A = [[x ** 2 for x in row] for row in A]
```

В первом генераторе происходит перебор строк (вложенных списков) матрицы A,

а во вложенном генераторе – обход элементов строк матрицы. Каждое значение возводится в квадрат и на основе этого формируется текущая строка. Обратите внимание, что во вложении мы можем использовать переменные из внешнего генератора списка, в частности, переменную row, ссылающуюся на текущую строку матрицы A.

Если бы мы здесь использовали list comprehension с вложенными циклами:

```
A = [x ** 2 for row in A for x in row]
```

то получили бы другой результат, а именно одномерный список:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

В этом ключевое различие между использованием вложенных циклов и вложенных генераторов.

Другой пример – это транспонирование матрицы A (то есть, замена строк на столбцы) с использованием вложенных генераторов. Сделать это можно следующим образом:

```
A = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
A = [[row[i] for row in A] for i in range(len(A[0]))]
```

Здесь сначала значение i = 0, и переменная row[i] пробегает первые значения строк матрицы A. В результате формируется первая строка транспонированной матрицы. Далее, i увеличивается на 1 и row[i] пробегает уже вторые элементы строк матрицы A. Получаем вторую строку транспонированной матрицы. И так для всех столбцов. На выходе формируется транспонированная матрица.

С таким видом вложений, я думаю, в целом все должно быть понятно. Другой вариант, когда генератор списка фигурирует в качестве итерируемого объекта. Да, сам по себе генератор списка, а точнее его элементы, также можно перебирать через механизм итератора и использовать в операторе for, например, следующим образом:

```
g = [u ** 2 for u in [x+1 for x in range(5)]]
```

Здесь сначала отрабатывает вложенный генератор списка, получаем список [1, 2, 3, 4], а затем, эти значения перебираются первым генератором и возводятся в квадрат. Получаем на выходе результат:

```
[1, 4, 9, 16, 25]
```

Это очень похоже на вычисление значений сложной (вложенной) функции:

$$g\left(u(x+1)\right)=\left(x+1\right)^2$$

Вот такие варианты вложений list comprehension, а также их комбинации, можно использовать в Python.

ГЛАВА VI. СЛОВАРИ, КОРТЕЖИ, МНОЖЕСТВА

Строки и списки – это не единственные коллекции данных, которые имеются в языке Python. Среди стандартных (встроенных) существуют еще три коллекции:

- Словари. Позволяют хранить данные в виде пары «ключ-значение».
- Кортежи. То же, что и списки, но относятся к неизменяемым типам данных.
- Множества. Позволяют хранить уникальный набор данных.

В этой главе мы с вами познакомимся со всеми этими типами данных, которые очень часто можно встретить в программах на языке Python. Начнем со словарей.

6.1. Словари языка Python

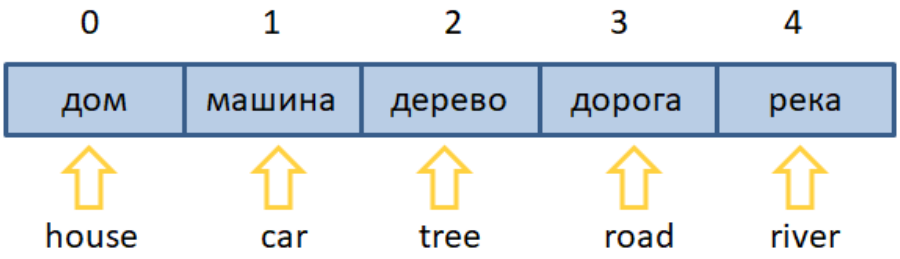
Давайте представим, что в программе требуется хранить и обрабатывать следующие зависимости:

- house -> дом
- car -> машина
- tree -> дерево
- road -> дорога
- river -> река

Как это лучше всего сделать? Если определить значения через список строк:

```
d = ["дом", "машина", "дерево", "дорога", "река"]
```

то получим коллекцию, где каждое значение (русское слово) будет связано с числом (индексом). И изменить эти индексы невозможно – они создаются автоматически самим списком. Но нам требуется обращаться к этим значениям по английским словам: house, car, tree и т.д. Как раз это позволяют делать словари. Словарь – это такая коллекция данных, которая формирует набор данных в виде ассоциативного массива (хэш-таблицы) с доступом к элементу по ключу.



Для создания словаря используется следующий синтаксис:

{key1: value1, key2: value2, …, keyN:valueN}

Причем все ключи, начиная с версии Python 3.7, сохраняют свой порядок добавления в словарь, то есть, следуют друг за другом так, как они были записаны. В более ранних версиях это условие может не соблюдаться.

Давайте объявим словарь со следующим набором данных:

```
d = {"house": "дом", "car": "машина",
     "tree": "дерево", "road": "дорога",
     "river": "река"}
```



Как видите, определение словаря можно записывать в несколько строчек. При этом, ключи будут идти строго в порядке их указания: от "house" до "river".

После того, как словарь создан, к любому его значению можно обращаться по заданному ключу. Для этого записывается имя словаря и в квадратных скобках указывается ключ. Например:

```
d["house"]   # 'дом'
```

возвращается значение «дом», которое связано с этим ключом. Если же указать несуществующий ключ:

```
d[100]       # KeyError
```

то получим ошибку KeyError.

Так как с ключом может быть связано одно какое-либо значение, то ключи в словарях уникальны. Если записать два одинаковых:

```
d = {"house": "дом", "house": "хижина", "car": "машина"}
```

то ключ «house» будет ассоциирован с последним значением «хижина».

Помимо использования фигурных скобок, словари можно формировать с помощью специальной встроенной функции dict. Например, так:

```
d2 = dict(one=1, two=2, three="3", four="4")
```

Здесь каждый аргумент образует пару «ключ-значение». При этом имена аргументов (one, two, three, four) преобразовываются в строки и образуют наборы ключей с соответствующими значениями. То есть, например, для обращения к ключу «two» словаря d2 следует прописать команду:

```
res = d2['two']   # res = 2
```

Обратите внимание, что ключ здесь – это строка "two", а не просто имя two.

Если вызвать функцию dict без параметров:

```
d = dict()
```

то формируется пустой словарь. Это эквивалентно команде:

```
d = {}
```

Основным ограничением функции dict является то, что имена аргументов можно записывать только по правилам имен переменных. Например, использовать числа в качестве имени недопустимо, поэтому следующая команда приведет к ошибке:

```
d2 = dict(1="one", two=2, three="3", four="4")  # SyntaxError
```

Но есть и положительные стороны у функции dict. С ее помощью удобно создавать словари из списков вида:

[[ключ1, значение1], [ключ2, значение2], …, [ключN, значениеN]]

Например, на основе списка:

```
lst = [[2, "неудовлетворительно"], [3, "удовлетворительно"],
       [4, "хорошо"], [5, "отлично"]]
```

словарь формируется одной командой:

```
d3 = dict(lst)
```

Получаем результат:

```
{2: 'неудовлетворительно', 3: 'удовлетворительно', 4: 'хорошо',
5: 'отлично'}
```

Причем здесь в качестве ключей можно указывать разные типы данных, не только строки, но, например, и числа. Вообще, в качестве ключей словаря можно использовать любые неизменяемые типы языка Python (а, точнее, любые хэшируемые типы данных). В наших примерах это были числа или строки. Но можно взять, например, и булево значение:

```
d = {}  # пустой словарь

d[True] = "Истина"
d[False] = "Ложь"
```

В результате, получим:

```
{True: 'Истина', False: 'Ложь'}
```

Этот пример показывает, что присваивая словарю значение с новым ключом, оно автоматически добавляется в словарь. В результате, наш изначально пустой словарь стал содержать две записи. Если же существующему ключу присвоить новое значение:

```
d[True] = 1
```

то он просто поменяет свое значение:

```
{True: 1, False: 'Ложь'}
```

Вот так можно добавлять и менять значения в словаре. То есть, словарь относится к изменяемым типам данных.

Если в качестве ключа попытаться записать список:

```
d[[1,2]] = 1   # TypeError
```

то возникнет ошибка, т.к. список – это изменяемый объект.

Ограничение на использование неизменяемых типов данных относится только к ключам словаря. Его значения могут быть любыми типами и хранить все что требуется. Например:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
```

Здесь в качестве значений используются числа, строка и список.

6.1.1. Операции и функции работы со словарями

Со словарями можно выполнять различные операции, а также применять некоторые стандартные функции языка Python. Давайте посмотрим на основные из них.

Функция **len**:

```
length = len(d)  # length = 4
```

возвращает количество элементов (количество пар «ключ-значение») в словаре.

С помощью оператора **del**:

```
del d[True]
```

выполняется удаление указанного ключа (вместе с его значением). Однако если записать несуществующий ключ:

```
del d["abc"]   # KeyError
```

то оператор del сгенерирует ошибку KeyError. Поэтому перед удалением лучше проверить существует ли данный ключ в словаре. Это можно сделать с помощью оператора **in**:

```
"abc" in d   # False
```

который возвращает булево значение True, если ключ присутствует в словаре, и False – в противном случае. Обратите внимание, оператор in проверяет наличие ключа, а не значения. Например, если добавить в словарь значение:

```
d[1] = "abc"
```

и повторно выполнить команду:

```
"abc" in d   # False
```

то снова получим значение False.

Используя операцию **not**, можно выполнять противоположную проверку на отсутствие ключа в словаре, например:

```
res = "abc" not in d    # res = True
```

Это бывает полезно, когда нужно по некоторому условию выполнить действия при отсутствии какого-либо ключа.

6.1.2. Методы словаря

Каждый объект языка Python имеет свои методы и словари здесь не исключение. Давайте рассмотрим основные методы, которые есть у этой коллекции. Начнем с метода

dict.fromkeys(список[, значение по умолчанию])

который формирует словарь с ключами, указанными в списке и некоторым значением. Например, если указать список с кодами стран:

```
a = dict.fromkeys(["+7", "+6", "+5", "+4"])
```

то получим следующий словарь:

```
{'+7': None, '+6': None, '+5': None, '+4': None}
```

Обратите внимание, что все значения ключей равны None. Это значение по умолчанию. Чтобы его изменить используется второй необязательный аргумент, например:

```
a = dict.fromkeys(["+7", "+6", "+5", "+4"], "код страны")
```

на выходе получаем словарь вида:

```
{'+7': 'код страны', '+6': 'код страны', '+5': 'код страны', '+4': 'код
страны'}
```

Следующий метод

dict.clear()

служит для очистки словаря, то есть, удаляет все его элементы. Например, команда:

```
a.clear()
```

удалит все данные из словаря `a` и он станет пустым `{}`.

Далее, для создания копии словаря можно воспользоваться методом **copy**. Например:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
d2 = d.copy()  # возвращает копию словаря d
```

Также копию можно создать и с помощью функции dict, о которой мы с вами уже говорили:

```
d2 = dict(d)
```

Результат будет абсолютно тем же, что и при вызове метода copy.

Следующий метод **get** позволяет получать значение словаря по ключу:

```
value = d.get("list")   # value = [1, 2, 3]
```

Отличие метода get от оператора:

```
value = d["list"]
```

в том, что при указании неверного ключа метод get не генерирует ошибки и по умолчанию возвращает значение None. Например:

```
res = d.get(3)   # res = None
```

Это значение можно изменить, указав его вторым аргументом в методе get, например, так:

```
res = d.get(3, False)  # res = False
```

Конечно, вместо False можно записывать любое желаемое значение, которое будет возвращаться при отсутствии ключа.

Похожий метод:

dict.setdefault(key[, default])

возвращает значение, ассоциированное с ключом key, и если его нет, то автоматически добавляет в словарь отсутствующий ключ со значением None, либо со значением default, если оно указано. Например:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}

a1 = d.setdefault("list")     # a1 = [1, 2, 3]
a2 = d.setdefault(3)          # a2 = None
a3 = d.setdefault('two', 0)   # a3 = 0
```

В итоге словарь d будет иметь вид:

```
{True: 1, False: 'Ложь', 'list': [1, 2, 3], 5: 5, 3: None, 'two': 0}
```

Видим, что в него был добавлен ключ 3 со значением None и ключ 'two' со значением 0. То есть, метод setdefault способен создать новую запись, но только в том случае, если ключ отсутствует в словаре.

Следующий метод:

dict.pop(<key>)

удаляет указанный ключ (вместе со значением) и возвращает значение удаляемого

ключа. Например, команда:

```
v = d.pop('two')   # v = 0
```

выполнит удаление ключа 'two' и возвратит значение 0. Словарь d примет вид:

```
{True: 1, False: 'Ложь', 'list': [1, 2, 3], 5: 5, 3: None}
```

Если при вызове метода pop указывается несуществующий ключ, то возникает ошибка. Например:

```
d.pop("abc")   # KeyError
```

Но при желании мы легко можем изменить это поведение, если вторым аргументом указать значение, возвращаемое при отсутствии ключа. Например:

```
res = d.pop("abc", False)  # res = False
```

Следующий метод:

dict.popitem()

выполняет удаление из словаря последней добавленной записи. Например:

```
track = {}
track['start'] = [0, -5]
track['point1'] = [10, 5]
item = track.popitem()     # item = ('point1', [10, 5])
```

Здесь возвращается пара «ключ-значение» при удалении последней добавленной записи. Если словарь пуст и вызывается метод popitem, то возникает ошибка:

```
track = {}
track.popitem()   # KeyError
```

Следующий метод:

dict.keys()

возвращает коллекцию ключей. Например:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
keys_d = d.keys()
```

Переменная keys_d будет ссылаться на коллекцию, содержащую ключи словаря:

```
dict_keys([True, False, 'list', 5])
```

Следует отметить, что по умолчанию цикл for обходит именно эту коллекцию, при указании словаря. Например:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
for x in d:
    print(x)
```

Увидим на экране строчки:

```
True
False
list
5
```

То есть, указание в цикле for словаря d эквивалентно записи:

```
for x in d.keys():
    print(x)
```

Следующий похожий метод:

dict.values()

возвращает значения словаря. Например:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
v = d.values()   #  dict_values([1, 'Ложь', [1, 2, 3], 5])

for x in d.values():
    print(x)
```

Увидим на экране строчки:

```
1
Ложь
[1, 2, 3]
5
```

Следующий метод:

dict.items()

возвращает записи в виде кортежей (ключ, значение). Например:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
items = d.items()
print(items)

for x in d.items():
    print(x)
```

На экране полчим строчки:

```
dict_items([(True, 1), (False, 'Ложь'), ('list', [1, 2, 3]), (5, 5)])
(True, 1)
(False, 'Ложь')
('list', [1, 2, 3])
(5, 5)
```

О кортежах мы с вами еще будем говорить, здесь лишь отмечу, что к элементу кортежа можно обратиться по индексу и вывести отдельно ключи и значения,

подобно спискам:

```
for x in d.items():
    print(x[0], x[1])
```

Или, используя синтаксис распаковки (множественного присваивания):

```
x, y = 1, 2
```

можно записать цикл for в таком виде:

```
for key, value in d.items():
    print(key, value)
```

что гораздо удобнее и нагляднее.

Следующий метод:

dict.update()

позволяет обновлять словарь содержимым другого словаря. Например, пусть имеются два словаря:

```
d = dict(one=1, two=2, three="3", four="4")
d2 = {2: "неудовлетворительно", 3: "удовлетворительно", 'four': "хорошо",
5: "отлично"}
```

Требуется в первый добавить содержимое второго. С помощью метода update это можно сделать следующим образом:

```
d.update(d2)
```

В результате словарь d примет вид:

```
{'one': 1, 'two': 2, 'three': '3', 'four': 'хорошо', 2:
'неудовлетворительно', 3: 'удовлетворительно', 5: 'отлично'}
```

Обратите внимание, что ключ `'four'` был заменен строкой «хорошо», а остальные несуществующие ключи были добавлены в конец словаря. При этом словарь d2 остается без изменений.

Если же нам нужно создать новый словарь, который бы объединял содержимое обоих, то, начиная с версии Python 3.9, это можно сделать командой:

```
d3 = d | d2
```

Если же требуется к одному словарю добавить другой, то это можно выполнить командой (аналог метода update):

```
d |= d2
```

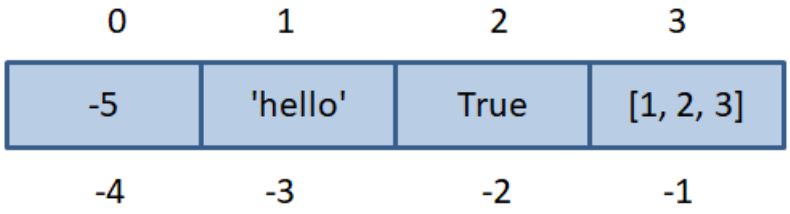
До версии Python 3.9 словари можно было объединять с помощью следующей конструкции:

```
d3 = {**d, **d2}
```

Принцип ее работы станет понятным после того, как мы познакомимся с операторами распаковки коллекций. Кроме того, ее можно считать устаревший и она приведена здесь лишь для понимания ее смысла, если вдруг она встретится вам в сторонних программах.

6.2. Кортежи (tuple)

Помимо списков в Python имеется еще одна упорядоченная коллекция данных под названием **кортеж**. В отличие от списков кортежи относятся к неизменяемым типам данных, то есть, элементы кортежей и размер кортежей менять нельзя.



Для объявления кортежа в программе достаточно перечислить значения через запятую, например, так:

```
a = 1, 2
```

или же записать их в круглых скобках (это одно и то же):

```
a = (1, 2, 3)
```

Но, обратите внимание, при определении кортежа с одним значением всегда следует ставить запятую. Например:

```
b = 1,  # кортеж из одного элемента
```

или

```
b = (1,)
```

Эта «висячая» запятая указывает, что единицу следует воспринимать как элемент кортежа, а не как число 1. Без запятой – это будет просто число:

```
digit = 1  # целое число 1
```

Вот этот момент нужно очень хорошо запомнить, т.к. начинающие программисты здесь часто делают ошибку, не прописывая висячую запятую.

Также мы с вами уже знаем, что элементы кортежа можно распаковкой сразу присвоить требуемым переменным. Например:

```
x, y = (1, 2)
```

Только здесь нужно следить за тем, чтобы число переменных совпадало с числом элементов кортежа. Если это не так, то возникнет ошибка ValueError:

```
x, y = (1, 2, 3)   # ValueError
```

Кстати, присваивание распаковкой можно выполнять не только для кортежей, но и вообще для любых коллекций. Например:

```
x, y = [1, 2]             # распаковка списка
x, y = {'a': 1, 'b': 2}   # распакова словаря (по ключам): x = 'a', y = 'b'
```

Подробнее об этом мы с вами еще будем говорить.

Далее, длину кортежа (число его элементов) можно определить с помощью известной нам функции len:

```
a = (1, 2, 3)
L = len(a)     # L = 3
```

А доступ к его отдельным элементам осуществляется так же, как и к элементам списков:

- по индексу:

```
a[0]      a[2]           a[-1]
```

- через срезы:

```
a[1:2]       a[0:-1]        a[:3]          a[1:]         a[:]
```

О срезах мы с вами уже подробно говорили, когда рассматривали списки, здесь все абсолютно так же, кроме одной операции – взятия полного среза:

```
b = a[:]  # копия не создается (для неизменяемого объекта смысла нет)
```

В случае с кортежами копия не создается. Переменная b будет ссылаться на тот же объект, что и переменная a:

```
print(id(a), id(b))   # увидим одинаковые id
```

Напомню, что для списков операция полного среза приводит к их копированию. Этот момент нужно также очень хорошо запомнить: для кортежей возвращается тот же самый объект (и вообще для всех неизменяемых объектов), а для списков создается копия.

Возможно, здесь у вас возникает вопрос. Зачем придумали кортежи, если списки выполняют ту же самую роль? И, кроме того, у списков дополнительно можно менять значения элементов? Да, все верно, по функциональности списки шире кортежей, но у последних все же есть свои преимущества. Во-первых, кортеж относится к неизменяемому типу данных, а значит, его можно использовать для «запрета» в программе менять значения элементов в коллекции. Например, следующая операция:

```
a = ('abc', True, 5, 10)
a[1] = 100    # TypeError
```

приведет к ошибке, т.к. менять значения элементов кортежей нельзя.

Во-вторых, кортежи можно использовать в качестве ключей в словарях благодаря, как раз, их неизменяемости:

```
point = (1, 2)

d = {}
d[point] = "кортеж"
```

Словарь d примет вид:

```
{(1, 2): 'кортеж'}
```

Напомню, что списки в качестве ключей использовать недопустимо, так как они относятся к изменяемым типам данных. Ключами же словаря могут выступать только неизменяемые типы.

Однако здесь есть один нюанс. Если в кортеже будет хотя бы один изменяемый элемент, например, список:

```
point = ('first', [1, 2])
```

то в качестве ключа словаря его использовать уже не получится:

```
d[point] = "кортеж"   # TypeError: unhashable type: 'list'
```

В-третьих, кортеж занимает меньше памяти, чем аналогичный список. Например:

```
Lst = [1,2,3]
t = (1,2,3)

print(lst.__sizeof__())   # 72
print(t.__sizeof__())     # 48
```

Размер кортежа оказался меньше размера списка. (Здесь для определения размера объекта в байтах был использован метод __sizeof__.) Хотя для очень длинных списков разница в объеме памяти становится не так ярко выраженной. Поэтому это третье преимущество не самое главное для кортежей.

Теперь, когда мы поняли, что кортежи играют свою значимую роль в программах на Python, вернемся к их рассмотрению.

Чтобы создать пустой кортеж достаточно записать круглые скобки:

```
a = ()
```

или воспользоваться специальной встроенной функцией:

```
b = tuple()
```

Но здесь сразу может возникнуть вопрос. Зачем создавать пустой кортеж, если он относится к неизменяемым типам данных? Это действительно так. Однако в программе с кортежами можно выполнять различные операции, формируя новые

кортежи. И здесь, в рамках логики программы, пустые кортежи могут играть свою роль. Например, операция «+» позволяет объединять два кортежа между собой, возвращая новый кортеж:

```
a = ()        # пустой кортеж
b = a + (1,)  # объединение пустого кортежа с кортежем (1,)
```

Если результат операции «+» присвоить исходной переменной, например, так:

```
a = ()
a = a + (1, 2)  # эффект изменения исходного кортежа
```

то возникает эффект как бы добавления в конец пустого кортежа новых элементов. Хотя, конечно, никакого добавления не происходит, а лишь объединение с возвратом нового сформированного кортежа.

Или же, можно построить конструкцию как бы добавления данных в начало кортежа следующим образом:

```
a = (2, 3) + a
```

Конечно, если требуется сформировать новый кортеж с добавлением в него элементов, то вместо операции:

```
a = a + ('1', '2')
```

удобнее прописывать операцию:

```
a += ('1', '2')  # то же самое, что и a = a + (кортеж)
```

Следующая операция «*» позволяет дублировать элементы кортежа, создавая новый. Например:

```
b1 = (0,)*10               # (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
b2 = ("hello", "world")*5  # ('hello', 'world', 'hello', 'world', ...)
```

Все эти операции создают новые кортежи с соответствующими значениями. А можно ли удалять отдельные элементы из кортежей? В действительности, нет, т.к. кортежи – неизменяемый тип. Если, например, попытаться выполнить команду:

```
del a[1]   # TypeError
```

то возникнет ошибка. Хотя, удалить объект целиком все же можно:

```
del a
```

В этом случае кортеж перестанет существовать, то есть, не будет даже пустого кортежа, объект удаляется целиком. Но эта команда, скорее, относится к удалению произвольных объектов и применима к любым стандартным типам данных.

Далее, с помощью функции tuple можно создавать кортежи на основе любого итерируемого объекта, например, списков и строк:

```
a = tuple([1,2,3])   # (1, 2, 3)
a = tuple("Привет!")  # ('П', 'р', 'и', 'в', 'е', 'т', '!')
```

Получается, что функция tuple превращает список в кортеж. А можно ли сделать обратное преобразование из кортежа в список? Да, для этого используется известная нам функция list, которая формирует список из любого итерируемого объекта. Например:

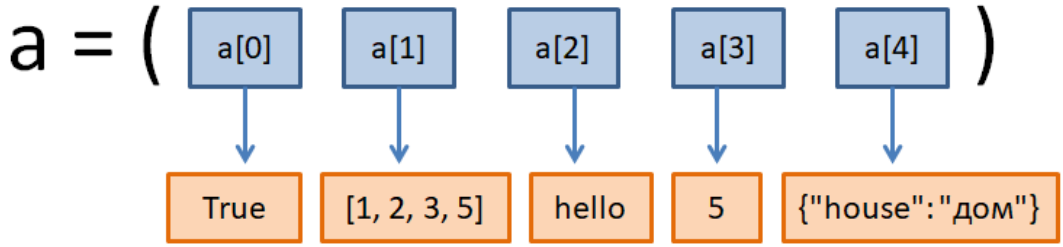
```
t = (1, 2, 3)
lst = list(t)   # lst = [1, 2, 3]
```

Вообще, кортежи могут содержать самые разные типы данных, включая другие кортежи. Например:

```
a = (1, 2, ("a", "hello"))
```

А также списки и словари:

```
a = (True, [1,2,3], "hello", 5, {"house": "дом"})
```



Причем, смотрите, если обратиться, например, к списку:

```
lst = a[1]   # lst = [1, 2, 3]
```

то будет возвращена ссылка на изменяемый объект. Следовательно, этот список даже в кортеже можно спокойно менять. Например:

```
a[1].append("5")
```

Получим кортеж с элементами:

```
(True, [1, 2, 3, '5'], 'hello', 5, {'house': 'дом'})
```

То есть, неизменность кортежа относится к его структуре элементов и неизменным значениям ссылок на объекты. Но, если сами объекты, при этом, могут меняться, то изменения можно по-прежнему вносить. То есть, обращаясь ко второму элементу кортежа, мы, фактически, имеем список, с которым возможны все его операции без каких-либо ограничений.

Объект кортежа содержит два нередко используемых метода:

- tuple.count(значение) – возвращает число найденных элементов с указанным значением;
- tuple.index(значение[, start[, stop]]) – возвращает индекс первого найденного элемента с указанным значением (start и stop – необязательные параметры;

индексы начала и конца поиска).

Эти методы работают так же, как и у списков. Пусть имеется кортеж:

```
a = ("abc", 2, [1,2], True, 2, 5)
```

Тогда метод count возвратит количество элементов с заданным значением. Например:

```
a.count("abc")  # 1
a.count(2)      # 2
a.count("a")    # 0
```

Второй метод index возвращает индекс первого найденного элемента с указанным значением. Например:

```
res = a.index(2)  # res = 1
```

Если значение не находится, например:

```
a.index(3)    # ValueError
```

то возникает ошибка. Поэтому, прежде чем его использовать, лучше проверить наличие искомого значения в кортеже. Сделать это можно с помощью операции in следующим образом:

```
res = 3 in a   # res = False
```

Операция in возвращает булево значение True, если элемент присутствует в кортеже, и False при его отсутствии.

Метод index имеет второй необязательный параметр start, который позволяет задавать индекс начала поиска:

```
a.index(2, 3)   # поиск числа 2 начинается с 3-го индекса; результат 4
```

А третий аргумент stop задает индекс, до которого идет поиск (не включая его):

```
a.index(2, 3, 5) # поиск числа 2 с 3-го индекса по 5-й (не включая его)
```

Если в заданных интервалах (индексов) искомый элемент не находится:

```
a.index(2, 3, 4)   # ValueError
```

то возникнет все та же ошибка ValueError.

6.3. Множества (set)

Формально, множество (set) – это неупорядоченная коллекция, состоящая из уникальных данных. Другими словами, в ней отсутствуют дублирующие значения.

Для задания множества используются фигурные скобки, в которых через запятую перечисляются значения элементов. Например, так:

```
a = {1, 2, 3, "hello"}
```

Это немного похоже на определение словаря, но в отличие от него, здесь не прописываются ключи, только значения. В результате получаем совершенно другой тип объекта – множество. В этом легко убедиться, выполнив команду:

```
type(a)   # <class 'set'>
```

Так как множество состоит только из уникальных значений, то попытка поместить в него несколько одинаковых значений:

```
a = {1, 2, 3, "hello", 2, 3, "hello"}
```

приведет к тому, что останутся только неповторяющиеся:

```
{1, 2, 3, 'hello'}
```

Это ключевая особенность работы данной коллекции: она автоматически отбрасывает все дубли. Кроме того, в множестве можно прописывать только неизменяемые данные, такие как:

- числа;
- булевы значения;
- строки;
- кортежи

Например:

```
a = {1, 4.5, "hi", "hi", (4, True), ("a", False)}
```

А вот изменяемые типы данных:

- списки;
- словари;
- множества

добавлять нельзя, возникнет ошибка:

```
b = {[1, 2], 3, 4}        # TypeError
b = {{1: 1, 2: 2}, 3, 4}  # TypeError
b = {{1, 2}, 3, 4}        # TypeError
```

В действительности, множество в Python реализовано на основе хэш-таблицы, и подобно словарю, в котором присутствуют только ключи. Поэтому данные в множествах должны удовлетворять тем же требованиям, что и ключи в словаре, т.е. относиться к неизменяемым типам данных (более строго, к хэшируемым данным).

Для создания множества используется встроенная функция set. Если ее записать без аргументов:

```
a = set()
```

то получим пустое множество. Но, обратите внимание, если попытаться создать пустое множество вот так:

```
b = {}  # словарь
```

то получим не пустое множество, а пустой словарь! Пустое множество создается именно с помощью функции set.

В функцию set можно передавать какой-либо итерируемый объект, например, список, значения которого будут сразу добавлены в множество (без дублирования). Например:

```
a = set([1, 2, 1, 0, -1, 2])  # {0, 1, 2, -1}
```

Или прописать строку:

```
a = set("abrakadabra")  # {'a', 'b', 'r', 'k', 'd'}
```

Тогда получим множество из уникальных символов этой строки. И так для любого итерируемого объекта, даже можно указать функцию range, например:

```
b = set(range(7))   # {0, 1, 2, 3, 4, 5, 6}
```

Обратите внимание, множества представляют собой неупорядоченную коллекцию, то есть, значения элементов могут располагаться в любом порядке. Это также означает, что с множествами нельзя выполнять операции доступа по индексу или срезам. Например, следующая команда приведет к ошибке:

```
res = b[0]   # TypeError
```

Для чего может потребоваться такая коллекция в программировании? Я, думаю, большинство из вас уже догадались – с их помощью, например, можно легко и быстро убирать дубли из данных. Представим, что некоторая логистическая фирма составила список городов, куда доставляли товары:

```
cities = ["Калуга", "Краснодар", "Тюмень", "Ульяновск", "Москва", "Тюмень",
"Калуга", "Ульяновск"]
```

И отдел статистики хотел бы получить список уникальных городов, с которыми было сотрудничество. Очевидно, это можно сделать с помощью множества:

```
unique_cities = set(cities)
```

Получим результат:

```
{'Москва', 'Ульяновск', 'Краснодар', 'Калуга', 'Тюмень'}
```

Обратите внимание, порядок городов может быть любым, так как множество – неупорядоченная коллекция данных. Однако из множества всегда можно получить список или кортеж:

```
lst = list(unique_cities)  # список на базе множества
tp = tuple(unique_cities)  # кортеж на базе множества
```

Учитывая, что множество – это итерируемый объект, то все его элементы можно перебирать с помощью цикла for, например, так:

```
for c in unique_cities:
     print(c)
```

На экране увидим строчки (порядок может быть любым):

```
Москва
Ульяновск
Краснодар
Калуга
Тюмень
```

Или, то же самое можно сделать через механизм итераторов:

```
it = iter(unique_cities)
city = next(it)   # 'Москва'
```

Далее, с помощью функции len можно получить количество элементов в множестве. Например:

```
a = {"abc", (1, 2), 5, 4, True}
length = len(a)    # length = 5
```

А для проверки наличия некоторого значения в множестве можно воспользоваться уже знакомой нам операцией in:

```
res = "abc" in a   # res = True
```

Эта операция возвращает True, если значение имеется в множестве, и False в противном случае.

Обратная проверка на отсутствие какого-либо значения делается использованием операции not, например, так:

```
res_not = 7 not in a   # res_not = True
```

6.3.1. Методы добавления/удаления элементов множества

Так как множество относится к изменяемым типам данных, то в него можно добавлять и удалять элементы. Для этого существуют специальные методы и первый такой метод **add** выполняет добавление нового значения в множество. Например, так:

```
b = {1, 2, 3}
b.add(7)
```

В результате множество b будет иметь следующий вид:

```
{1, 2, 3, 7}
```

Обратите внимание, что значение добавляется в множество только в том случае, если оно в нем отсутствует. Если попробовать, например, еще раз добавить

семерку:

```
b.add(7)
```

то множество b не изменится. При этом ошибок тоже никаких не будет.

Если в множество требуется добавить сразу несколько значений, то для этого хорошо подходит метод **update**. Например:

```
b = {1, 2, 3}
b.update(["a", "b", (1, 2)])
```

Получим:

```
{1, 2, 3, (1, 2), 'a', 'b'}
```

В качестве аргумента метода update указывается любой итерируемый объект (в нашем случае список), уникальные значения которого добавляются в множество. Также можно указать строку, кортеж, другое множество и т.п.:

```
b.update("abrakadabra")
b.update((5, 4, 10))
b.update({10, 5, 10, 11})
```

Для удаления из множества элемента по значению можно воспользоваться методом **discard**, например, так:

```
b = {1, 2, 3, 4}
b.discard(2)
```

В результате множество b будет содержать элементы:

```
{1, 3, 4}
```

Если еще раз попытаться удалить то же значение, т.е. двойку:

```
b.discard(2)
```

то ошибок не возникнет и множество не изменится.

Другой метод **remove** также выполняет удаление элемента из множества по значению. Например:

```
b.remove(4)
```

В результате множество b будет содержать элементы:

```
{1, 3}
```

Но при повторном удалении того же значения:

```
b.remove(4)  # KeyError
```

возникнет ошибка. То есть, метод remove возвращает ошибку при попытке удаления несуществующего значения. Это его единственное отличие от

аналогичного метода discard.

Еще один метод **pop** выполняет удаление одного, в общем-то, случайного элемента из множества (т.к. множество неупорядоченная коллекция) и возвращает его значение. Например:

```
b = {5, 3, 10, 8}
res = b.pop()     # res = 8
```

Если вызвать этот метод для пустого множества, то возникнет ошибка:

```
a = set()
a.pop()    # KeyError
```

Наконец, если нужно удалить все элементы из множества, то используется метод **clear** следующим образом:

```
b.clear()  # удаление всех элементов
```

В результате множество b становится пустым.

6.3.2. Операции над множествами

Из математики мы знаем, что над множествами можно выполнять следующие стандартные операции:

- пересечение множеств;
- объединение множеств;
- вычитание множеств;
- вычисление симметричной разности.

Все они доступны в языке Python. Рассмотрим их по порядку.

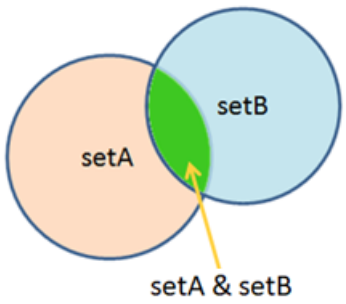
6.3.2.1. Пересечение множеств

Для простоты определим два следующих множества с целыми числами:

```
setA = {1, 2, 3, 4}
setB = {3, 4, 5, 6, 7}
```

На их основе можно построить третье множество, как результат пересечения этих двух:

```
res = setA & setB   # {3, 4}
```



На выходе получаем новое множество res, состоящее из значений общих для множеств setA и setB. Так работает операция пересечения двух множеств. При этом исходные множества setA и setB остаются без изменений. Бинарная операция «&» создает новое множество с соответствующими значениями. Однако эту же операцию можно записать и в таком виде:

```
setA &= setB
```

Это будет эквивалент команды:

```
setA = setA & setB
```

То есть, сначала вычисляется результат пересечения, который затем сохраняется в переменной setA. Возникает, как бы, эффект изменение самого множества setA.

Если же взять множество, которое не имеет общих значений с другим пересекающимся множеством, например:

```
setC = {9, 10, 11}
```

то результатом будет пустое множество:

```
res = setA & setC    # set()
```

Бинарная операция пересечения «&» двух множеств может быть также записана с использованием метода **intersection**. Например:

```
setA = {1, 2, 3, 4}
setB = {3, 4, 5, 6, 7}
res = setA.intersection(setB)   # {3, 4}
```

Это аналог команды:

```
res = setA & setB
```

Другой метод **intersection_update** вычисляет пересечение двух множеств в соответствии с унарной операцией «&=». Например:

```
setA.intersection_update(setB)    # setA = {3, 4}
setA &= setB                      # аналог предыдущего метода
```

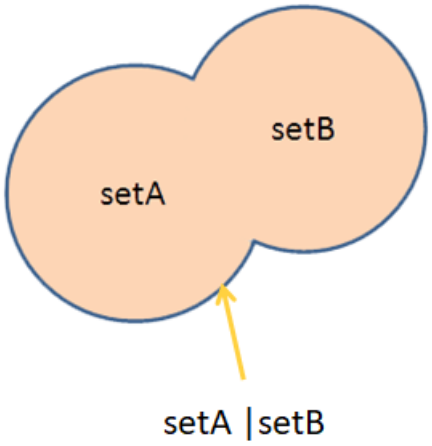
6.3.2.2. Объединение множеств

Следующая операция, которую мы рассмотрим, это объединение двух множеств. Пусть имеется два множества:

```
setA = {1, 2, 3, 4}
setB = {3, 4, 5, 6, 7}
```

Тогда результат их объединения можно вычислить с помощью операции «|» следующим образом:

```
res = setA | setB    # res = {1, 2, 3, 4, 5, 6, 7}
```



На выходе получаем новое множество res, состоящее из всех значений обоих множеств, разумеется, без дублирования.

Или же можно воспользоваться эквивалентным методом **union**:

```
res = setA.union(setB)     # res = {1, 2, 3, 4, 5, 6, 7}
```

который возвращает множество из объединенных значений. При этом сами множества setA и setB остаются без изменений.

Операцию объединения можно записать в виде:

```
setA |= setB               # setA = {1, 2, 3, 4, 5, 6, 7}
```

В этом случае результат объединения будет сохраняться в множестве setA. Множество setB останется без изменений.

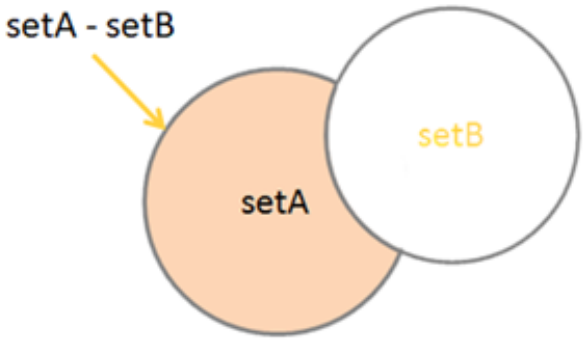
6.3.2.3. Вычитание множеств

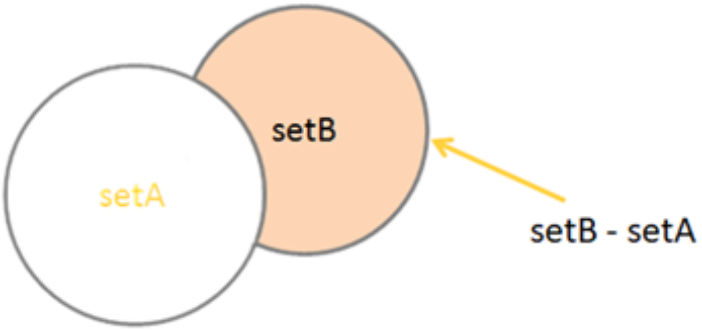
Следующая операция – это вычитание множеств. Пусть имеется два множества:

```
setA = {1,2,3,4}
setB = {3,4,5,6,7}
```

Тогда бинарная операция «-» возвратит новое множество, в котором из множества setA будут удалены все значения, повторяющиеся в множестве setB:

```
res = setA - setB     # res = {1, 2}
```





Или, наоборот, если из множества setB вычесть множество setA:

```
res = setB – setA      # res = {5, 6, 7}
```

то получим значения множества setB, из которых исключены величины, входящие в множество setA.

Два последних примера показывают, что результат работы операции вычитания зависит от расположения операндов.

Также вычитание двух множеств можно вычислить с помощью метода **difference**. Например, команда:

```
res = setA.difference(setB)    # res = {1, 2}
```

аналогична команде:

```
res = setA - setB
```

Другой похожий метод **difference_update** вычисляет разность между множествами с изменением значений непосредственно в множестве setA:

```
setA.difference_update(setB)   # setA = {1, 2}
```

Это эквивалент операции:

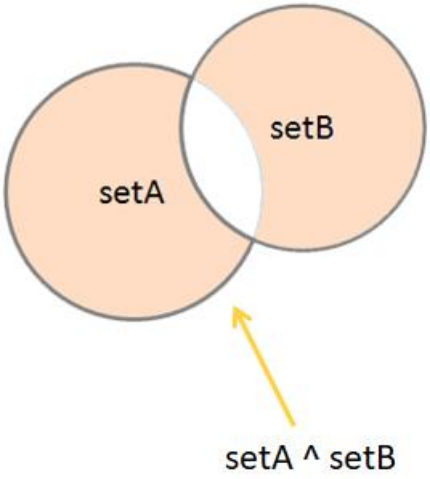
```
setA -= setB     # вычитание из setA множества setB
```

6.3.2.4. Симметричная разность

Наконец, последняя операция симметричная разность, возвращает множество, состоящее из неповторяющихся значений обоих множеств. Реализуется она следующим образом:

```
setA = {1,2,3,4}
setB = {3,4,5,6,7}
res = setA ^ setB     # res = {1, 2, 5, 6, 7}
```

На выходе получаем третье новое множество res, состоящее из значений, не входящих одновременно в оба множества.



Эту же команду можно записать и с помощью метода **symmetric_difference** следующим образом:

```
res = setA.symmetric_difference(setB)
```

А метод **symmetric_difference_update**:

```
setA.symmetric_difference_update(setB)   # setA = {1, 2, 5, 6, 7}
```

будет аналогом операции:

```
setA ^= setB
```

6.3.3. Сравнение множеств

Следующая группа операций над множествами – это операции сравнения. Как и многие другие объекты в Python множества тоже можно сравнивать между собой.

Пусть в программе заданы два следующих множества:

```
setA = {7, 6, 5, 4, 3}
setB = {3, 4, 5, 6, 7}
```

Требуется определить, раны ли они. Для этого можно воспользоваться уже знакомой нам операцией сравнения на равенство:

```
res = setA == setB      # res = True
```

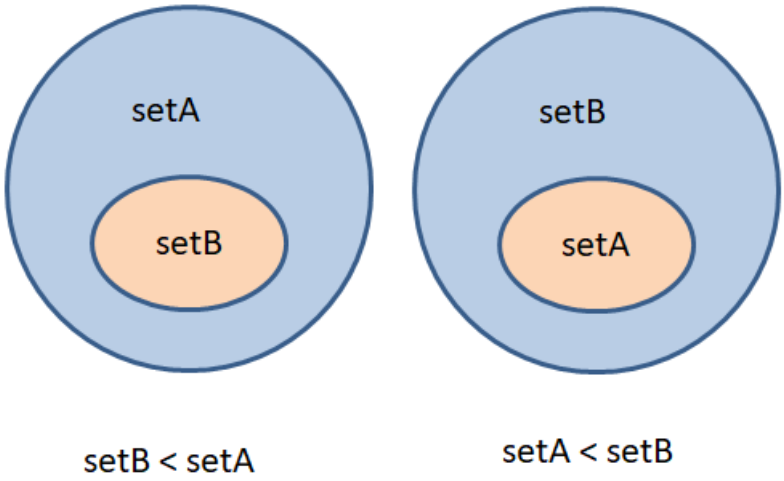
В данном случае получим значение True (истина). Почему? Дело в том, что множества считаются равными, если все элементы одного из них (setA) принадлежат второму (setB) и мощности этих множеств равны (то есть они содержат одинаковое количество элементов). Множества setA и setB удовлетворяют этим условиям, а значит, они равны.

Противоположное сравнение на неравенство записывается следующим образом:

```
r = setA != setB        # r = False
```

Следующие две операции сравнения на больше (>) и меньше (<), фактически,

определяют, входит ли одно множество в другое или нет.



Например, для двух множеств:

```
setA = {7, 6, 5, 4, 3}
setB = {3, 4, 5}
```

операции на вхождение дадут следующие результаты:

```
r1 = setB < setA     # r1 = True
r2 = setB > setA     # r2 = False
```

Но, если хотя бы один элемент множества setB не будет принадлежать множеству setA:

```
setB.add(22)    # setB = {3, 4, 5, 22}
```

то обе операции вернут False:

```
r1 = setB < setA     # r1 = False
r2 = setB > setA     # r2 = False
```

Также для равных множеств:

```
setA = {7, 6, 5, 4, 3}
setB = {3, 4, 5, 6, 7}
```

операции проверки на вхождение также вернут False:

```
r1 = setB < setA     # r1 = False
r2 = setB > setA     # r2 = False
```

Но если дополнительно проверять на равенство, то получим значение True:

```
r1 = setA <= setB    # r1 = True
r2 = setA >= setB    # r2 = True
```

Эти примеры показывают особенности выполнения различных операций сравнения множеств между собой. В целом, все логично и просто.

6.4. Генераторы множеств и словарей

Ранее мы с вами уже рассматривали генераторы списков (list comprehension), которые в самом простом случае реализуются согласно следующему синтаксису:

[<способ формирования значения> for <счетчик> in <итерируемый объект>]

Например:

```
a = [x ** 2 for x in range(1, 5)]
```

Получаем список:

```
[1, 4, 9, 16]
```

Так вот, те же самые конструкции можно прописывать и для генерации множеств и словарей.

Чтобы сгенерировать множество достаточно вместо квадратных скобок прописать фигурные, например:

```
a = {x ** 2 for x in range(1, 5)}
```

и вместо списка получим множество:

```
{16, 1, 4, 9}
```

Здесь может возникнуть вопрос. Почему было сформировано множество, а не словарь? Смотрите. Множество представляет собой набор отдельных значений, а словарь пары «ключ-значение». Так как при генерации формируются отдельные значения, то они образуют коллекцию под названием множество. Если же в генераторе формировать данные в формате «ключ-значение», например, так:

```
a = {x: x ** 2 for x in range(1, 5)}
```

то получим словарь:

```
{1: 1, 2: 4, 3: 9, 4: 16}
```

Следующий частый вопрос, где такие генераторы вообще используются? Давайте представим, что нужно выделить уникальные значения из списка:

```
d = [1, 2, '1', '2', -4, 3, 4]
```

Причем все элементы следует привести к целым числам перед записью в множество. Для решения этой задачи достаточно прописать следующий генератор множества:

```
a = {int(x) for x in d}
```

Получим результат:

```
{1, 2, 3, 4, -4}
```

Видите, как легко, просто, буквально, в одну строчку это делается. Мало того, в

некоторых реализациях генераторы работают быстрее циклов. В частности, следующая программа может работать медленнее, чем ранее приведенный генератор:

```
set_d = set()
for x in d:
    set_d.add(int(x))
```

Поэтому там, где это возможно, лучше использовать соответствующие генераторы. Они имеют более короткую и часто более понятную запись, и могут давать ускорение при работе программы. Хотя последнее не гарантируется.

То же самое и со словарем. Допустим, необходимо все ключи следующего словаря записать заглавными буквами, а значения целыми числами:

```
m = {"неудовл.": 2, "удовл.": 3, "хорошо": '4', "отлично": '5'}
```

Используя генератор словарей, получаем решение:

```
a = {key.upper(): int(value) for key, value in m.items()}
```

На выходе имеем словарь:

```
{'НЕУДОВЛ.': 2, 'УДОВЛ.': 3, 'ХОРОШО': 4, 'ОТЛИЧНО': 5}
```

Также как и со списками, в генераторах множеств и словарей можно использовать условия, вложенные циклы и вложенные генераторы. Например, нужно сформировать множество, состоящее из целых положительных чисел, используя следующий список:

```
d = [1, 2, '1', '2', -4, 3, 4]
```

Генератор множества для решения этой задачи можно записать в виде:

```
a = {int(x) for x in d if int(x) > 0}
```

Получаем результат:

```
{1, 2, 3, 4}
```

Другой пример со словарем. Допустим, требуется поменять местами ключи и значения и, кроме того, оставить только отметки от 2 до 5 включительно:

```
m = {"безнадежно": 0, "убого": 1, "неудовл.": 2, "удовл.": 3, "хорошо":
'4', "отлично": '5'}
```

Получаем следующую запись генератора для словаря:

```
a = {int(value): key for key, value in m.items() if int(value) >= 2 and
int(value) <= 5}
```

Со следующим результатом его работы:

```
{2: 'неудовл.', 3: 'удовл.', 4: 'хорошо', 5: 'отлично'}
```

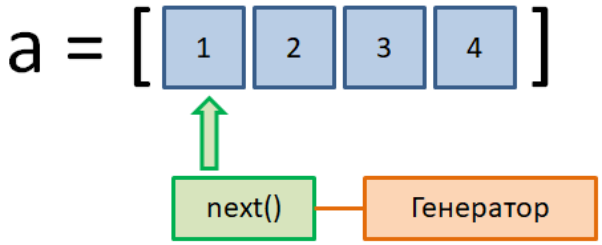
Вот так просто и легко можно использовать генераторы множеств и словарей в своих программах. Основное их преимущество перед обычными циклами – это наглядность текста программы и в ряде случаев более высокая скорость работы. Поэтому, если циклы можно относительно легко заменить генераторами, то имеет смысл это делать. Также в генераторах множеств и словарей можно применять вложенные схемы, когда один генератор вложен в другой. Причем можно комбинировать генераторы списков с генераторами множеств и словарей. Делается все по аналогии с вложенными генераторами списков, о которых мы с вами уже говорили.

6.5. Выражения-генераторы

Когда мы рассматривали различные виды генераторов, например, list comprehension, то в квадратных скобках описывали некий алгоритм формирования его значений, например:

```
a = [x for x in range(1, 5)]
```

В действительности, все эти значения были сформированы внутренним генератором, с помощью которого и заполнялся весь этот список.



То же самое происходит при генерации множеств и словарей – внутренний генератор автоматически заполняет значениями эти коллекции. Так вот, в Python можно определить такой генератор без привязки к какой-либо коллекции. Для этого используется похожий синтаксис, только все записывается в круглых скобках:

(<формирование значения> for <переменная> in <итерируемый объект>)

Обратите внимание, круглые скобки здесь не означают кортеж. Генераторов кортежей не существует. Когда прописываются круглые скобки, то формируется «чистый» генератор. Например, команда:

```
a = (x ** 2 for x in range(6))
```

создает генератор для формирования квадратов целых чисел от 0 до 5. Если вывести значение переменной a, то увидим, что это объект-генератор:

```
<generator object <genexpr> at 0x032DEAE0>
```

Но как получить конкретные значения из этого генератора? Очень просто! Генератор сам по себе является так же и итератором, то есть, его значения можно перебирать с помощью функции next:

```
res1 = next(a)  # res1 = 0
res2 = next(a)  # res2 = 1
res3 = next(a)  # res3 = 4
…
```

пока не возникнет исключение StopIteration. Соответственно, пройтись по значениям генератора можно только один раз. Повторно их перебрать уже не получится.

Так как генератор является объектом-итератором, то перебрать его можно и в цикле for, например, следующим образом:

```
gen = (x ** 2 for x in range(2, 6))

for x in gen:
    print(x)
```

Увидим на экране строчки:

```
4
9
16
25
```

Но, опять же, перебрать его можно только один раз, поэтому повторный цикл for не выведет никаких значений:

```
for x in gen:  # не сработает ни разу
    print(x)
```

Как мы уже знаем, некоторые стандартные функции Python работают с коллекциями через механизм итератора. Например:

list, set, sum, max, min и другие

Следовательно, их также можно применять и к «чистым» генераторам. Например:

```
a = (x ** 2 for x in range(6))
lst = list(a)    # [0, 1, 4, 9, 16, 25]
```

будет сформирован список из выходных значений генератора. Или так:

```
a = (x ** 2 for x in range(6))
s = set(a)       # {0, 1, 4, 9, 16, 25}
```

получим множество из соответствующих значений. По аналогии и для других функций, которые работают с коллекциями, например:

```
sum((x ** 2 for x in range(6)))   # 55
max((x ** 2 for x in range(6)))   # 25
```

Еще раз, обратите внимание, использовать эти и другие подобные им функции для одного и того же генератора можно только один раз. Например, вызывая

функцию sum дважды:

```
a = (x ** 2 for x in range(6))
res1 = sum(a)    # 55
res2 = sum(a)    # 0
```

во втором случае получим 0, так как элементы генератора нельзя пройти второй раз.

Возможно, здесь вы задаетесь вопросом, зачем нужны эти выражения-генераторы? Дело в том, что они по сравнению, например, с обычными списками не хранят в памяти все значения сразу, а генерируют их по мере необходимости, то есть, только при переходе к следующему значению. Например, если возникает необходимость оперировать очень большим списком:

```
lst = list(range(1000000000000))
```

то у компьютера может не хватить памяти для его хранения и возникнет ошибка. А вот с генераторами таких проблем нет, так как они не хранят все числа сразу, а формируют их по мере необходимости «на лету»:

```
lst = (x for x in range(1000000000000))

for i in lst:
    print(i, end=" ")
    if i > 50:
        break
```

И, кроме того, работает такая конструкция достаточно быстро. Правда, для генераторов нельзя определить число элементов через функцию len:

```
a = (x for x in range(10, 20))
length = len(a)                 # TypeError
```

или получить доступ к его отдельному элементу по индексу:

```
item = a[2]                     # TypeError
```

так как этих элементов попросту нет, они генерируются последовательно при вызове функции next.

Если все же требуется работать со значениями генератора, как с элементами упорядоченной коллекции, например, списка или кортежа, то его сначала нужно преобразовать в список или кортеж, а затем работать. Например:

```
a = (x for x in range(10, 20))
b = list(a)    # b = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Обратите внимание, если попробовать преобразовать этот же генератор к списку еще раз:

```
b2 = list(a)   # b2 = []
```

то получим пустой список, так как элементы генератора уже были перебраны в первой функции list, и сделать это второй раз не получится.

Также не получится преобразовать генератор к списку, используя квадратные скобки:

```
lst = [(x ** 2 for x in range(6))]
```

В этом случае первым элементом списка будет ссылка на объект-генератор:

```
[<generator object <genexpr> at 0x000001FAA2B74EE0>]
```

Забегая вперед, отмечу, что выйти из этой ситуации можно, воспользовавшись операцией распаковки коллекции «*» следующим образом:

```
lst = [*(x ** 2 for x in range(6))]
```

Получим список из значений генератора:

```
[0, 1, 4, 9, 16, 25]
```

Но особого смысла в такой конструкции нет.

6.6. Моржовая операция присваивания

Начиная с версии Python 3.8, появилась новая операция присваивания:

var := <вычисляемое выражение>

под названием «walrus operator», что в переводе означает «**моржовая операция**» присваивания. Обратите внимание, что это именно *операция*, а не *оператор*. То есть, ее нельзя прописать, как завершенную, законченную команду языка Python. Именно поэтому запись вида:

```
d := 77
```

приведет к ошибке SyntaxError. Она может использоваться только, как составная часть какого-либо оператора. Например, условного:

```
if d := 77:
    print(d)
```

В этом случае будет создана новая переменная d, если ранее она не существовала, ей присваивается значение 77, присвоенное значение возвращается моржовой операцией, и по условию выводится на экран с помощью функции print.

Конечно, представленный пример условного оператора нам мало что дает. На практике, чаще всего, справа от моржовой операции указываются не конкретные значения, а вычисляемое выражение, например, какая-либо функция. Например:

```
if d := int(input()):
    print(d)
```

Это условие будет срабатывать, пока пользователь вводит с клавиатуры

ненулевые целые числа. Если ввести 0, то оно интерпретируется, как False, и функция print не выполняется.

Однако если бы потребовалось выводить только положительные введенные целые числа, с условием записанным в виде:

```
if d := int(input()) > 0:
    print(d)
```

то при вводе любого положительного числа, переменная d принимала бы булево значение True, а не введенное число. Почему так работает приведенная конструкция? Дело в том, что приоритет моржовой операции один из самых низких, то есть, она отрабатывает в последнюю очередь. Поэтому вначале выполняется чтение целого значения, затем, его сравнение с нулем и только после этого полученное булево значение присваивается переменной d. Отсюда мы и видим такой результат работы этой программы. Чтобы переменная d продолжала хранить именно числовое значение, введенное с клавиатуры, моржовую операцию присваивания следует заключить в круглые скобки:

```
if (d := int(input())) > 0:
    print(d)
```

6.6.1. Преимущества моржовой операции присваивания

В чем же преимущества этой операции по сравнению с обычным оператором присваивания? Как мы говорили вначале, моржовая операция – это именно операция, а не завершенный оператор. То есть, ее можно прописывать, как выражение в любом другом операторе языка Python. При этом она по-прежнему позволяет присваивать переменной указанные значения и создавать новые переменные, если ранее они не существовали. А вот обычный оператор присваивания в нашем примере прописать было бы нельзя. Следующая программа:

```
if (d = int(input())) > 0: # ошибка
    print(d)
```

приведет к синтаксической ошибке. Хорошо, но мы можем легко это исправить и записать программу в виде:

```
d = int(input())
if d > 0:
    print(d)
```

Поэтому остается вопрос, когда моржовая операция действительно может давать какое-либо преимущество? Для ответа на этот вопрос, давайте запишем программу чтения целых чисел с клавиатуры и подсчета их суммы, пока не будет введено число -1. Это можно реализовать следующим образом:

```
s = 0
d = int(input())
while d != -1:
```

```
    s += d
    d = int(input())
```

Или, как вариант, начальное значение d присвоить нулю. В рамках нашей задачи результат будет тем же самым. Однако, используя моржовую операцию, эту же программу можно записать в более элегантном виде:

```
s = 0
while (d := int(input())) != -1:
    s += d
```

Или, другой пример. Нужно среди первых n целых чисел выделить те, квадраты которых меньше n и сохранить эти квадраты в виде списка:

```
n = 10
lst = [b for x in range(n) if (b := x * x) < n]
```

С помощью моржовой операции это удобно реализовать, вычисляя квадрат x только один раз.

Или можно описать такую конструкцию:

```
lst2D = [
    row1 := [1, 2, 3],
    row2 := [4, 5, 6],
    row3 := [7, 8, 9],
]

row2[1] = 100 # изменение центрального элемента
```

Получаем двумерный список с набором переменных row1, row2, row2, ссылающихся на каждую отдельную строку (вложенный список). Затем, через эти переменные можно обращаться к соответствующему списку и менять значения.

То есть, главное назначение моржовой операции присваивания – это упрощение текста программы для чтения и дальнейшего редактирования. На скорость работы она практически не влияет.

6.5.2. Недостатки моржовой операции присваивания

Однако любой инструмент можно использовать, как во благо, так и во вред. И моржовую операцию также следует применять только по мере необходимости, когда она явно напрашивается при реализации алгоритмов. Сами разработчики Python указывают, что следует ограничиваться использованием этой операции в целях сокращения объема вычислений или более лучшей читаемости программного кода. В противном случае она легко может привести к непредвиденным ошибкам и запутанным программным реализациям. Например, следующий фрагмент программы:

```
if (a := int(input())) % 2 == 0 and (b := abs(a)) < 30:
    print(a, b)
else:
```

```
    print(b)
```

содержит потенциальную ошибку. Здесь переменная b может не существовать при переходе к ветке else. Например, если значение a будет не кратно 2, то вторая часть условия по and отрабатывать не будет, и, как следствие, переменная b не создается (если ее ранее не было). Хотя, при a кратном 2, никаких ошибок не возникнет, обе переменные a, b будут существовать в ветке else.

Другой пример, когда применения моржовой операции снижает восприятие текста программы:

```
var = (d := 77) + 10 / 2
```

Здесь в одном операторе (в одной строчке) сразу создаются две переменные, причем визуально это воспринимается не очень хорошо. Мало того, выражения могут быть много сложнее и в них легко потерять объявление отдельных переменных. Таких конструкций следует избегать.

Наконец, из-за низкого приоритета моржовой операции присваивания, конструкции вида:

```
if not a := int(input()):
    print(a)

b = d := 10
```

приводят к синтаксическим ошибкам. Здесь обязательно нужно моржовую операцию заключать в круглые скобки:

```
if not (a := int(input())):
    print(a)

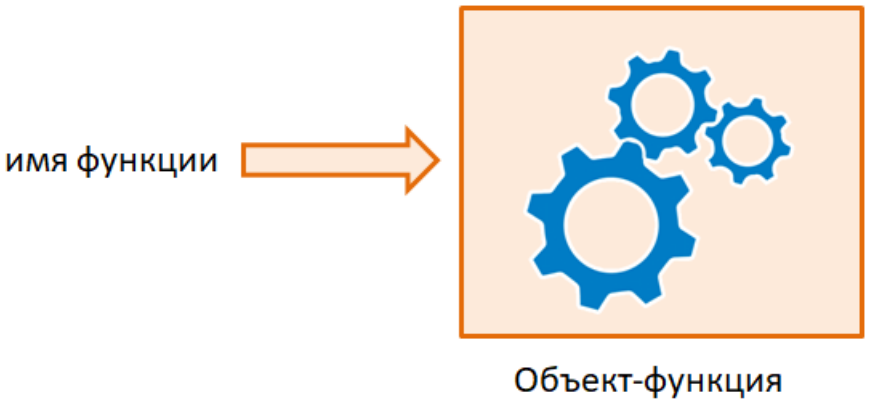
b = (d := 10)
```

Учитывая все эти особенности и недостатки, моржовую операцию присваивания следует использовать крайне осмотрительно и, как указали разработчики, лишь в целях повышения производительности и ясности текста программ. От себя добавлю, что об использовании этой операции следует задумываться только в тех случаях, когда она явно напрашивается в программный код.

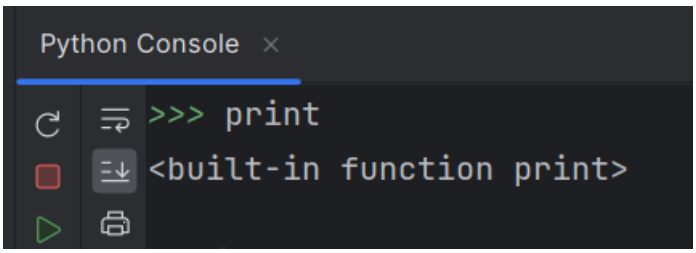
ГЛАВА VII. ФУНКЦИИ

7.1. Что такое функции. Их объявление и вызов

Мы начинаем с вами одну из ключевых и важных тем в программировании – функции. Что это такое? Условно, функцию можно представить как активный объект, который не просто хранит какие-либо данные, а выполняет заданный фрагмент программы. Ссылка на такой объект называется **именем функции**.



Например, знакомая нам функция print выполняет вывод в консоль переданных ей данных. Имя этой функции следует воспринимать, как ссылку на свой, определенный объект-функцию. Действительно, если в интерактивном режиме языка Python напечатать имя print:



то увидим, что оно ссылается на встроенную функцию:

```
<built-in function print>
```

Чтобы активизировать программу, заложенную в объект-функции, нужно поставить круглые скобки после ее имени:

```
print()
```

В результате выполняется определенный код этой функции и в консоль выводится пустая строка.

Круглые скобки, прописанные после имени функции, являются **оператором вызова**. То есть, само по себе имя – это ссылка на функцию, а имя функции с круглыми скобками – это уже вызов функции. Этот момент нужно очень хорошо и четко понимать, чтобы уметь правильно обращаться с функциями при построении программ.

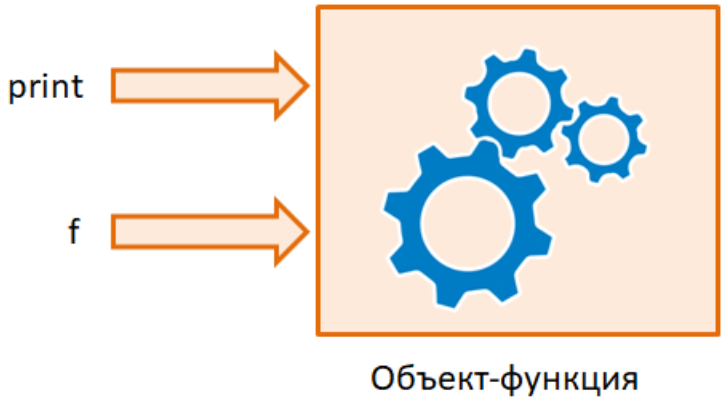
Итак, раз print – это ссылка на функцию, то что нам тогда мешает создать еще

одну ссылку на тот же объект-функцию с помощью оператора присваивания, например, следующим образом:

```
f = print
```

А затем вызвать функцию print через ее второе имя f:

```
f("hello")
```



Такая конструкция действительно будет работать. Теперь функцию print можно вызывать и через имя (переменную-ссылку) f. А можно ли тогда исходному имени функции print присвоить другой объект, например, строку? Выполним команду:

```
print = "это была функция print"
```

Ошибки никакой не произошло и переменная print теперь ссылается на присвоенную строку. А прежнюю функцию можно вызывать только через имя f следующим образом:

```
f("hello")    # печатает строку "hello"
f(print)      # печатает строку "это была функция print"
```

То есть, имя функции – это просто ссылка на объект-функцию и не более того. Конечно, заменять стандартные имена функций на другие без острой необходимости не стоит. Лучше даже совсем этого не делать.

Хорошо, мы теперь знаем, как правильно понимать функции в Python. Но зачем они нужны? Разве нельзя писать программы без них? Конечно можно. И самые первые языки программирования их не имели. Но писать программы без функций это все равно, что поехать в другой город на лошади, а не на автомобиле. С функциями программы становятся более модульными, а процесс программирования значительно упрощается. За счет чего это происходит?

Давайте представим, что мы пишем программу, в которой постоянно нужно делать отправку электронных писем по одному и тому же алгоритму. В результате, у нас будут постоянно появляться одни и те же строчки кода для отправки писем. Такое дублирование программного кода негативно влияет на надежность программ, возможность их быстрого редактирования, визуального восприятия и т.п. В программировании дублирование кода кратко называется:

что в переводе означает «не повторяйся». Как только, вы замечаете в программе повторение (дублирование) кода, значит ее архитектура построена неверно и структуру следует пересмотреть. Повторяемость кода – это довольно значимая (хотя и не критическая) ошибка при построении программ.

Так вот, чтобы устранить дублирование при отправке электронных писем, повторяющийся фрагмент можно поместить в функцию, а затем вызывать ее в нужных местах программы. Это первое, для чего были введены функции в языки программирования – устранять дублирование кода.

7.1.1. Объявление и вызов пользовательских функций

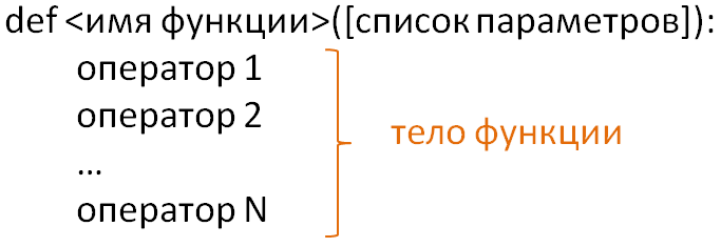
В Python существует множество стандартных, встроенных функций, но на все случаи жизни их не предусмотришь. Поэтому программист может сам создавать свои собственные по мере необходимости. Для этого используется следующий синтаксис:

```
def <имя функции>([список параметров]):
        оператор 1
        оператор 2
        …
        оператор N
```

Здесь имя функции придумывается программистом подобно именам переменных и, так как функция – это определенное действие, то в ее имени рекомендуется использовать глаголы, например:

go, show, get, set и т.п.

Далее идет набор операторов, которые образуют **тело функции**. Именно они выполняются при ее вызове.



Давайте, в качестве примера, зададим простую функцию, которая будет имитировать отправку письма:

```
def send_mail():
    text = "Уважаемый, Сергей Балакирев! Я так и не понял, что такое
функция. Объясните лучше!"
    print(text)
```

Обратите внимание, имя функции send_mail – это фраза-глагол, действие. Кроме того, два разных слова send и mail отделены символом подчеркивания. Я его

записал для лучшей читаемости текста программы. И, глядя на название функции, сразу понятно, что она должна делать. Это рекомендуемый подход – называть переменные и функции понятными именами.

Тело функции send_mail состоит из двух команд:

- переменной text присваивается строка;
- функция print выводит строку text на экран.

Обратите внимание, что эти две команды образуют единый блок операторов тела функции, так как у них единый отступ от левого края (обычно четыре пробела). Здесь используется тот же синтаксис, что и в условных операторах или операторах циклов.

Итак, мы с вами объявили функцию send_mail. Но, если сейчас запустить эту программу, то в консоли ничего не отобразится. Это означает, что функция не сработала. Дело в том, что на данном этапе мы лишь определили функцию в программе, но не вызывали. Как было сказано ранее, для вызова функции нужно записать ее имя и поставить круглые скобки (оператор вызова функции):

```
send_mail()
```

Целиком программа теперь выглядит следующим образом:

```
def send_mail():
    text = "Уважаемый, Сергей Балакирев! Я так и не понял, что такое
функция. Объясните лучше!"
    print(text)


send_mail()
```

При ее запуске видим заветное сообщение:

```
Уважаемый, Сергей Балакирев! Я так и не понял, что такое функция. Объясните
лучше!
```

Обратите внимание, что между объявлением функции и последующими командами рекомендуется прописывать две пустые строчки. Это оформление в соответствии с рекомендацией стандарта PEP8. В интегрированной среде PyCharm, чтобы привести текст программы в соответствие с этим стандартом, достаточно нажать комбинацию клавиш Ctrl + Alt + L.

После того, как объект-функция send_mail определена в программе, ее можно вызывать множество раз:

```
send_mail()
send_mail()
```

Соответственно, при двух вызовах, она сработает два раза подряд и на экран будут выведены две строчки:

```
Уважаемый, Сергей Балакирев! Я так и не понял, что такое функция. Объясните лучше!
Уважаемый, Сергей Балакирев! Я так и не понял, что такое функция. Объясните лучше!
```

Обратите внимание, что вызывать функцию можно только после ее объявления. То есть, сначала сделать вызов, а потом объявить не получится, возникнет ошибка, что имя send_mail не определено. Нужно сначала объявлять функции и только потом их вызывать.

7.1.2. Параметры функций

Сейчас у функции send_mail нет никаких параметров. Давайте добавим один, через который будем передавать имя отправителя. Для этого в круглых скобках прописывается параметр с именем from_name (это имя мы придумываем сами, как и имена переменных) и затем в многострочной F-строке указывается это имя внутри сообщения:

```
def send_mail(from_name):
    text = f"""Уважаемый, Сергей Балакирев!
Я так и не понял, что такое функция.
Объясните лучше!
Ваш навсегда, {from_name}!"""

    print(text)
```

Теперь функцию send_mail следует вызывать с одним аргументом для параметра from_name – именем отправителя. По нашей задумке этот аргумент должен быть строкой, поэтому команду вызова функции запишем следующим образом:

```
send_mail("Иван Иванович")
```

После запуска программы увидим сформированное сообщение:

```
Уважаемый, Сергей Балакирев!
Я так и не понял, что такое функция.
Объясните лучше!
Ваш навсегда, Иван Иванович!
```

Обратите внимание на используемую терминологию. Определение внутри функции называется **параметром**, а передаваемое значение при вызове функции **аргументом**. В дальнейшем, мы будем часто использовать эти слова, поэтому важно четко запомнить их смысл. Второй важный момент заключается в том, что при вызове функции, ее параметр **ссылается** на переданный аргумент. В итоге, в теле функции переменная (параметр) from_name принимает значение "Иван Иванович", которое и подставляется в шаблон сообщения.

А что будет, если теперь попытаться вызвать эту же функцию без аргументов, так как мы это делали вначале? Если прописать команду:

```
send_mail()    # TypeError
```

то возникнет ошибка, что функция ожидает один аргумент, а ей ничего не передается. То же самое будет, если передать больше одного аргумента, например, два:

```
send_mail("Иван Иванович", "Сергей Балакирев")
```

То есть, при вызове функции нужно передавать ровно столько аргументов, сколько параметров в ней определено (за исключением параметров с начальными значениями, о которых речь еще впереди).

Давайте пропишем еще один, второй параметр для указания возраста отправителя следующим образом:

```
def send_mail(from_name, age):
    text = f"""Уважаемый, Сергей Балакирев!
Я так и не понял, что такое функция.
Объясните лучше!
Ваш, навсегда {from_name}! И не судите строго, мне всего {age} лет."""

    print(text)
```

Как видите, параметры записываются в круглых скобках через запятую и принимают значения аргументов, переданных при вызове функции. Например:

```
send_mail("Иван Иванович", 7)
send_mail("Иван Иванович", "Сергей Балакирев")
```

При таком вызове на первый аргумент всегда будет ссылаться первый параметр from_name, а на второй аргумент – параметр age. То есть, порядок записи аргументов имеет значение. Также, обратите внимание, что при втором вызове функции send_mail вместо числа указана строка. Параметру age без разницы тип данных, на который он ссылается, так как в Python используется динамическая типизация. В итоге, второе сообщение несколько бессмысленно, но, тем не менее, функция срабатывает без ошибок.

7.2. Оператор return в функциях

После того, как мы с вами научились объявлять и вызывать функции с параметрами и без них, сделаем следующий шаг и разберемся, как функции могут возвращать результаты своих вычислений.

Пусть в тексте программы имеется объявление следующей функции:

```
def send_mail():
    text = "Уважаемый, Сергей Балакирев! Я так и не понял, что такое
функция. Объясните лучше!"

    print(text)
```

Ниже, после нее, объявим еще одну функцию, которая бы вычисляла квадратный корень из положительного числа. В результате текст программы примет

следующий вид:

```
def send_mail():
    text = "Уважаемый, Сергей Балакирев! Я так и не понял, что такое
функция. Объясните лучше!"

    print(text)


def get_sqrt(x):
    res = None if x < 0 else x ** 0.5
```

Обратите внимание, что в соответствии со стандартом PEP8 каждое объявление функции разделяется двумя пустыми строчками. Это рекомендуемая практика, чтобы текст программы визуально хорошо воспринимался.

Новая функция get_sqrt (имя придумываем сами) имеет один параметр x, в который предполагается передавать различные числа. В теле этой функции с помощью тернарной условной операции вычисляется квадратный корень, но только для положительных чисел. Если переданное число отрицательное, то возвращается значение None.

Вызвать эту функцию можно командой:

```
result = get_sqrt(49)
```

То есть, переменной res присваивается результат вызова функции get_sqrt. Что такое результат работы функции мы сейчас узнаем. Давайте выведем на экран значение переменной result:

```
print(result)
```

Видим значение None:

```
None
```

Фактически, это означает, что функция get_sqrt не вернула никакого результата! Но почему? Мы же в теле этой функции записали вычисление квадратного корня? И для положительного аргумента 49 должны получить значение 7.0? Дело в том, что внутри функции get_sqrt нужно явно указать значение, которое функция возвращает. Делается это с помощью специального оператора **return**, после которого через пробел указываются возвращаемые данные. В нашем примере следует прописать переменную res:

```
def get_sqrt(x):
    res = None if x < 0 else x ** 0.5
    return res
```

Теперь, после запуска программы, переменная result ссылается на вычисленное значение 7.0:

```
7.0
```

В деталях это работает так. Оператор `return res` вернул объект с вычисленным значением 7.0, а значит, и функция get_sqrt вернула это число. Затем, с помощью оператора присваивания, была создана переменная result со ссылкой на этот объект (число 7.0).

Также, обратите внимание, что параметр `x` и переменная res существуют только в момент вызова функции get_sqrt и пропадают за ее пределами (после завершения вызова). Об этом мы еще подробно будем говорить. Пока просто запомните, что параметры и переменные, объявленные внутри какой-либо функции, существуют только в момент ее вызова и пропадают после завершения ее работы. Поэтому, при попытке вывода переменной res за пределами функции get_sqrt:

```
def get_sqrt(x):
    res = None if x < 0 else x ** 0.5
    return res


print(res)
```

получим ошибку:

```
NameError: name 'res' is not defined
```

Особенность работы оператора return заключается в том, что он не только возвращает указанные данные, но и приводит к завершению выполнения тела функции, даже если после оператора return прописаны какие-либо команды. Например, если в функции get_sqrt записать два подряд идущих оператора return:

```
def get_sqrt(x):
    res = None if x < 0 else x ** 0.5
    return res
    return x


result = get_sqrt(100)
print(result)
```

то при выполнении этой программы на экране увидим:

```
10.0
```

Это произошло, как раз по той причине, что функция get_sqrt, выполнив первый оператор return, вернула вычисленное значение res и завершила свою работу. Второй оператор return выполняться никогда не будет. Даже интегрированная среда PyCharm подчеркивает этот последний оператор, сообщая об его бессмысленности.

То же самое увидим, прописывая любой другой оператор, после первого return, например, функцию print:

```
def get_sqrt(x):
    res = None if x < 0 else x ** 0.5
    return res
    print(x)
```

Функция print внутри get_sqrt никогда не будет выполнена. А вот если print записать до оператора return:

```
def get_sqrt(x):
    res = None if x < 0 else x ** 0.5
    print(x)
    return res
```

то дополнительно увидим вывод значения параметра x:

```
100
10.0
```

Но, что делать, если в функции get_sqrt нужно вернуть сразу два значения: res и x? Самый распространенный вариант – это поместить возвращаемые переменные в кортеж следующим образом:

```
def get_sqrt(x):
    res = None if x < 0 else x ** 0.5
    return (res, x)
```

Причем круглые скобки обычно не пишут, а сразу указывают переменные через запятую:

```
def get_sqrt(x):
    res = None if x < 0 else x ** 0.5
    return res, x
```

Это общепринятая практика. Теперь, при вызове этой функции, будет возвращаться кортеж с данными:

```
res = get_sqrt(49)       # res = (7.0, 49)
a, b = get_sqrt(100)     # a = 10.0    b = 100
```

Обратите внимание, что во второй команде сразу выполнено присваивание распаковкой. Если нам известно, что функция возвращает строго два значения в коллекции, то их можно сразу распаковать в отдельные переменные и затем использовать в программе. Это часто бывает очень удобно.

7.3. Варианты объявлений и вызовов функций

Давайте в программе объявим функцию для определения максимального значения из двух чисел:

```
def get_max2(a, b):
    return a if a > b else b
```

Казалось бы, что может быть примечательного в такой простой функции? Но, не спешите, сейчас перед вами откроется некоторая «магия». Вызвать эту функцию

можно очевидным образом, например, так:

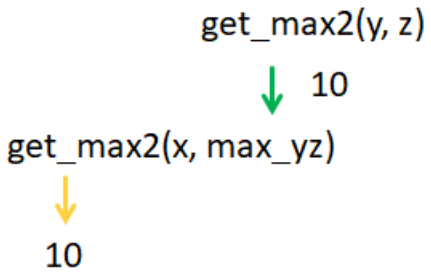
```
x, y = 5, 7
print(get_max2(x, y))
```

После запуска в консоли увидим значение 7:

```
7
```

Теперь усложним задачу и будем искать максимум среди трех чисел. Если помните, то ранее мы для этого использовали не очень читаемые вложенные условия. Здесь же поступим иначе. Определим три переменные и воспользуемся все той же функцией get_max2:

```
x, y, z = 5, 7, 10
result = get_max2(x, get_max2(y, z))
print(result)   # 10
```



Вот это и есть обещанная «магия». Сначала будет вызвана функция, записанная в качестве аргумента, которая возвратит максимальное среди чисел 7 и 10, то есть, значение 10, а затем, вызывается первая функция, которая определяет максимум из чисел 5 и 10. Соответственно, на выходе получаем результат 10, который и выводится в консоль.

Этот пример показывает, что вызов одной функции можно записывать в аргументах другой функции (или той же самой). Мало того, мы можем полученную конструкцию использовать и при объявлении новой функции, вычисляющий максимум среди трех чисел:

```
def get_max3(a, b, c):
    return get_max2(a, get_max2(b, c))
```

А затем вызвать ее с тремя аргументами:

```
print(get_max3(x, y, z))
```

Видите, как элегантно можно решить задачу поиска наибольшего среди трех значений, используя вложенный вызов функций.

Вообще, объявление функций в Python очень похоже на объявление переменных. Пусть, например, по некоторому условию переменная get_rect принимает числовые значения 1 или 2:

```
PERIMETR = True
```

```
if PERIMETR:
    get_rect = 1
else:
    get_rect = 2
```

Так вот, то же самое можно делать и с функциями, только вместо переменной get_rect объявлять разные функции, например, так:

```
if PERIMETR:
    def get_rect(a, b):
        return 2 * (a + b)
else:
    def get_rect(a, b):
        return a * b
```

В результате в программе (при PERIMETR = True) сформируется функция, которая вычисляет периметр прямоугольника:

```
res = get_rect(1.5, 3.8)    # res = 10.6
```

Если же переменную PERIMETR изменить на False, то будет объявляться функция, вычисляющая площадь прямоугольника:

```
res = get_rect(1.5, 3.8)    # res = 5.69999
```

Такую гибкость определения функций дает язык Python. Хотя, злоупотреблять данным подходом не стоит, т.к. это несколько снижает понимание логики работы программы. Все же функции следует прописывать явно со строго определенной логикой их работы.

Если в нашей программе убрать условие и два раза подряд определить одну и ту же функцию:

```
def get_rect(a, b):
    return 2 * (a + b)


def get_rect(a, b):
    return a * b
```

то ошибок никаких не возникнет, но имя get_rect будет связано со вторым объектом-функцией. Соответственно, команда:

```
res = get_rect(1.5, 3.8)
```

будет вычислять площадь прямоугольника.

Давайте рассмотрим еще один пример использования собственных функций. Пусть функция с именем even возвращает булево значение True для четных чисел, и False для нечетных:

```
def even(x):
```

```
    return x % 2 == 0
```

А затем в цикле воспользуемся ей для определения четности числа:

```
for i in range(1, 20):
    if even(i):
        print(i)
```

После запуска программы увидим в консоли отображение только четных чисел:

```
2
4
6
8
...
```

Все эти примеры показывают, что вызовы функций можно прописывать в самых разных конструкциях языка Python. И в аргументах других функций, и в условных операторах, и независимым образом, присваивая результат их работы отдельным переменным.

7.4. Алгоритм Евклида для нахождения НОД

Давайте теперь рассмотрим один практический пример использования функций для решения задачи нахождения наибольшего общего делителя (НОД) для двух натуральных чисел `a` и `b`. Причем мы не просто напишем алгоритм, а еще выполним его тестирование с применением тестирующей функции. То есть, это будет полноценный пример, показывающий принцип разработки программ с использованием функций и тестов.

Но вначале пару слов о самом алгоритме Евклида, о принципе его работы. Существует медленная и быстрая его реализации. Медленная реализация более понятная, поэтому начнем с нее.

Пусть, например, даны два натуральных числа: a = 18 и b = 24. Чтобы вычислить для них НОД будем действовать следующим образом. Из большего значения вычтем меньшее и результат сохраним в переменной с большим значением, то есть, в b. Фактически, это означает, что выполняется операция: b = b - a. Получаем два значения a = 18, b = 6. Для них повторяется тот же самый процесс. Здесь большее значение уже хранится в переменной `a`, поэтому, корректируем ее значение, вычитая меньшее. Получаем новую пару a = 12, b = 6. Снова повторяем этот процесс и видим, что a = 6, b = 6, то есть, переменные равны. В этом случае алгоритм останавливается и вычисленный НОД(18, 24) равен 6.

Данный алгоритм можно представить следующим псевдокодом:

```
пока a != b
         находим большее среди a и b
         уменьшаем большее на величину меньшего

выводим полученное значение величины a (или b)
```

На языке Python его можно записать следующим образом:

```
def get_nod(a, b):
    """Вычисляется НОД для натуральных чисел a и b
        по алгоритму Евклида.
        Возвращает вычисленный НОД.
    """
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a

    return a
```

Здесь объявлена функция get_nod с двумя параметрами a и b. В теле этой функции вначале прописана многострочная строка с описанием ее работы. Так рекомендуется делать для ключевых функций программы, чтобы другие программисты могли быстро понимать, как их применять на практике. После описания следует сам алгоритм Евклида.

Выполним эту функцию со значениями аргументов 18 и 24:

```
res = get_nod(18, 24)
print(res)
```

Видим в консоли верное значение 6.

Это пример правильного оформления ключевых функций программы. Мало того, встроенная функция:

```
help(get_nod)
```

позволяет выводить описание указанных функций. И мы видим в консоли сформированное сообщение:

```
Help on function get_nod in module __main__:

get_nod(a, b)
    Вычисляется НОД для натуральных чисел a и b
    по алгоритму Евклида.
    Возвращает вычисленный НОД.
```

Это очень удобно, особенно при групповой работе над проектом.

После того, как функция определена, ее следует протестировать и убедиться в корректности возвращаемых результатов. Для этого тестировщик создает свою вспомогательную функцию. Используя наши текущие знания, мы ее опишем следующим образом:

```
import time
```

```
def test_nod(func):
    # -- тест №1 ------------------------------
    a = 28
    b = 35
    res = func(a, b)
    if res == 7:
        print("#test1 - ok")
    else:
        print("#test1 - fail")

    # -- тест №2 ------------------------------
    a = 100
    b = 1
    res = func(a, b)
    if res == 1:
        print("#test2 - ok")
    else:
        print("#test2 - fail")

    # -- тест №3 ------------------------------
    a = 2
    b = 100000000

    st = time.time()
    res = func(a, b)
    et = time.time()
    dt = et - st
    if res == 2 and dt < 1:
        print("#test3 - ok")
    else:
        print("#test3 - fail")
```

В первых двух тестах проверяется корректность вычисленных значений, а в третьем скорость работы функции. Конечно, это довольно примитивное тестирование, показывающее лишь принцип разработки программы, но для учебных целей вполне достаточно.

Для запуска тестирующей функции выполним команду:

```
test_nod(get_nod)
```

Увидим в консоли следующие строчки:

```
#test1 - ok
#test2 - ok
#test3 - fail
```

Первые два теста прошли успешно, а третий был провален, так как функция get_nod слишком долго вычисляла результат для пары чисел a = 2, b = 100000000. Реализация алгоритма оказалась слишком медленной. Давайте его ускорим. Как это сделать? Смотрите, если взять два числа a = 2 и b = 100, то по медленному алгоритму нужно делать многочисленные вычитания из значений b значение a,

пока величины не сравняются. То есть, здесь, фактически, вычисляется остаток от вхождения двойки в сотню, а это есть не что иное, как операция:

$$b = b \% a = 0$$

И никаких циклических вычитаний! Это, очевидно, будет работать намного быстрее. При этом, как только получаем остаток равный нулю, то НОД – это значение меньшей переменной, то есть, в нашем примере 2.

То же самое будет справедливо и для предыдущих значений a = 18, b = 24. Получаем серию вычислений:

1. b = 24 % 18 = 6
2. a = 18 % 6 = 0

И НОД(18, 24) = 6. Видите, как это быстро и просто! На уровне псевдокода быстрый алгоритм Евклида можно записать так:

```
пока меньшее число больше 0
         большему числу присваиваем остаток от деления на меньшее число
выводим большее число
```

Реализуем его в виде функции get_fast_nod на языке Python:

```
def get_fast_nod(a, b):
    """Вычисляется НОД для натуральных чисел a и b
        по быстрому алгоритму Евклида.
        Возвращает вычисленный НОД.
    """
    if a < b:
        a, b = b, a

    while b != 0:
        a, b = b, a % b

    return a
```

И проведем тестирование:

```
test_nod(get_fast_nod)
```

Получим результаты:

```
#test1 - ok
#test2 - ok
#test3 – ok
```

Все три теста прошли успешно.

Вот пример того, как можно оформлять функции на языке Python, добавляя в них описание, и проводя тестирование.

7.5. Позиционные и именованные аргументы. Параметры со значениями

На данный момент мы с вами уже знаем, что любая пользовательская функция может содержать произвольный набор параметров, в которые передаются значения при ее вызове. Например, простая функция вычисления объема прямоугольного параллелепипеда, может быть записана с тремя параметрами (ширина, высота и глубина) следующим образом:

```
def get_V(a, b, c):
    print(f"a = {a}, b = {b}, c = {c}")
    return a * b * c
```

И вызвана с конкретными числовыми значениями (аргументами):

```
v = get_V(1, 2, 3)
print('v = ', v)
```

В консоли появятся строчки:

```
a = 1, b = 2, c = 3
v =  6
```

Видим, что параметру `a` соответствует число 1, параметру `b` – число 2, а параметру `c` – число 3. То есть, значения параметров a, b, c определяются порядком записи аргументов. Такие аргументы получили название **позиционных**. А можно ли, не меняя порядка записи, параметру `b` присвоить 1, параметру `c` – 2, а параметру `a` – 3? Оказывается да, в языке Python такое возможно, если явно указывать имена параметров при вызове функции:

```
v = get_V(b=1, a=2, c=3)
```

Такие аргументы называются **именованными**. Теперь, при запуске программы, видим следующий результат вывода:

```
a = 2, b = 1, c = 3
```

А можем ли мы комбинировать позиционные и именованные аргументы? Да и такое тоже возможно. Только вначале следует указывать позиционные аргументы, а в конце именованные, например, так:

```
v = get_V(1, c=2, b=3)
```

Получим результат:

```
a = 1, b = 3, c = 2
```

Если нарушить это правило и позиционные аргументы прописать после именованного:

```
v = get_V(a=1, 2, 3)   # SyntaxError
```

то возникнет синтаксическая ошибка. Сначала всегда следует записывать позиционные аргументы и только потом именованные:

```
v = get_V(1, 2, c=3)
```

Причем, обратите внимание, последним именованным аргументом здесь может быть только имя параметра c. Если записать любой другой из трех, например b:

```
v = get_V(1, 2, b=3)   # TypeError
```

то у нас получится дублирование передаваемых данных. Второй позиционный аргумент уже присваивается параметру b, а далее, мы снова этому же параметру присваиваем значение 3. Так делать нельзя.

7.5.1. Параметры со значениями по умолчанию

Вернемся теперь к параметрам самой функции. До сих пор мы их объявили просто через запятую с именами, например, a, b, c. Однако можно задавать параметры, прописывая у них значения по умолчанию, например, так:

```
def get_V(a, b, c, verbose=True):
    if verbose:
        print(f"a = {a}, b = {b}, c = {c}")

    return a * b * c
```

В чем отличие обычных параметров от параметров со значениями по умолчанию? Главным образом в том, что их не обязательно указывать при вызове функции. Например, прежний вызов функции:

```
v = get_V(1, 2, 3)
```

сработает без каких-либо проблем. Мы не указали аргумент для последнего параметра verbose, но это и не обязательно, т.к. по умолчанию (если не передано какое-либо другое значение) он будет равен True.

Если же вызов записать так:

```
v = get_V(1, 2, 3, False)
```

то функция print внутри get_V вызвана не будет, так как значение параметра verbose в этом случае равно False.

Также вызов можно записать, используя соответствующе имя параметра, например:

```
v = get_V(1, 2, 3, verbose=False)
```

Результат работы этой функции будет тем же.

Зачем вообще нужны параметры с начальными значениями и когда их следует использовать? Я, думаю, ответ здесь очевиден – для удобства вызова функций. Как мы только что видели, указывать аргументы для параметров с начальными значениями не обязательно, если нас устраивает поведение функции по умолчанию. В других, как предполагается, редких случаях, всегда можно поменять значение такого параметра на другое и скорректировать работу

функции.

Чтобы все это было понятнее, приведу следующий простой пример. Допустим, мы собираемся сравнивать строки в разных режимах: с учетом и без учета регистра, а также учитывать или не учитывать пробелы вначале и в конце строк. Для этого объявим функцию compare_str следующим образом:

```
def compare_str(s1, s2, reg=False, trim=True):
    if reg:
        s1 = s1.lower()
        s2 = s2.lower()
    if trim:
        s1 = s1.strip()
        s2 = s2.strip()

    return s1 == s2
```

Предполагается, что параметры reg и trim определяют наиболее частый вариант использования операции сравнения строк: с учетом регистра и с удалением пробелов. Поэтому чаще всего функция будет вызываться с двумя аргументами, например, так:

```
res = compare_str("Python ", "  Python")
print(res)    # True
```

Если же, в редкой ситуации, нужно поменять ее поведение, указываются другие значения параметров reg и trim. Например:

```
r1 = compare_str("Python ", "  Python", trim=False)
r2 = compare_str("Python", "PYTHON", True, False)
```

В последнем вызове используются обычные позиционные аргументы. В этом случае параметр reg примет значение True, а параметр trim значение False. Конечно, всегда можно воспользоваться и именованными аргументами, здесь полная свобода выбора:

```
print(compare_str("Python", "PYTHON", reg=True, trim=False))
```

Однако в Python в качестве начальных значений параметров следует использовать неизменяемые типы, например, числа, строки, булевы значения, кортежи. А вот изменяемые лучше не прописывать. Почему так? Давайте посмотрим на следующий простой пример, в котором определена функция add_value с параметром lst, ссылающийся по умолчанию на пустой список (список – это изменяемый тип данных):

```
def add_value(value, lst=[]):
    lst.append(value)
    return lst
```

Конечно, никто не запрещает этого делать, но давайте посмотрим, к чему это приведет. Вызовем эту функцию дважды:

```
l = add_value(1)
l = add_value(2)
```

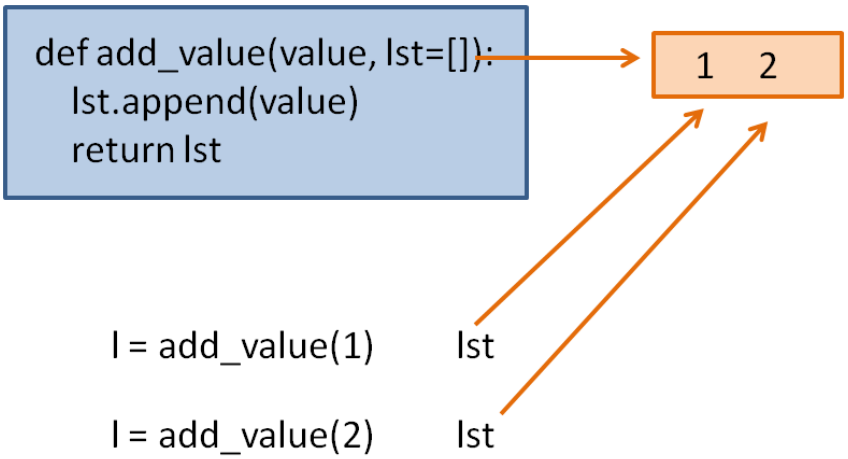
И выведем на экран результирующий список:

```
print(l)
```

Видим два значения в списке:

```
[1, 2]
```

Объявляя таким образом функцию, мы ожидали, что при каждом ее вызове параметр lst будет ссылаться на пустой список и функция будет каждый раз возвращать один элемент в этом списке. Но, получилось, что она сохраняет прежнее состояние списка при повторном вызове. Почему так произошло? На самом деле, все просто. Когда объявляется какая-либо функция, то создается объект этой функции и пустой список для параметра lst. Затем, при вызове этой функции параметр lst ссылается на этот ранее созданный список и добавление нового элемента происходит в него. При повторном вызове функции параметр lst продолжает ссылаться на этот же список и в него добавляется следующее значение. То есть, при вызовах функции список не инициализируется повторно, а используется один и тот же объект. Список создается один раз в момент создания объекта-функции.



Конечно, можно исправить эту ситуацию, передавая при каждом вызове пустой список:

```
l = add_value(1, [])
l = add_value(2, [])
```

И тогда параметр lst будет ссылаться на переданный пустой список. А как сделать так, чтобы такое поведение было по умолчанию, чтобы нам явно не приходилось передавать пустой список? Обычно это делается следующим образом. Параметр lst инициализируется неизменным значением, например, None:

```
def add_value(value, lst=None):
    if lst is None:
        lst = []
```

```
    lst.append(value)
    return lst
```

А в самой функции проверяется, если параметр lst равен None, то есть, функция вызвана с одним первым аргументом, то создается новый пустой список и в него помещается указанное значение. В этом случае, при каждом вызове функции:

```
l = add_value(1)
l = add_value(2)
```

будет автоматически создаваться новый список. То, что мы и хотели. А если требуется добавлять элементы в предыдущий список, то достаточно указать его вторым аргументом:

```
l = add_value(2, l)
```

Этот нюанс при определении параметров функции со значениями по умолчанию следует всегда учитывать.

7.6. Функции с произвольным числом аргументов. Коллекции args и kwargs

Функции в языке Python можно объявлять так, чтобы они при вызове принимали произвольное число аргументов. Например, стандартная функция max для выделения максимального значения:

```
res1 = max(1, 2, 3, -4)
res2 = max(1, 2)
res3 = max(1, 2, 10, 20, 30, 40, 100, 0)
```

принимает произвольное количество аргументов. Давайте посмотрим, как в программе можно определять такие функции.

В качестве примера создадим функцию, которая формирует маршрут к файлу из его отдельных фрагментов (частей), количество которых может быть произвольным:

F:\~stepik.org\Добрый, добрый Python (Питон)\39\p39. Функции.docx

Пусть функция называется os_path, а вместо списка параметров пропишем «*args» следующим образом:

```
def os_path(*args):
    print(args)
```

Для начала, посмотрим, как это будет работать. Вызовем функцию с тремя строковыми аргументами:

```
os_path("F:\\~stepik.org", "Добрый, добрый Python (Питон)", "39\\p39.
Функции.docx")
```

Не забываем здесь про экранирование обратных слешей. После запуска программы в консоли увидим, что переменная args ссылается на кортеж со

значениями переданных трех аргументов:

```
('F:\\~stepik.org', 'Добрый, добрый Python (Питон)', '39\\p39.
Функции.docx')
```

То есть, чтобы функция принимала произвольное число аргументов, в ее объявлении в параметрах достаточно прописать оператор «*» и указать имя кортежа, в который будут помещены все значения аргументов. По сути, оператор «*» выполняет упаковку переданных данных в кортеж.

Перепишем теперь функцию os_path, чтобы она формировала полный путь и возвращала его. Сделать это можно с помощью ранее рассмотренного метода строк join следующим образом:

```
def os_path(*args):
    path = "\\".join(args)
    return path
```

Вызовем эту функцию:

```
p = os_path("F:\\~stepik.org",
            "Добрый, добрый Python (Питон)",
            "39\\p39. Функции.docx")
print(p)
```

Увидим на экране следующий сформированный маршрут:

```
F:\~stepik.org\Добрый, добрый Python (Питон)\39\p39. Функции.docx
```

Вот так, достаточно просто, объявляются функции, которые способны принимать произвольное количество аргументов.

Однако параметр функции «*args» способен принимать только позиционные аргументы. При попытке вызвать эту же функцию с любым именованным аргументом, например, так:

```
p = os_path("F:\\~stepik.org",
            "Добрый, добрый Python (Питон)",
            "39\\p39. Функции.docx",
            sep='/'   # именованный аргумент
            )
```

получим ошибку:

```
TypeError: os_path() got an unexpected keyword argument 'sep'
```

Как это можно поправить? Здесь есть, по крайней мере, два способа. В самом простом варианте достаточно прописать этот параметр при объявлении функции, например, так:

```
def os_path(*args, sep='\\'):
    path = sep.join(args)
    return path
```

Теперь никаких проблем с вызовом нет. Но, конечно, указать, какой-либо другой именованный аргумент не получится. Следующий вызов:

```
p = os_path("F:\\~stepik.org",
            "Добрый, добрый Python (Питон)",
            "39\\p39. Функции.docx",
            sep='/', trim=True
            )
```

приведет к аналогичной ошибке:

```
TypeError: os_path() got an unexpected keyword argument 'trim'
```

Спрашивается, как можно объявить функцию, которая бы принимала произвольное количество самых разных именованных аргументов? Делается это с помощью следующего синтаксиса:

```
def os_path(*args, **kwargs):
    print(kwargs)
    path = kwargs['sep'].join(args)
    return path
```

Обозначение «**kwargs» означает словарь, в который будут упаковываться переданные именованные аргументы. Ключами здесь являются имена, а значениями – соответствующие значения именованных аргументов. После выполнения этой программы, увидим следующее содержимое коллекции **kwargs:

```
{'sep': '/', 'trim': True}
```

Обратите внимание, что при объявлении функции os_path коллекция «**kwargs» обязательно должна следовать после коллекции «*args», наоборот записывать нельзя, так как вначале должны идти позиционные аргументы и только потом именованные.

Если при объявлении функции нужно явно прописывать какие-либо параметры с начальными значениями, то они указываются до коллекции «**kwargs», например, так:

```
def os_path(*args, sep='\\', **kwargs):
    path = sep.join(args)
    return path
```

Явное обозначение параметров гарантирует их существование внутри тела функции. Поэтому делать проверку на существование переменной sep уже не нужно. А вот другие передаваемые именованные аргументы следует проверять, прежде чем использовать. Например, для параметра trim необходимо делать проверку его существование в словаре kwargs, например, так:

```
def os_path(*args, sep='\\', **kwargs):
    if 'trim' in kwargs and kwargs['trim']:
        args = [x.strip() for x in args]
```

```
    path = sep.join(args)
    return path
```

Если параметр trim был передан со значением, которое интерпретируется, как True, то удаляются пробелы до и после у всех фрагментов путей к файлу.

По аналогии можно прописывать обычные параметры до коллекции «*args». Например, так:

```
def os_path(disk, *args, sep='\\', **kwargs):
    args = (disk,) + args

    if 'trim' in kwargs and kwargs['trim']:
        args = [x.strip() for x in args]

    path = sep.join(args)
    return path
```

Здесь первый параметр disk будет всегда ссылаться на первый позиционный аргумент, а остальные позиционные аргументы помещаются в коллекцию args:

```
p = os_path("F:", "~stepik.org",
            "Добрый, добрый Python (Питон)",
            "39\\p39. Функции.docx",
            sep='/', trim=True
            )
```

Соответственно, можно прописывать произвольное количество обычных параметров до «*args» и произвольное количество параметров с начальными значениями до коллекции «**kwargs». Вот общий принцип, по которому объявляются функции, принимающие произвольное количество позиционных и именованных аргументов.

7.7. Операции упаковки и распаковки коллекций

При рассмотрении функций, принимающих произвольное число аргументов, мы с вами столкнулись с двумя операциями «*» и «**», которые упаковывали данные в кортеж и словарь соответственно. Однако эти операции можно использовать не только в объявлении функций, но и при работе с разными коллекциями. Например, мы с вами уже видели, что любой кортеж из двух значений можно распаковать в две переменные следующим образом:

```
x, y = (1, 2)
```

Однако если прописать более двух значений, например, четыре:

```
x, y = (1, 2, 3, 4)
```

то получим ошибку:

```
ValueError: too many values to unpack (expected 2)
```

так как элементов четыре, а переменных всего две. Но, используя операцию «*», можно упаковать оставшиеся значения во вторую переменную, например, так:

```
x, *y = (1, 2, 3, 4)
```

В результате значения переменных будут следующими:

```
x = 1, y = [2, 3, 4]
```

Обратите внимание, что при упаковке значения помещаются в список, а не в кортеж.

Можно записать и наоборот, упаковать первые три значения в x, а четвертое значение поместить в y:

```
*x, y = (1, 2, 3, 4)
```

Тогда:

```
x = [1, 2, 3], y = 4
```

То же самое можно проделывать и со списками:

```
x, *y = [1, "a", True, 4]    # x = 1, y = ["a", True, 4]
```

и строками:

```
*x, y, z = "Hello Python!"
```

В этом случае получим результат:

```
x=['H', 'e', 'l', 'l', 'o', ' ', 'P', 'y', 't', 'h', 'o'], y='n', z='!'
```

И так можно делать с любыми итерируемыми объектами языка Python. То есть, операция «*» упаковывает оставшиеся значения в список. Но есть одно важное ограничение. Мы не можем упаковывать уже упакованные данные, например, следующая команда:

```
*y = 1, 2, 3
```

приведет к ошибке SyntaxError. Но такая будет выполнена без ошибок:

```
x, *y = 1, 2, 3
```

Это следует иметь в виду, при использовании операции упаковки «*».

Однако эта же операция «*» может выполнять и обратное действие – распаковывать коллекции в набор данных. Пусть имеется список:

```
a = [1, 2, 3]
```

Мы уже знаем, что его можно преобразовать в кортеж командой:

```
t = tuple(a)  # t = (1, 2, 3)
```

Но то же самое можно сделать путем распаковки элементов списка

непосредственно в кортеж. Для этого достаточно прописать команду:

```
t = (*a,)
```

Получим тот же результат.

Этот прием удобно использовать при вызове функций. Допустим, имеется кортеж из двух значений:

```
d = -5, 5
```

и затем требуется вызвать функцию range с этими значениями, то есть:

```
r = range(-5, 5)
```

Если передать непосредственно кортеж d:

```
r = range(d)
```

то возникнет ошибка, так как функция ожидает числа в качестве аргументов, а не кортеж. А вот если распаковать d:

```
r = range(*d)
```

то никаких ошибок не будет. Получим эквивалент вызова:

```
r = range(-5, 5)
```

Давайте посмотрим, что вернет эта функция. Преобразуем все в список:

```
lst = list(range(*d))
```

Видим значения в диапазоне от -5 до 5 (исключая правую границу):

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

Или то это же преобразование генератора range в список можно записать и так:

```
lst = [*range(*d)]
```

То есть, здесь первая операция «*» распаковывает (перебирает) элементы объекта range в список, а вторая операция «*» распаковывает кортеж d в аргументы функции range.

Этот прием можно использовать для объединения разных коллекций в одну, например, список:

```
a = [1, 2, 3]
total = [*range(*d), *(True, False), *a]
```

Получим результат:

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, True, False, 1, 2, 3]
```

Все те же самые действия можно делать с любыми коллекциями, в том числе и со

словарями. Пусть задан словарь с расшифровкой оценок:

```
d = {0: "безнадежно", 1: "убого", 2: "неуд.",
     3: "удовл.", 4: "хорошо", 5: "отлично"}
```

Тогда при его распаковке командой:

```
s = {*d}
```

получим множество, состоящее из ключей этого словаря:

```
{0, 1, 2, 3, 4, 5}
```

Почему здесь фигурируют именно ключи? Мы уже знаем, что по умолчанию словарь представляет собой итерируемый объект, в котором перебираются ключи. А операция «*», как раз, перебирает словарь как обычный итерируемый объект. В результате, внутри фигурных скобок получается набор ключей из словаря d, то есть множество.

По аналогии можно сформировать список из ключей словаря d:

```
lst_keys = [*d]
```

Если вместо ключей нужно перебрать значения, то следует использовать метод values следующим образом:

lst = [*d.values()]

Получаем список из значений:

```
['безнадежно', 'убого', 'неуд.', 'удовл.', 'хорошо', 'отлично']
```

Соответственно, конструкция:

```
lst = [*d.items()]
```

сформирует список из кортежей пар «ключ-значение»:

```
[(0, 'безнадежно'), (1, 'убого'), (2, 'неуд.'), (3, 'удовл.'), (4,
'хорошо'), (5, 'отлично')]
```

Если же требуется распаковать словарь как словарь, то перед ним следует прописать две звездочки:

```
res = {**d}
```

Теперь вместо множества получаем копию словаря d:

```
{0: 'безнадежно', 1: 'убого', 2: 'неуд.', 3: 'удовл.',
4: 'хорошо', 5: 'отлично'}
```

Используя последнюю конструкцию, можно выполнять объединение нескольких словарей в один. Например, объявим еще один словарь:

```
d2 = {6: "превосходно", 7: "элитарно", 8: "божественно"}
```

Тогда команда:

```
res = {**d, **d2}
```

объединит их следующим образом:

```
{0: 'безнадежно', 1: 'убого', 2: 'неуд.', 3: 'удовл.', 4: 'хорошо', 5:
'отлично', 6: 'превосходно', 7: 'элитарно', 8: 'божественно'}
```

Сами же словари d и d2 остаются без изменений. Однако, начиная с версии Python 3.9, объединение словарей выполняется с помощью операции «|»:

```
res = d | d2
```

Получим тот же результат. Сейчас в программах, в основном, используют эту команду объединения.

Следует отметить, что для упаковки операция «**» не используется. Если записать команду вида:

```
a, **b = d
```

то получим синтаксическую ошибку. В такой конструкции можно указывать только операцию «*».

7.8. Символ звездочка в параметрах функций

Ранее мы с вами научились объявлять функции, которым можно передавать произвольное количество позиционных и/или именованных аргументов. Для этого в параметрах прописываются коллекции вида:

- *args – для хранения значений позиционных аргументов;
- **kwargs – для хранения пар ключ-значение именованных аргументов.

Давайте, для примера, объявим функцию box_nd, которая вычисляет либо периметр многомерного прямоугольного параллелепипеда, либо его объем, в зависимости от значения параметра perimeter:

```
import math

def box_nd(a, b, c, *args, perimeter=True):
    if perimeter:
        return sum((a, b, c, *args)) * 2 ** (len((a, b, c, *args)) - 1)
    else:
        return math.prod((a, b, c, *args))
```

При вызове этой функции обязательно нужно передать первые три аргумента для параметров a, b, c и, при необходимости, дополнительные, которые помещаются в коллекцию *args. Например:

```
res1 = box_nd(5, 7, 3)
res2 = box_nd(5, 7, 3, 5, 2)
```

```
res3 = box_nd(5, 7, 3, 5, 2, perimeter=False)
```

Обратите внимание, что в последнем вызове значение параметра perimeter мы вынуждены передавать через именованный аргумент, т.к. если прописать только значение False:

```
res3 = box_nd(5, 7, 3, 5, 2, False)
```

то оно будет автоматически помещено в коллекцию *args, т.к. становится позиционным аргументом. И этот момент здесь ключевой. Получается, что в параметры функции, которые записаны после *args, можно передавать значения только с помощью именованных аргументов и никак иначе.

Давайте добавим в функцию box_nd еще один параметр initial и коллекцию **kwargs, которая будет сохранять любые переданные именованные аргументы:

```
def box_nd(a, b, c, *args, perimeter=True, initial, **kwargs):
    if perimeter:
        return sum((a, b, c, *args))*2**(len((a, b, c, *args))-1)+initial
    else:
        return math.prod((a, b, c, *args)) + initial
```

Обратите внимание, здесь специально у параметра initial не указано начальное значение, чтобы показать, что это не приведет к ошибке. Хотя, если бы не было коллекции *args, это стало бы ошибкой. Почему так происходит? Как мы с вами только что выяснили, все параметры функции после *args воспринимаются интерпретатором Python, как именованные и он «знает», что initial – это именованный параметр, у которого просто нет начального значения. В результате мы можем вызывать функцию box_nd, например, следующими способами:

```
res1 = box_nd(5, 7, 3, initial=2)
res2 = box_nd(5, 7, 3, 5, 2, initial=0, verbose=True)
res3 = box_nd(5, 7, 3, 5, 2, initial=-1, perimeter=False)
```

То есть, здесь обязательно нужно прописывать именованный аргумент initial.

Конечно, если значение параметра initial будет задано по умолчанию, например:

```
def box_nd(a, b, c, *args, perimeter=True, initial=0, **kwargs):
    …
```

то при вызове функции его можно не указывать.

А теперь смотрите. Имя args, которое мы постоянно прописывали после звездочки – это некий стандарт, принятый в языке Python. Но, конечно же, никто нам не мешает использовать любое другое, например, оставить первые две буквы ‘ar’ следующим образом:

```
def box_nd(a, b, c, *ar, perimeter=True, initial=0, **kwargs):
    if perimeter:
        return sum((a, b, c, *ar))*2**(len((a, b, c, *ar)) - 1) + initial
    else:
```

```
        return math.prod((a, b, c, *ar)) + initial
```

Ничего кардинально не поменяется, хотя делать так не рекомендуется, чтобы не ухудшать визуальное восприятие программы. Но мы можем пойти еще дальше и вообще убрать символы перед звездочкой и записать все в следующем виде:

```
def box_nd(a, b, c, *, perimeter=True, initial=0, **kwargs):
    if perimeter:
        return sum((a, b, c)) * 2 ** (len((a, b, c)) - 1) + initial
    else:
        return math.prod((a, b, c)) + initial
```

Что же в итоге получилось? Теперь в функции box_nd нет коллекции, которая бы сохраняла переданные позиционные аргументы. Поэтому в теле функции мы не можем с ней никак обращаться. Ее пришлось убрать. И, кроме того, теперь, при вызове этой функции, можно передавать только первые три аргумента в a, b, c и никакие дополнительные. А вот в остальные параметры функции по-прежнему можно передавать только именованные аргументы:

```
res1 = box_nd(5, c=7, b=3, initial=2) # OK
res2 = box_nd(5, 7, 3) # OK
res3 = box_nd(5, 7, 3, False) # ERROR
res4 = box_nd(5, 7, 3, 5, 2, perimeter=False) # ERROR
```

Таким образом, получается, что символ * в параметрах функций разделяет параметры на обычные и именованные. Это значит, что вполне можно задать следующее объявление функции:

```
def func(a, b=2, c=3, *, verbose, key=None): …
```

Здесь в параметры a, b, c можно передавать произвольные аргументы (и позиционные и именованные), а в параметры verbose и key только именованные.

Или же, можно сформировать аналогичное объявление, прописав дополнительно имя args после звездочки:

```
def func(a, b=2, c=3, *args, verbose, key=None): …
```

Здесь по-прежнему будет выполняться разделение параметров на обычные и строго именованные, но дополнительно в функцию func можно передавать произвольное количество позиционных аргументов. Вот такую роль выполняет символ «*» в параметрах функций.

7.9. Символ слэш в параметрах функции

Следующий аналогичный символ, который можно прописывать в параметрах функций, – это слэш (/). Он позволяет отделять позиционные параметры от всех остальных. В результате, имеем два специальных символа для управления передачи аргументов при вызове функций:

- * (звездочка) – отделяет именованные параметры функции от всех остальных;

- / (слэш) – отделяет позиционные параметры функции от всех остальных.

Давайте, для примера, объявим очень простую функцию с именем triangle, которая имеет три обычных параметра a, b, c, и возвращает периметр треугольника:

```
def triangle(a, b, c):
    return a + b + c
```

Мы уже знаем, что эту функцию можно вызвать самыми разными способами, например, так:

```
res1 = triangle(10, 20, 30)
res2 = triangle(b=10, a=20, c=30)
res3 = triangle(10, c=20, b=30)
```

Во всех случаях результат (сумма) будет 60. Но, что если нам нужно запретить вызов функций с передачей именованных аргументов в параметры a, b, c? Тем более что в нашей задаче имена первых трех параметров – это просто длины сторон треугольника, которые не несут какой-либо специальной информации. Чтобы это сделать, в определении функции достаточно поставить слэш после параметров a, b, c:

```
def triangle(a, b, c, /):
    return a + b + c
```

Теперь в a, b, c можно передавать только позиционные аргументы. В результате функцию triangle можно вызвать только так:

```
res1 = triangle(10, 20, 30)
```

Два следующих вызова:

```
res2 = triangle(b=10, a=20, c=30)
res3 = triangle(10, c=20, b=30)
```

приведут к ошибкам, т.к. используют именованные аргументы a, b, c.

Однако, обратите внимание, сами по себе параметры функции могут быть определены и как именованные. Например, так:

```
def triangle(a, b, c=4, /):
    return a + b + c
```

В результате, эту функцию теперь можно вызвать следующими способами:

```
res1 = triangle(10, 20, 30)
res2 = triangle(10, 20)
```

Но по-прежнему нельзя указывать именованные аргументы. Например, следующий вызов приведет к ошибке:

```
res3 = triangle(10, 20, c=30) # Error
```

А вот после символа слэш (/) можно прописывать самые разные параметры и передавать в них как позиционные, так и именованные аргументы. Например:

```
def triangle(a, b, c, /, mul, offset=0):
    return (a + b + c) * mul + offset


res1 = triangle(10, 20, 30, 2)
res2 = triangle(10, 20, 30, offset=1, mul=1)
res3 = triangle(10, 20, 30, 1, 10)
```

То есть, теперь при вызове функции triangle ей нужно передавать первые три позиционных аргумента, а остальные два могут использоваться произвольным образом: и как позиционные, и как именованные.

Давайте теперь посмотрим, как можно комбинировать символ-разделитель слэш со специальными параметрами *args и **kwargs. Предположим, что необходимо вычислять длину ломаной линии, состоящей из двух и более сегментов, длины которых передаются в функцию polyline:

```
def polyline(a, b, /, *args, mul=1, offset=0, **kwargs):
    return (a + b + sum(args)) * mul + offset
```

Обратите внимание на порядок записи параметров. Произвольный набор позиционных аргументов *args должен указываться строго после символа слэш, т.к. накладывать на *args какие-либо ограничения не имеет смысла, потому что в эту коллекцию попадают только позиционные аргументы. А коллекция **kwargs должна прописываться в самом конце.

Теперь можно вызвать эту функцию, например, следующими способами:

```
res1 = polyline(0, 5, 4, -3, 8)
res2 = polyline(0, 5, 4, -3, mul=2)
res3 = polyline(0, 5, offset=1, mul=2)
res4 = polyline(0, 5, offset=1, mul=2, verbose=True)
```

Конечно, последний параметр verbose=True здесь не имеет смысла, он записан просто для примера использования коллекции **kwargs.

7.9.1. Совместное использование символов / и *

Последний пример показывает, что в функциях языка Python можно прописывать оба управляющих символа / и *, например, следующим образом:

```
def rect(width, height, /, offset, *, perimetr=True, **kwargs):
    if perimetr:
        return (width + height) * 2 + offset
    else:
        return width * height + offset
```

В первые два параметра width, height можно передавать только позиционные аргументы, в параметр offset – любые, а в параметр perimetr – только

именованные. Например:

```
res1 = rect(10, 20, 0)
res2 = rect(10, 20, perimetr=False, offset=5)
res3 = rect(10, 20, 10, perimetr=False)
```

Возможно у вас здесь возникает вопрос, зачем вообще нужны такие управляющие символы в параметрах функций? Здесь есть, по крайней мере, две веские причины. Во-первых, когда можно передавать только позиционные аргументы (например, параметры width, height), то это гарантирует, что при вызове функции не были использованы имена width и height, а значит, разработчик в будущем сможет их поменять без какого-либо ущерба в работоспособности остальной части программы. Во-вторых, обязательное применение именованных аргументов для соответствующих параметров функции, позволяет прописывать их в произвольном порядке, как при объявлении функции, так и при ее вызове. Это, опять же, дает некоторую гибкость при разработке и дальнейшего развития программного продукта.

Надо сказать, что в программах на Python нередко встречаются управляющие символы / и *, поэтому важно понимать их назначение и принцип работы.

7.10. Рекурсивные функции

В языке Python, как и во многих других языках программирования, функции могут вызывать сами себя. Это называется рекурсией, а сами функции **рекурсивными**. Поэтому, изучая рекурсивные функции, вы, фактически, знакомитесь с фундаментальным материалом, который применим и в других языках программирования.

На первый взгляд может показаться невероятным, что функция может вызывать саму себя. Но это так. Чтобы все было предельно понятно, вначале разберем работу программы с набором из трех разных функций, которые вызываются друг из друга. Затем все эти функции заменим на одну рекурсивную функцию.

Итак, пусть имеются следующие три функции:

```
def rcs3(x):
    print(f"down: x = {x}")
    print(f"up: x = {x}")


def rcs2(x):
    print(f"down: x = {x}")
    rcs3(x-1)
    print(f"up: x = {x}")


def rcs1(x):
    print(f"down: x = {x}")
    rcs2(x-1)
```

```
    print(f"up: x = {x}")
```

И вызывается функция rcs1 с аргументом x=3:

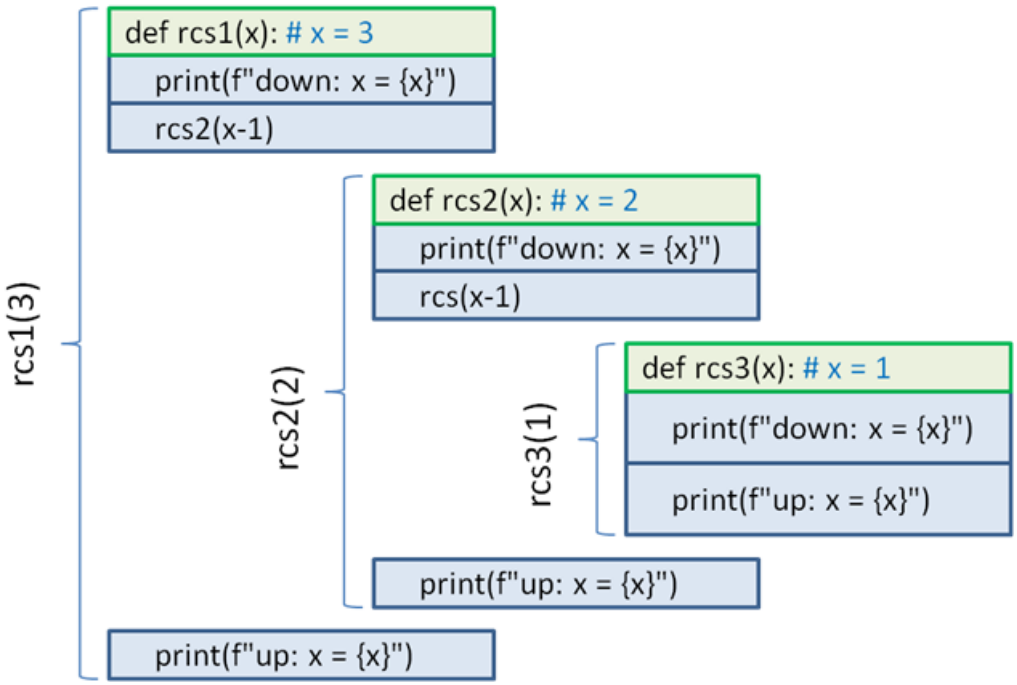
```
rcs1(3)
```

В результате на экране увидим строчки:

```
down: x = 3
down: x = 2
down: x = 1
up: x = 1
up: x = 2
up: x = 3
```

Думаю, вы без особого труда поймете, как работает эта простая программа. Вначале вызывается первая функция rcs1 с параметром x=3. В ней выполняется первая команда print с выводом первой строчки на экран. Затем управление передается следующей функции rcs2, но с параметром x=2. Она, в свою очередь, отображает следующую строчку и вызывает еще одну функцию rcs3 с параметром x=1. Эта последняя функция печатает две строчки:

```
down: x = 1
up: x = 1
```



Что происходит дальше? А дальше управление передается предыдущей функции rcs2, из которой была вызвана функция rcs3. При этом функция rcs2 продолжает свою работу, отрабатывает функция print с выводом строки:

```
up: x = 2
```

После этого она завершается и управление передается функции rcs1, которая так же продолжает выполнять команды. Отрабатывает print с выводом строки:

```
up: x = 3
```

На этом блок из последовательного вызова функций завершает свою работу.

Все это вы уже должны хорошо знать и понимать. А вот следующий шаг будет новым. Мы заменим все эти три функции одной рекурсивной, которая будет выглядеть следующим образом:

```
def rcs(x):
    print(f"down: x = {x}")
    if x > 1:
        rcs(x-1)
    print(f"up: x = {x}")
```

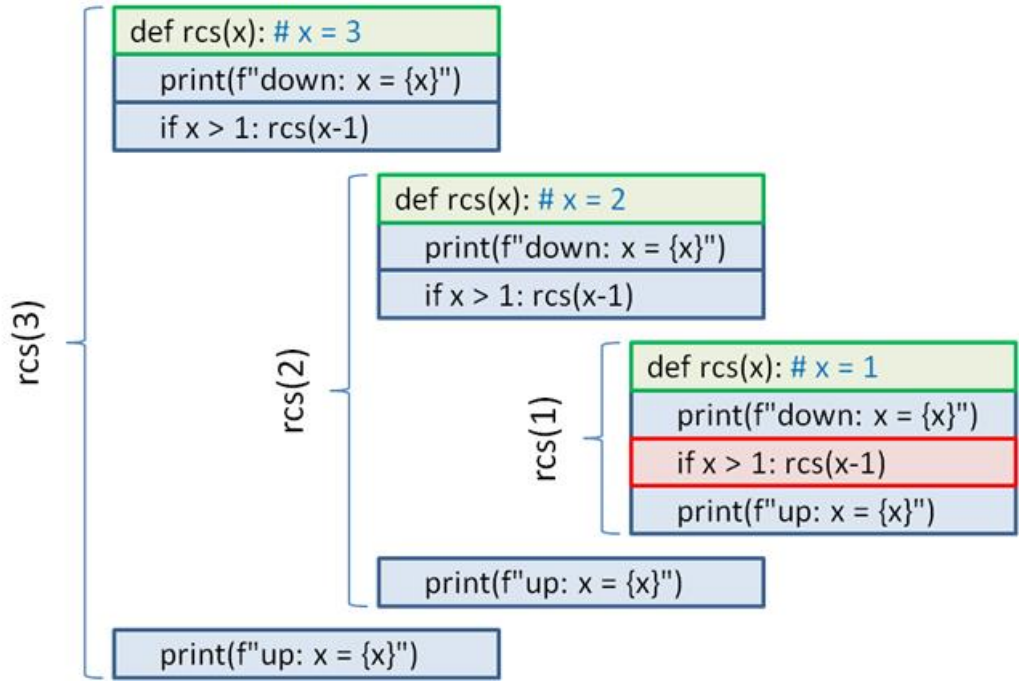
Вызовем эту функцию с начальным значением x=3 и посмотрим, что получится:

```
rcs(3)
```

После запуска программы в консоли появятся те же самые строчки:

```
down: x = 3
down: x = 2
down: x = 1
up: x = 1
up: x = 2
up: x = 3
```

Почему получился такой результат и как это работает? На самом деле все предельно просто. Вся эта «магия» получается благодаря тому, что интерпретатор языка Python запоминает порядок вызова функций, то есть, какая функция из какой и с какими аргументами была вызвана. Поэтому работу нашей рекурсивной функции можно представить в виде последовательных вложенных вызовов, изображенных на рисунке ниже.



Все начинается с первого вызова функции rcs с аргументом x=3. В теле этой функции срабатывает print с выводом в консоль первой строчки:

```
down: x = 3
```

Затем проверяется условие, что x должен быть меньше 1. Так как x=3, то условие срабатывает (истинно) и снова вызывается функция rcs, но уже с аргументом x=2. И вот здесь самый главный момент понимания работы рекурсии. Когда функция вызывает какую-либо другую функцию (в том числе и саму себя), то вызов первой функции не завершается, она продолжает работать, но появляется еще один вызов точно такой же функции, только с аргументом x=2. Для Python – это независимо работающие функции. Параметр x второй функции rcs никак не связан с параметром x первой функции rcs. Это как бы независимые копии одной и той же функции. Это ключевой момент в понимании работы рекурсивных функций: первая функция не завершается, и последующие вызовы этой же функции работают совершенно независимо друг от друга, так, словно это разные функции.

Итак, мы вызвали (рекурсивно) функцию rcs с аргументом x=2. В ней снова отрабатывает print с выводом строки:

```
down: x = 2
```

и проверяется условие, что x должен быть меньше 1. Условие истинно, поэтому еще раз вызывается рекурсивно функция rcs, но с аргументом x=1. Опять отрабатывает print с выводом строки:

```
down: x = 1
```

И теперь условие оказывается ложным, т.к. один не меньше одного. В результате очередного рекурсивного вызова функции rcs не происходит. Выполнение переходит к следующей команде – функции print с выводом строки:

```
up: x = 1
```

На этом работа функции rcs(1) завершается и управление передается предыдущей функции rcs(2). Но в предыдущей функции первые две строчки (команды) уже были выполнены, поэтому выполняется следующая с вызовом функции print, которая выводит на экран строчку:

```
up: x = 2
```

На этом функция rcs(2) также завершает свою работу. Управление переходит к первой функции rcs(1) и здесь выводится строчка:

```
up: x = 3
```

Вот общий принцип работы рекурсивных функций. Как видите, все предельно просто. При этом, когда движение идет вглубь, то имеем **прямой ход** рекурсии, а при возврате – **обратный ход**. Максимальное число вызовов рекурсивной функции называют **глубиной рекурсии**.

А что будет, если глубину ничем не ограничивать? Функция будет вызываться до бесконечности? Давайте проверим. Изменим рекурсивную функцию:

```
def rcs(x):
    print(f"down: x = {x}")
    rcs(x + 1)
```

И вызовем ее с начальным значением x=1:

```
rcs(1)
```

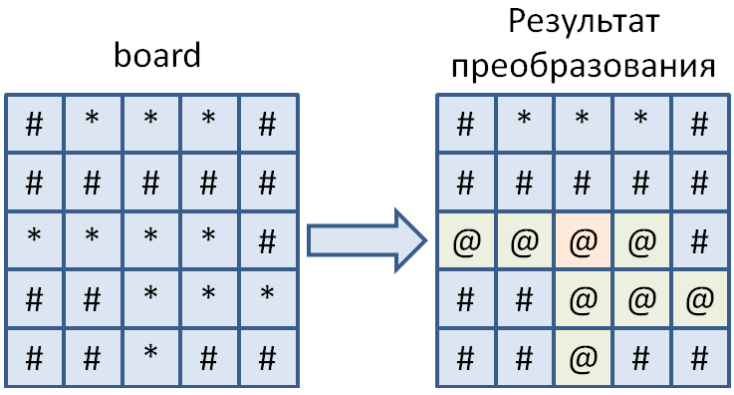
В моем случае функция была вызвана 997 раз, после чего возникла ошибка RecursionError – достигнута максимальная глубина рекурсии:

```
...
down: x = 995
down: x = 996
down: x = 997
RecursionError: maximum recursion depth exceeded while calling a Python
object
```

То есть, максимальная глубина последовательного вызова функций ограничена. В действительности это ограничение есть во всех языках программирования и бесконечно вызывать функции друг из друга недопустимо. Поэтому при описании рекурсивной функции нужно гарантировать ограничение ее глубины вызовов. На практике считается нормальной глубина в 10-20 последовательных вызовов. Максимум следует делать не более 100 вызовов. Если же алгоритм предполагает большие значения глубины рекурсии, то такую программу следует пересмотреть и реализовать другим алгоритмом.

7.10.1. Примеры использования рекурсивных функций

Надеюсь, общий принцип работы рекурсивных функций вам понятен. И, возможно, остается один важный вопрос, зачем они нужны и где имеет смысл их применять? Давайте для этого рассмотрим задачу близкую к реальной.



Предположим, имеется некоторое игровое поле board размером 5x5 клеток и в каждой клетке проставлен либо символ ‘#’, либо символ ‘*’. Наша задача заменить связанную область из звездочек символами ‘@’.

Эту задачу достаточно просто можно реализовать с помощью рекурсивной

функции, например, так:

```
def set_zeros(p, i, j, sz):
    if p[i][j] != '*':
        return

    p[i][j] = '@'

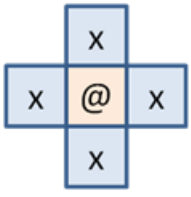
    if i-1 >= 0 and p[i-1][j] == '*':
        set_zeros(p, i-1, j, sz)
    if i+1 < sz and p[i+1][j] == '*':
        set_zeros(p, i+1, j, sz)
    if j-1 >= 0 and p[i][j-1] == '*':
        set_zeros(p, i, j-1, sz)
    if j+1 < sz and p[i][j+1] == '*':
        set_zeros(p, i, j+1, sz)


board = [
    ['#', '*', '*', '*', '#'],
    ['#', '#', '#', '#', '#'],
    ['*', '*', '*', '*', '#'],
    ['#', '#', '*', '*', '*'],
    ['#', '#', '*', '#', '#'],
]

N = len(board)
set_zeros(board, 2, 2, N)

print(*board, sep='\n')
```

Здесь рекурсивная функция set_zeros заменяет символы ‘*’ связанной области на символы ‘@’. Работает она очень просто. Вначале ей передается двумерный список board и индексы начальной клетки. Если текущая клетка списка содержит ‘*’, то она заменяется на символ ‘@’ и запускается рекурсия во все соседние клетки:



Если в соседних клетках также будет символ ‘*’, то они заменятся на символ ‘@’, и рекурсия продолжится дальше. И так, пока вся группа связанных клеток из звездочек не будет заменена символами ‘@’.

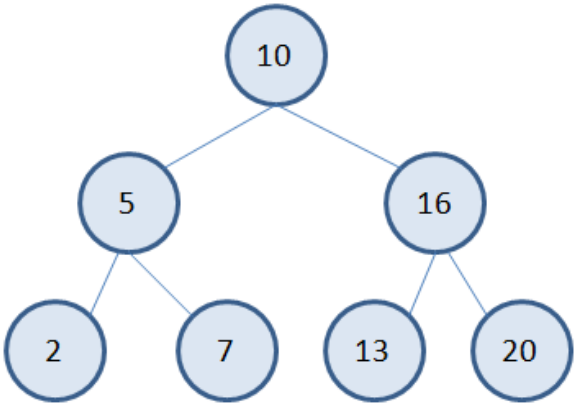
После запуска программы получим следующий результат:

```
['#', '*', '*', '*', '#']
['#', '#', '#', '#', '#']
['@', '@', '@', '@', '#']
```

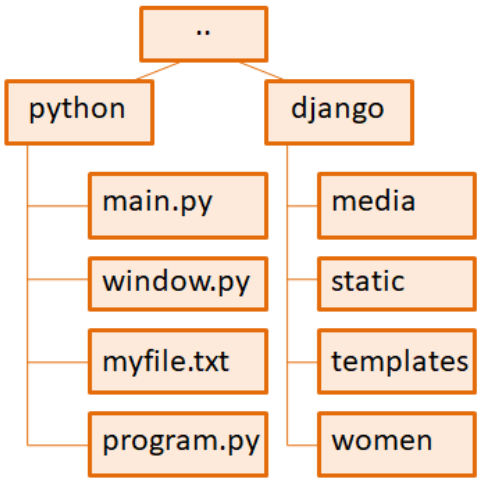
```
['#', '#', '@', '@', '@']
['#', '#', '@', '#', '#']
```

Как видите, все связанные клетки с центральной клеткой (2, 2) были заменены на символ ‘@’.

Вообще рекурсивные функции очень удобны, когда нужно перебирать некие иерархические данные, которые имеют сложную структуру, и обычными циклами их проходить было бы затруднительно. В частности, рекурсия используется для переборов вершин бинарных деревьев – довольно распространенной структуры данных следующего вида:



Или для перебора каталогов и файлов:



И так далее. Везде, где присутствует иерархичность можно подумать об использовании рекурсивных функций. Но нужно всегда помнить и об их недостатках. Они следующие:

- Рекурсивные вызовы работаю медленнее, чем итерации операторов циклов, поэтому, если задачу относительно просто можно реализовать через циклы, то именно их и следует применять.
- Рекурсия не может идти до бесконечности. Программа аварийно завершится, когда стек вызова будет полностью заполнен, поэтому если в задаче сложно оценить максимальную глубину рекурсии, то от использования рекурсивных функций лучше отказаться.

7.10.2. Хвостовая рекурсия

Среди рекурсивных функций выделяют отдельный вид рекурсии под названием **хвостовая**. Ее особенность в том, что в функциях, которые ее реализуют, выполняются следующие условия:

- последним действием функции должен быть рекурсивный вызов;
- результат рекурсивного вызова должен сразу же возвращаться функцией.

Почему акцентрируют внимание на такой реализации? Дело в том, что современные компиляторы и интерпретаторы языков программирования «умеют» хорошо оптимизировать такие рекурсивные вызовы. В частности они делают следующее:

- хвостовая рекурсия не заполняет стек вызова функций, а значит, стек не переполняется (к сожалению, эта оптимизация не поддерживается в Python);
- превращение компилятором хвостовой рекурсии в обычный цикл, который работает быстрее многократного вызова функции (к сожалению, не поддерживается в Python).

Несмотря на то, что хвостовая рекурсия в Python реализуется, как обычная, о ней следует знать, т.к. в других языках программирования это может улучшить качество программного кода.

Давайте рассмотрим классический пример реализации хвостовой рекурсии для вычисления факториала числа. Напомню, математически факториал вычисляется по следующей формуле:

$$n! = 1 \cdot 2 \cdot ... \cdot (n-1) \cdot n$$

Эту же формулу можно записать и в таком виде:

$$n! = (n-1)! \cdot n$$

То есть, факториал натурального числа n можно вычислять рекурсивно, добавляя каждый раз новое значение.

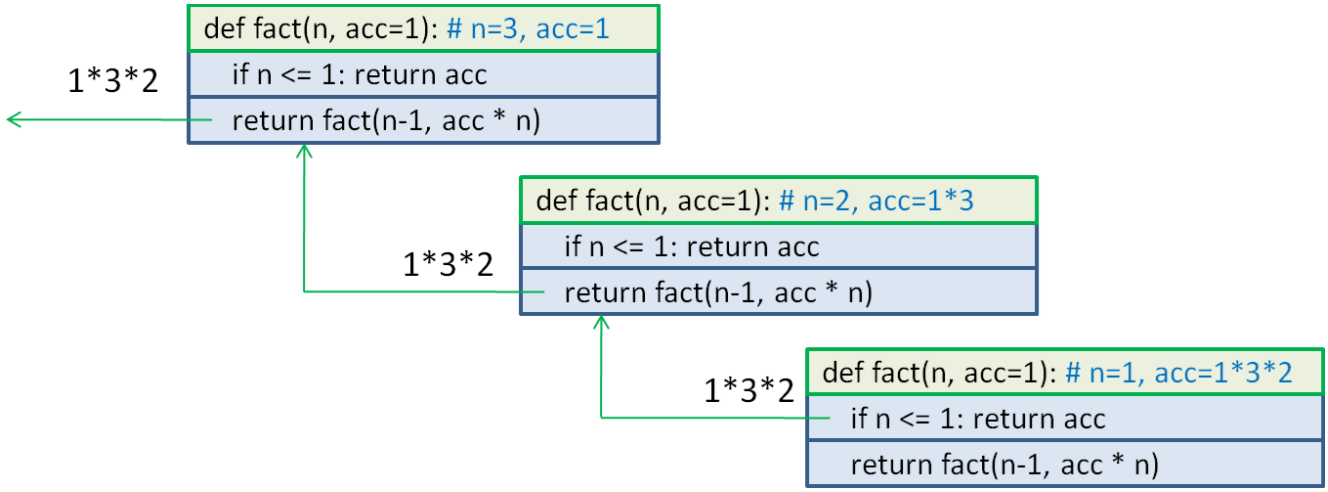
Функцию, которая реализует такую логику, на языке Python можно записать следующим образом:

```
def fact(n, acc=1):
    if n <= 1:
        return acc
    return fact(n - 1, acc * n)
```

Обратите внимание, что рекурсивный вызов функции fact выполняется непосредственно в операторе return. Это вполне допустимо, т.к. в return может быть записано любое вычисляемое выражение. В нашем случае – это вызов функции `fact(n - 1, acc * n)`.

Давайте детально разберем порядок работы функции fact, при ее вызове командой:

```
res = fact(3)
```



Изначально параметры n=3, acc=1 и первое условие не срабатывает. Переходим к оператору return, в котором рекурсивно вызывается функция `fact(n-1, acc*n)`. При этом выполнение текущей функции «замораживается» и выполнение программы переходит к новому вызову рекурсивной функции fact с параметрами n=2, acc=1*3. Напомню, что эти два вызова одной и той же функции fact, совершенно независимы между собой с независимыми параметрами n и acc. Условно можно считать, что это две разные функции, только с одним и тем же именем fact. При новом вызове функции fact условие также не срабатывает и мы оказываемся в операторе return. Здесь снова рекурсивно вызывается функции fact, но уже с аргументами n=1, acc=1*3*2. В этом новом вызове условие остановки рекурсии оказывается истинным, поэтому возвращается вычисленное значение acc и рекурсия завершается. Управление передается предыдущей функции fact, в которой уже выполнены все команды, кроме последнего оператора return. Он просто возвращает вычисленное значение acc=1*3*2 в предыдущую функцию, а текущая завершает свою работу. В самой первой функции fact также срабатывает последний оператор return, который возвращает значение acc=1*3*2. И это будет результат рекурсивного вычисления факториала от трех.

Вот принцип хвостовой рекурсии. То есть, она работает абсолютно так же, как и обычная рекурсивная функция, только рекурсивный вызов записывается непосредственно перед завершением текущей функции в операторе return.

В качестве примера реализации хвостовой рекурсии рассмотрим задачу вычисления суммы элементов арифметической прогрессии, члены которой формируются по правилу:

$$a_0, a_0 + d, a_0 + 2d, ..., a_0 + (n-1)d$$

где a_0 - начальное значение; d - разность арифметической прогрессии; n - общее количество членов арифметической прогрессии.

Для решения этой задачи объявим рекурсивную функцию:

```
def sum_ar(n, a0=0, d=1, acc=0):
    acc += a0
    if n <= 1:
        print(a0)
        return acc

    print(a0 + (n-1)*d, end=", ")
    return sum_ar(n-1, a0, d, acc=acc+(n-1)*d)
```

И вызовем ее следующим образом:

```
res = sum_ar(5, 2, 3)
print(res)
```

После запуска программы увидим:

```
14, 11, 8, 5, 2
40
```

Получается последовательность 14, 11, 8, 5, 2 и сумма ее элементов 40.

Я здесь специально прописал функции print, чтобы было понятно, как формируется последовательность и вычисляется сумма ее элементов. Конечно, в реальной практике, в подобных функциях никаких print не прописывают.

7.11. Анонимные (lambda) функции

Во многих языках программирования, в том числе и в Python, существуют, так называемые, анонимные (lambda) функции. Что это за функции и почему их называют анонимными или лямбда-функциями?

Лямбда функция определяется по достаточно простому синтаксису:

lambda param_1, param_2, …: команда

Например, вот так можно определить анонимную функцию для сложения двух значений:

```
lambda a, b: a + b
```

Здесь a, b – параметры функции. После параметров ставится двоеточие и прописывается одна операция, которая выполняется и автоматически возвращается лямбда-функцией.

Обратите внимание, что у этой функции нет имени, поэтому она и называется анонимной. Но как тогда ее вызывать? Для этого объект-функцию нужно присвоить какой-либо переменной, например, так:

```
s = lambda a, b: a + b
```

И, затем, через нее вызывать саму функцию:

```
result = s(1, 2)      # result = 3
```

В чем особенность такого определения функции? Зачем ее придумали и почему бы не пользоваться обычными функциями? У лямбда-функции есть одно принципиальное отличие от ранее рассматриваемых функций – она может быть записана как элемент любой конструкции языка Python. Например, в виде элемента списка:

```
a = [4, 5, lambda: print("lambda"), 7, 8]
```

Здесь лямбда-функция прописана непосредственно в списке. С обычными функциями так не получится. Они должны быть объявлены заранее и только потом можно было бы сохранить ссылку на нее в списке.

Если теперь обратиться к третьему элементу этого списка:

```
a[2]
```

то увидим ссылку на объект-функцию:

```
<function <lambda> at 0x000001E7426F84A0>
```

Следовательно, ее можно запустить, прописав круглые скобки:

```
a[2]()
```

В результате функция выполняется, и в консоли отображается сообщение:

```
lambda
```

Конечно, анонимные функции обычно не используются для вывода какой-либо информации. Как правило, они выполняют определенную операцию и возвращают полученный результат. Рассмотрим следующий пример. Пусть имеется список, состоящий из целых чисел:

```
lst = [5, 3, 0, -6, 8, 10, 1]
```

Требуется написать обычную функцию, которая бы выбирала значения из этого списка по определенному критерию (фильтру). Сам критерий будет определяться дополнительно с помощью переданной лямбда-функции. Получаем:

```
def get_filter(a, filter=None):
    if filter is None:
        return a

    res = []
    for x in a:
        if filter(x):
            res.append(x)

    return res
```

Здесь второй параметр filter – это ссылка на анонимную функцию, которая будет

отбирать значения из списка, переданного в параметр a. По умолчанию filter=None, что означает отсутствие фильтрации. В этом случае функция get_filter возвращает список без изменений. Если же filter не None, значит, в этот параметр была передана функция, и она используется для отбора итоговых значений.

Если вызвать функция get_filter только с первым аргументом:

```
lst = [5, 3, 0, -6, 8, 10, 1]
r = get_filter(lst)
print(r)
```

То увидим в консоли тот же самый список без изменений:

```
[5, 3, 0, -6, 8, 10, 1]
```

Но, если вторым аргументом передать лямбда-функцию, которая будет возвращать True для четных значений:

```
r = get_filter(lst, lambda p: p % 2 == 0)
```

то сформируется новый список, состоящий только из четных величин:

```
[0, -6, 8, 10]
```

Почему получился такой результат? При вызове функции get_filter второй параметр filter ссылается на лямбда-функцию, возвращающую True для четных значений и False для нечетных. Соответственно, в цикле при переборе элементов списка, условие будет срабатывать только для четных чисел и только они будут добавляться в список res.

Получился очень удобный инструмент отбора элементов списка по разным критериям, заданным в виде анонимной функции. Если бы таких функций не существовало, пришлось бы объявлять отдельную функцию:

```
def even(p):
    return p % 2 == 0
```

А затем, при вызове функции get_filter, передавать ссылку на нее:

```
r = get_filter(lst, even)
```

Согласитесь, это несколько более трудоемкий и менее удобный вариант. Мало того, если в дальнейшем потребуется быстро поменять фильтр, то в случае применения анонимной функции достаточно ее прописать в нужном нам виде. Например, следующий вызов выделит только положительные числа из списка:

```
r = get_filter(lst, lambda p: p > 0)
```

В этом и заключается одно из удобств анонимных функций – их можно сразу прописывать в составе любого оператора языка Python.

Лямбда-функции можно записывать и без параметров, например, так:

```
p = lambda: "hello python"
s = p()
```

В результате переменная s ссылается на строку "hello python", то есть, прописанная строка просто возвращается функцией.

Конечно, у анонимных функций есть свои ограничения. Они могут выполнять только одну указанную операцию языка Python. Также нельзя объявлять анонимные функции в несколько строк. Следующая запись приведет к синтаксической ошибке:

```
lambda a:
    print(a)
```

Наконец, в лямбда-функциях нельзя использовать оператор присваивания:

```
lambda a: a = 1
```

Это также вызовет синтаксическую ошибку, т.к. операторы недопустимо прописывать внутри операций. Это правило касается любых операторов языка Python: их нельзя использовать внутри лямбда-функций. Можно прописывать только операции, использующие входные параметры самой анонимной функции или глобальные переменные. Например:

```
MAX_VALUE = 100
s = lambda a, b: a + b if a+b <= MAX_VALUE else 0

res1 = s(5, 7)    # res1 = 12
res2 = s(100, 1)  # res2 = 0
```

Как видите, тернарная условная операция может быть использована в лямбда-функции, а вот условный оператор if привел бы к синтаксической ошибке.

7.12. Функции-генераторы

Ранее мы с вами рассматривали различные виды генераторов: генераторы списков, словарей, множеств и «чистые» генераторы (выражения-генераторы). Так вот, в Python можно определять еще один вид генераторов в виде функций, о которых сейчас пойдет речь.

Давайте предположим, что нам нужно вычислять средние арифметические для разных последовательностей из целых чисел, например, для таких

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
2, 3, 4, 5, 6, 7, 8, 9, 10
3, 4, 5, 6, 7, 8, 9, 10
4, 5, 6, 7, 8, 9, 10
5, 6, 7, 8, 9, 10
6, 7, 8, 9, 10
7, 8, 9, 10
8, 9, 10
```

```
9, 10
```

Для этого мы могли бы записать следующее выражение-генератор:

```
N = 10
a = (sum(range(i, N+1))/len(range(i, N+1)) for i in range(N))
```

Однако это не очень удобно для восприятия и редактирования и, кроме того, здесь тяжело прописывать сложную логику обработки данных. Поправить это можно двумя способами. В первом, объявить собственную функцию для вычисления среднего арифметического и вызывать ее в генераторе:

```
def avg(start, stop, step=1):
    a = range(start, stop, step)
    return sum(a) / len(a)


N = 10

a = (avg(i, N + 1) for i in range(1, N))
print(list(a))
```

А во втором случае создать собственную **функцию-генератор**, которая бы на выходе выдавала нужные значения. Давайте для начала запишем простую функцию-генератор, а потом вернемся к нашей исходной задаче. Функция будет просто возвращать значения списка:

```
def get_list():
    for x in [1, 2, 3, 4]:
        yield x
```

Смотрите, здесь в цикле записан новый для нас оператор yield, который возвращает текущее значение `x` и «замораживает» состояние функции до следующего обращения к ней (в том числе и все локальные переменные). Именно так определяются функции-генераторы. Оператор yield превращает обычную функцию в функцию-генератор.

Если ее вызвать обычным образом:

```
d = get_list()
print(d)
```

то переменная d будет ссылается на объект-генератор:

```
<generator object get_list at 0x0000021799614940>
```

то есть, вызов функции возвращает выражение-генератор, значения которого можно перебирать с помощью функции next:

```
print(next(d))   # 1
print(next(d))   # 2
```

Также его можно перебирать с помощью цикла for, например:

```
d = get_list()
for i in d:
    print(i, end=" ")
```

Увидим на экране значения:

```
1 2 3 4
```

То есть, сама по себе функция формирует «чистый» генератор, который возвращается при ее вызове:

```
d = get_list()
```

Затем функция next как бы вызывает функцию get_list, которая выполняет одну итерацию цикла и возвращает текущее значение, прописанное в операторе yield. При этом функция не завершает свою работу, а «замораживается» со всеми своими локальными переменными. Повторный вызов next приводит к «разморозке» функции и выполнению следующей итерации цикла с возвратом очередного значения. И так, пока цикл for не завершит свою работу. Так работает оператор yield и функция-генератор get_list.

Вернемся теперь к исходной задаче вычисления средних арифметических значений. Перепишем функцию avg с использованием оператора yield следующим образом:

```
def avg_gen(N, step=1):
    for i in range(1, N):
        a = range(i, N+1, step)
        yield sum(a) / len(a)
```

Принцип здесь тот же самый. На каждой итерации цикла будет возвращаться новое вычисленное среднее арифметическое значение. Воспользоваться этой функцией-генератором можно следующим образом:

```
N = 10

b = avg_gen(N)
print(list(b))
```

После запуска программы увидим список:

```
[5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5]
```

То есть, функция avg_gen полностью заменяет собой выражение-генератор:

```
a = (sum(range(i, N+1))/len(range(i, N+1)) for i in range(N))
```

Но, спрашивается, зачем это все было надо? Почему бы не пользоваться обычными генераторами? Преимущество здесь, главным образом, в удобстве использования. В выражении генератора можно записать лишь одну операцию для формирования значения, а в функции-генераторе – произвольный фрагмент

программы, реализующий нужную нам логику формирования очередного значения. В этом ключевое отличие функции-генератора от обычного генератора.

7.13. Области видимости. Ключевые слова global и nonlocal

В самом начале этой главы мы с вами говорили, что все параметры функции и переменные, объявленные внутри ее тела, существуют только в пределах этой функции и недоступны за ее пределами. Такое разграничение называется областью видимости. Существует глобальная область видимости данных и локальные области. Они могут взаимодействовать между собой. И сейчас мы с вами рассмотрим особенности их работы.

Текст любой программы хранится в определенном файле. Пусть, для определенности, это будет файл с именем:

myprog.py

При запуске программы из этого файла Python автоматически создает пространство имен с названием myprog. Изначально оно пустое, но по мере объявления каких-либо переменных:

```
N = 100
WIDTH, HEIGHT = 1000, 500
```

в нем появляются эти имена (ссылки). Так вот, все переменные, которые находятся непосредственно в этом пространстве имен, являются **глобальными** в пределах текущего программного модуля myprog. То есть, они доступны в любом месте программы внутри файла myprog.py.

Если добавить в эту программу функцию, например, такую:

```
def my_func(lst):
    for x in lst:
        n = x + 1
        print(n)
```

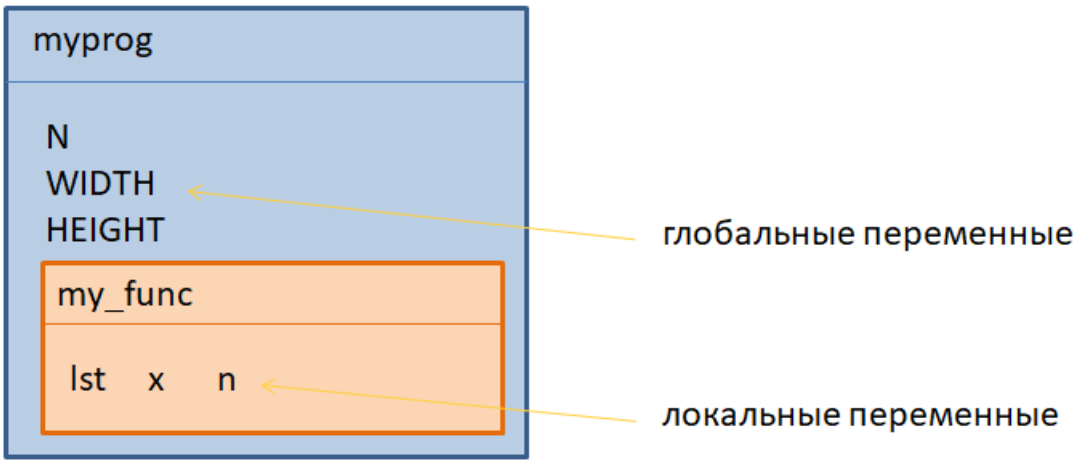
то в пространстве имен появится еще одна переменная my_func – ссылка на объект-функцию, причем эта функция образует свое, локальное пространство имен с названием my_func. Внутри этого локального пространства автоматически формируются переменные lst, x и n. Обратите внимание, цикл for не образовывает своего локального пространства, только функция, и все переменные внутри цикла находятся в области видимости функции.

Как вы уже догадались, переменные внутри локальной области видимости, называются **локальными** и доступны только в ее пределах. Например, вне функции мы не можем обращаться к локальным переменным lst, x и n. Строчка кода за пределами функции my_func:

```
print(x)
```

приведет к ошибке, что x не определена. А вот внутри функции все три

переменные lst, n, x существуют.



Также внутри функции my_func доступны все переменные из внешней области видимости, в данном случае, глобальной. Если внутри функции вывести переменные N и WIDTH:

```
def my_func(lst):
    for x in lst:
        n = x + 1
        print(n)

    print(N, WIDTH)
```

то значения этих переменных будут отображены при вызове этой функции.

А что будет, если внутри my_func определить локальную переменную N, то есть, с тем же именем, что и глобальная переменная? Давайте попробуем:

```
def my_func(lst):
    N = 10
    for x in lst:
        n = x + 1
        print(n)

    print(N, WIDTH)
```

В этом случае будет использоваться именно локальная переменная N, созданная в области видимости функции. То есть, алгоритм поиска переменных здесь такой. Сначала идет обращение к локальной области видимости и если переменная с нужным именем находится, то она и используется. Если же требуемая переменная не существует в локальной области, то поиск переходит на следующий уровень к внешней области и продолжается там. Поэтому, при отсутствии локальной переменной N внутри функции my_func, берется глобальная переменная N из внешней области видимости. При этом значение глобальной переменной никак не меняется, так как мы ее не используем. В этом легко убедиться, прописав в основном блоке программы строчки:

```
my_func([1, 2, 3])
```

```
print(N)   # 100
```

7.13.1. Ключевое слово global

Хорошо, а можем ли мы изменить значение глобальной переменной N внутри тела функции, т.е. внутри локальной области? Если функцию записать в виде:

```
def my_func(lst):
    N += 10          # попытка изменить значение глобальной переменной N
    for x in lst:
        n = x + 1
        print(n)

    print(N, WIDTH)
```

то при ее выполнении возникнет ошибка UnboundLocalError. Она связана с тем, что просто так взять и изменить значение переменной из внешней области видимости нельзя. Прочитать и вывести значение можно, а вот менять запрещено. Если же мы все-таки хотим работать с глобальной переменной N так, словно она объявлена внутри тела функции, то следует воспользоваться ключевым словом global, за которым перечисляются имена глобальных переменных. В нашем примере это будет выглядеть так:

```
def my_func(lst):
    global N         # взять N из глобальной области
    N += 10          # попытка изменить значение глобальной переменной N
    for x in lst:
        n = x + 1
        print(n)

    print(N, WIDTH)
```

Теперь имя N – это имя глобальной переменной и ее изменение внутри функции приведет к изменению и глобального значения. После запуска программы увидим строчки:

```
2
3
4
110 1000
110
```

Однако здесь следует быть аккуратным. Если в функции уже была создана локальная переменная N, то конструкция global приведет к ошибке:

```
def my_func(lst):
    N = 5
    global N         # взять N из глобальной области (SyntaxError)
    N += 10          # попытка изменить значение глобальной переменной N
    for x in lst:
        n = x + 1
        print(n)
```

```
    print(N, WIDTH)
```

После global можно указывать только отсутствующие переменные в локальной области. Конечно, если поменять строчки местами и global записать в самом начале тела функции:

```
def my_func(lst):
    global N          # взять N из глобальной области
    N = 5
    N += 10           # попытка изменить значение глобальной переменной N
    for x in lst:
        n = x + 1
        print(n)

    print(N, WIDTH)
```

то ошибки не будет, т.к. команда `N = 5` уже не создаст локальную переменную, а будет работать с глобальной переменной N.

Интересно, что если в полученной программе глобальную переменную N поставить в комментарий:

```
# N = 100
```

то есть, она изначально не будет существовать, то после выполнения функции my_func, глобальная переменная N будет создана.

7.13.2. Ключевое слово nonlocal

В Python имеется еще один интересный режим работы с локальными переменными с применением ключевого слова nonlocal. Давайте предположим, что имеется объявление одной функции внутри другой (так тоже можно делать):

```
x = 0
def outer():       # внешняя функция
    x = 1
    def inner():   # внутренняя функция
        x = 2
        print("inner:", x)

    inner()        # вызов внутренней функции
    print("outer:", x)

outer()            # вызов внешней функции
print("global:", x)
```

После запуска программы мы увидим следующие ожидаемые результаты:

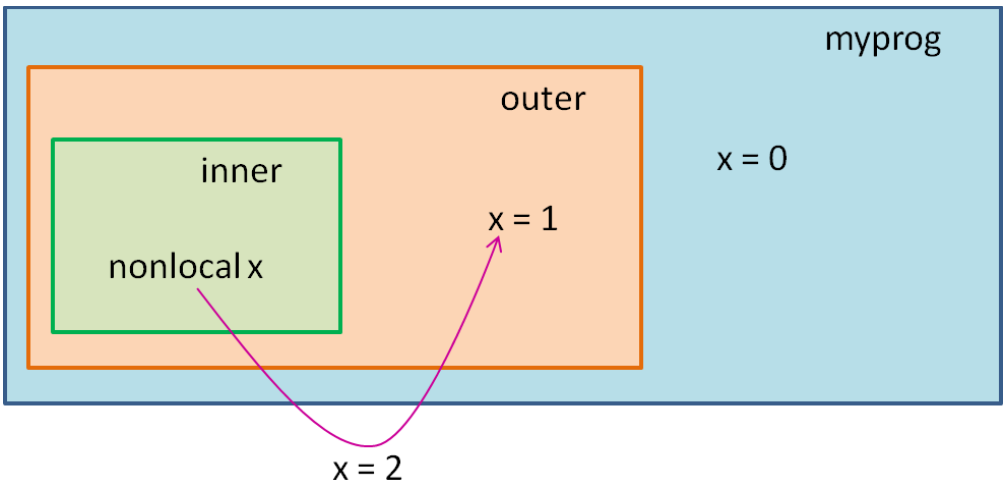
```
inner: 2
outer: 1
global: 0
```

А теперь представим, что внутри функции inner мы хотели бы работать с переменной x, объявленной уровнем выше, то есть, в функции outer. Для этого в теле функции inner прописывается ключевое слово nonlocal и указывается переменная, которую следует взять из внешней области видимости:

```
x = 0
def outer():          # внешняя функция
    x = 1
    def inner():      # внутренняя функция
        nonlocal x    # переменная x берется из функции outer
        x = 2
        print("inner:", x)

    inner()           # вызов внутренней функции
    print("outer:", x)

outer()               # вызов внешней функции
print("global:", x)
```



Теперь команда `x = 2` внутри функции inner будет менять значение переменной x, расположенной в области видимости функции outer. Поэтому после запуска программы получим несколько иные результаты:

```
inner: 2
outer: 2
global: 0
```

Здесь строчка «outer: 2» показывает, что изменение переменной x внутри inner привело к изменению переменной x внутри функции outer.

Однако ключевое слово nonlocal применимо только к внешним локальным переменным. Если внешняя область является глобальной, то ключевое слово nonlocal приведет к ошибке. В этом легко убедиться, если его прописать в функции outer следующим образом:

```
x = 0
def outer():        # внешняя функция
    nonlocal x      # этот nonlocal приведет к ошибке SyntaxError
```

```
    x = 1
    def inner():   # внутренняя функция
        nonlocal x
        x = 2
        print("inner:", x)

    inner()        # вызов внутренней функции
    print("outer:", x)
```

Здесь, вместо nonlocal следует использовать ключевое слово global, а nonlocal в функции inner убрать, иначе опять же получится ссылка на глобальную переменную:

```
x = 0
def outer():       # внешняя функция
    global x
    x = 1
    def inner():   # внутренняя функция
        x = 2
        print("inner:", x)

    inner()        # вызов внутренней функции
    print("outer:", x)

outer()            # вызов внешней функции
print("global:", x)
```

После выполнения программы увидим следующие строчки:

```
inner: 2
outer: 1
global: 1
```

Вот принцип разделения переменных (и вообще, произвольных данных) на глобальные и локальные, а также порядок работы и использования ключевых слов global и nonlocal.

7.14. Замыкания в Python. Вложенные функции

В предыдущем параграфе использовалась конструкция вложенных функций одна в другую. И это не просто интересный прием, а ход с далеко идущими последствиями. В частности, вложенные функции могут образовывать довольно интересную конструкцию, известную в программировании под названием **замыкание**. Давайте в деталях посмотрим, что это такое и что нового она дает.

Пусть имеется функция say_name, содержащая внутреннюю функцию say_goodbye. Причем внутренняя функция вызывается в теле внешней функции:

```
def say_name(name):
    def say_goodbye():
        print("Don't say me goodbye, " + name + "!")
```

```
    say_goodbye()
```

В результате вызова внешней функции командой:

```
say_name("Sergey")
```

В консоли увидим строчку:

```
Don't say me goodbye, Sergey!
```

Я думаю здесь все должно быть понятно. А теперь сделаем, следующее. Вместо вызова внутренней функции возвратим ссылку на нее:

```
def say_name(name):
    def say_goodbye():
        print("Don't say me goodbye, " + name + "!")

    return say_goodbye
```

Конечно, после запуска программы мы не увидим никакого сообщения, так как внутренняя функция нигде не вызывается. Исправим это. Сохраним ссылку на функцию say_goodbye в переменной f:

```
f = say_name("Sergey")
```

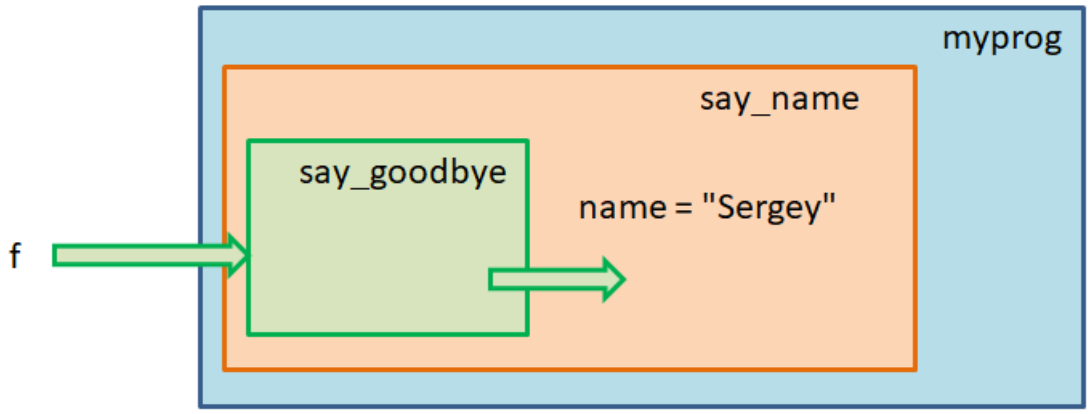
А, затем, вызовем ее:

```
f()
```

Увидим то же самое сообщение:

```
Don't say me goodbye, Sergey!
```

Вам не кажется здесь ничего странным? Например, откуда функция say_goodbye берет значение переменной name? Ведь внешняя функция say_name выполнилась и завершилась, а значит, все ее локальные переменные вроде как тоже должны были бы исчезнуть? Однако мы обращаемся к переменной name и успешно получаем ее значение! Почему? Давайте разберемся.



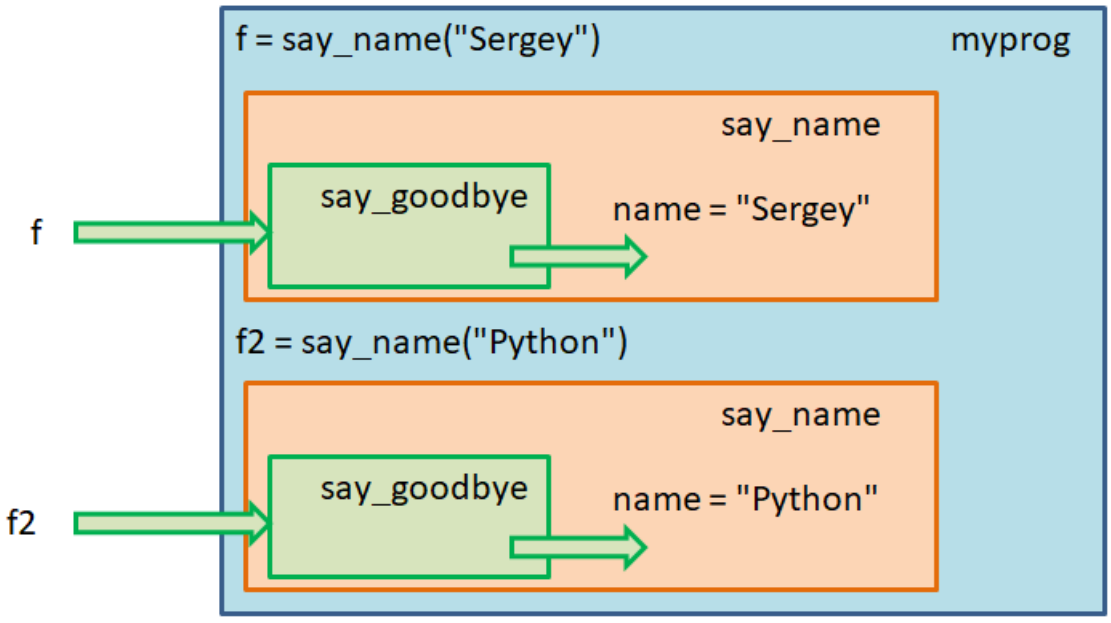
Дело в том, что когда у нас имеется глобальная ссылка f на внутреннее, локальное окружение функции say_goodbye, то это окружение продолжает существовать, оно не удаляется автоматически сборщиком мусора, именно из-за этой

глобальной ссылки на него. А вместе с ним, продолжают существовать и все внешние локальные окружения, в данном случае – окружение функции say_name, потому что также существует неявная, скрытая ссылка на него из внутреннего окружения. Такие ссылки формируются автоматически и позволяют, в частности, обращаться к переменным, объявленным во внешних окружениях. Именно поэтому функция print в say_goodbye имеет доступ к переменной name и эта переменная продолжает существовать, пока существует окружение say_goodbye, а значит и окружение say_name.

Такой эффект «удержания», когда внутреннее локальное окружение может использовать переменные из соответствующих внешних окружений, в программировании называется **замыканием**. Замыкание в том смысле, что мы держим внутреннее окружение say_goodbye переменной f из глобального окружения. Получается цепочка ссылок, замыкающаяся на глобальном окружении. Мало того, при каждом новом вызове внешней функции, формируется свое новое, независимое локальное окружение, со своими локальными переменными и соответствующими значениями. Например, два следующих последовательных вызова:

```
f = say_name("Sergey")
f2 = say_name("Python")
f()
f2()
```

сформируют два независимых объекта-функции со своими внутренними локальными окружениями:



Именно поэтому они отрабатывают независимо друг от друга и на экране появляются две строчки:

```
Don't say me goodbye, Sergey!
Don't say me goodbye, Python!
```

Где может пригодиться такой функционал? Например, можно создать функцию-счетчик, которая бы увеличивала значение локальной переменной на единицу при каждом своем запуске:

```
def counter(start=0):
    def step():
        nonlocal start
        start += 1
        return start

    return step
```

Обратите внимание, здесь использовано ключевое слово nonlocal, чтобы переменная start изменялась во внешней локальной области, а не создавалась бы в текущей. Без этой строчки возникла бы ошибка при выполнении команды `start += 1`.

Используя функцию counter, можно сформировать несколько независимых счетчиков, например, таких:

```
c1 = counter(10)
c2 = counter()
```

И вызвать их:

```
print(c1(), c2())
print(c1(), c2())
print(c1(), c2())
```

На экране увидим:

```
11 1
12 2
13 3
```

Видим, что оба счетчика работают независимо друг от друга.

Еще один пример с замыканиями функций. Предположим, требуется объявить функцию, которая бы удаляла ненужные символы в начале и в конце строки. Через замыкание это можно реализовать следующим образом:

```
def strip_string(strip_chars=" "):
    def do_strip(string):
        return string.strip(strip_chars)

    return do_strip
```

Обратите внимание, вложенная функция тоже имеет параметр – строку, из которой будут удаляться ненужные символы. Далее, создадим два объекта с разным набором удаляемых символов:

```
strip1 = strip_string()
```

```
strip2 = strip_string(" !?,.;")
```

Вызовем полученные функции, передавая им в качестве аргумента обрабатываемую строку:

```
print(strip1(" hello python!.. "))
print(strip2(" hello python!.. "))
```

Увидим результат:

```
hello python!..
hello python
```

Первая функция strip1 убрала только пробелы, а вторая strip2 еще и восклицательный знак с точками. С помощью таких функций в программе можно обрабатывать произвольные строки с удалением указанного набора символов. Причем этот набор был задан только один раз при вызове внешней функции strip_string, а затем он применяется при вызове внутренней функции do_strip. Получаем гибкий инструмент формирования нужного объекта-функции с дальнейшим многократным его использованием.

7.15. Декораторы функций

7.15.1. Декораторы функций без параметров

Наверное, самое широкое применение замыканий – это реализация декораторов для произвольных функций. Что такое декоратор и причем здесь замыкания? Смотрите, можно записать вложенные функции func_decorator и wrapper с передачей им ссылки func на некоторую стороннюю внешнюю функцию:

```
def func_decorator(func):
    def wrapper():
        print("------ что-то делаем перед вызовом функции ------")
        func()
        print("------ что-то делаем после вызова функции ------")

    return wrapper
```

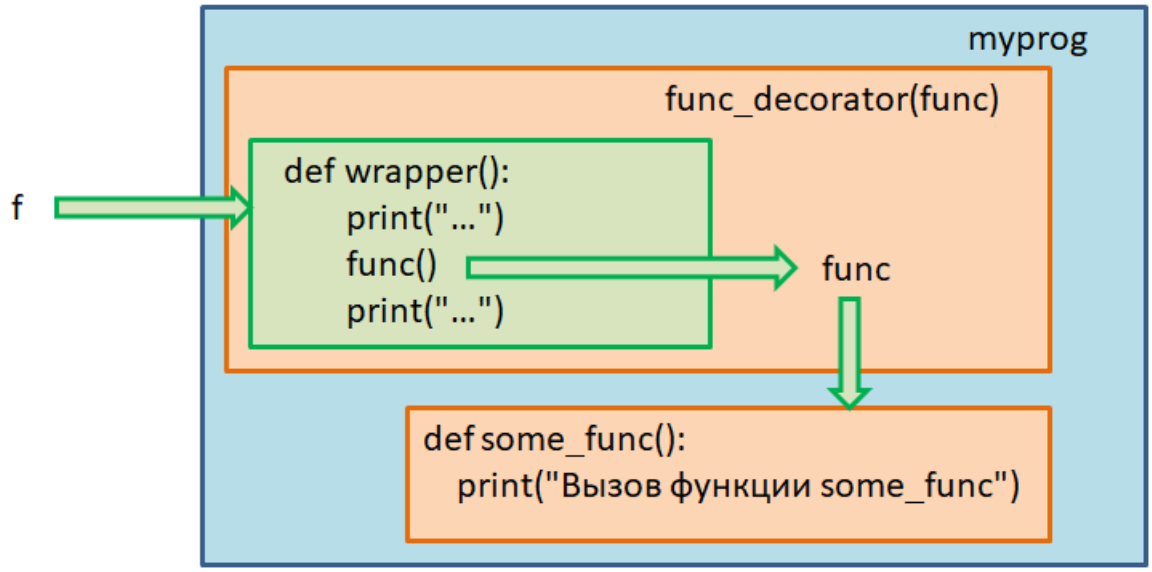
Затем во вложенной функции wrapper вызвать переданную функцию func, где дополнительно можно выполнить некоторые действия, изменяя, таким образом, логику работы исходной функции func. В этом и состоит смысл декораторов – они меняют поведение исходной функции, которая передается им через параметр func.

Для примера, объявим некоторую внешнюю функцию:

```
def some_func():
    print("Вызов функции some_func")
```

И передадим ссылку на нее декоратору func_decorator:

```
f = func_decorator(some_func)
```



В результате, переменная f будет ссылаться на вложенную функцию wrapper и при ее вызове:

```
f()
```

сначала запустится wrapper, а она, в свою очередь, запустит переданную функцию func, то есть, some_func. На экране увидим строчки:

```
------ что-то делаем перед вызовом функции ------
Вызов функции some_func
------ что-то делаем после вызова функции ------
```

То есть, декоратор – это функция, которая принимает ссылку на другую внешнюю функцию и расширяет ее функциональность за счет вложенной функции.

Часто, чтобы не создавать новые имена функций, вместо имени f используют то же самое имя функции, например:

```
some_func = func_decorator(some_func)
some_func()
```

В результате получается, словно, была модифицирована существующая функция. В этом и заключается смысл декоратора – наполнить уже существующую функцию дополнительным функционалом.

Однако если в функцию some_func добавить хотя бы один параметр:

```
def some_func(title):
    print(f"title = {title}")
```

А, затем, вызвать ее с одним аргументом:

```
some_func = func_decorator(some_func)
some_func("Python навсегда!")
```

то увидим ошибку:

```
TypeError: func_decorator.<locals>.wrapper() takes 0 positional
```

```
arguments but 1 was given
```

так как внутренняя функция wrapper прописана без параметров. А именно на нее и ссылается теперь some_func. В самом простом варианте можно просто добавить этот параметр в wrapper:

```
def func_decorator(func):
    def wrapper(title):
        print("------ что-то делаем перед вызовом функции ------")
        func(title)
        print("------ что-то делаем после вызова функции ------")

    return wrapper
```

И теперь программа будет запускаться без ошибок:

```
------ что-то делаем перед вызовом функции ------
title = Python навсегда!
------ что-то делаем после вызова функции ------
```

Но, очевидно, она перестанет работать, если число параметров декорируемой функции some_func изменится. Как тогда записать универсальную реализацию декоратора для функций с разным числом параметров? Конечно, для этого нужно определить вложенную функцию, принимающую произвольное число позиционных и именованных аргументов следующим образом:

```
def func_decorator(func):
    def wrapper(*args, **kwargs):
        print("------ что-то делаем перед вызовом функции ------")
        func(*args, **kwargs)
        print("------ что-то делаем после вызова функции ------")

    return wrapper
```

Затем коллекции *args и **kwargs распаковываются при вызове функции func. Получаем универсальную реализацию декоратора. И, действительно, если теперь добавить еще один параметр в функцию some_func:

```
def some_func(title, tag):
    print(f"title = {title}, tag = {tag}")
```

То достаточно будет при вызове декоратора передать эти два аргумента:

```
some_func = func_decorator(some_func)
some_func("Python навсегда!", 'h1')
```

И, наконец, последний штрих. Сама по себе функция some_func может возвращать какие-либо значения, например, помимо вывода сообщения в консоль, она еще будет возвращать сформированную строку:

```
def some_func(title, tag):
    print(f"title = {title}, tag = {tag}")
```

```
    return f"<{tag}>{title}</{tag}>"
```

Но, при попытке получить это значение увидим None:

```
some_func = func_decorator(some_func)
res = some_func("Python навсегда!", 'h1')

print(res)    # None
```

Конечно, это происходит по той причине, что вложенная функция wrapper ничего не возвращает. Исправим это и сделаем так, чтобы она возвращала значение декорируемой функции func:

```
def func_decorator(func):
    def wrapper(*args, **kwargs):
        print("------ что-то делаем перед вызовом функции ------")
        res = func(*args, **kwargs)
        print("------ что-то делаем после вызова функции ------")
        return res

    return wrapper
```

В результате получили окончательную, универсальную версию декоратора функции и при выполнении программы видим не только его работу, но и корректное возвращаемое значение:

```
------ что-то делаем перед вызовом функции ------
title = Python навсегда!, tag = h1
------ что-то делаем после вызова функции ------
<h1>Python навсегда!</h1>
```

Возможно, у вас все еще остается вопрос, зачем это надо? Где применяется? Приведу такой пример. Предположим, мы хотим протестировать различные функции на скорость их работы. Для этого возьмем функцию, которая реализует медленный алгоритм Евклида для поиска НОД (наибольшего общего делителя) двух натуральных чисел a и b (мы ее с вами ранее уже делали):

```
def get_nod(a, b):
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a
    return a
```

И перед ней запишем следующий декоратор-тестировщик:

```
def test_time(fn):
    def wrapper(*args, **kwargs):
        st = time.time()
        res = fn(*args, **kwargs)
        dt = time.time() - st
        print(f"Время работы: {dt} сек")
```

```
        return res

    return wrapper
```

Этот декоратор замеряет время работы функции и выводит его в консоль. Чтобы воспользоваться функцией time, в самом начале программы импортируем модуль time:

```
import time
```

После этого декорируем функцию get_nod и выполним ее тестирование:

```
get_nod = test_time(get_nod)
res = get_nod(2, 1000000)
print(res)
```

После запуска программы видим время ее выполнения и вычисленный результат:

```
Время работы: 0.030779361724853516 сек
2
```

Мало того, если у нас будет еще какая-либо функция, например, тот же алгоритм Евклида, но с быстрой реализацией:

```
def get_fast_nod(a, b):
    if a < b:
        a, b = b, a
    while b:
        a, b = b, a % b

    return a
```

То мы легко можем применить тот же самый декоратор для тестирования скорости работы этих двух функций:

```
get_nod = test_time(get_nod)
get_fast_nod = test_time(get_fast_nod)
res = get_nod(2, 1000000)
res2 = get_fast_nod(2, 1000000)
print(res, res2)
```

Получим результаты:

```
Время работы: 0.022696971893310547 сек
Время работы: 0.0 сек
2 2
```

То есть, с помощью декоратора test_time можно выполнять тестирование скорости работы любых функций. И это очень удобно!

7.15.2. Применение декоратора с помощью операции @

Так как декораторы – это общепринятый прием модификации функций, то специально для них в Python существует довольно удобная синтаксическая

конструкция, позволяющая прописывать декоратор сразу при объявлении функции. Например, вместо выполнения команды:

```
get_nod = test_time(get_nod)
```

можно test_time применить сразу при объявлении функции get_nod. Это делается с помощью специального символа @, за которым следует имя декоратора:

```
@test_time
def get_nod(a, b):
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a
    return a
```

И то же самое для второй функции get_fast_nod:

```
@test_time
def get_fast_nod(a, b):
    if a < b:
        a, b = b, a
    while b:
        a, b = b, a % b

    return a
```

В результате, эти функции, как бы, «обертываются» декоратором test_time и расширяются его функционалом. Если теперь запустить эти функции командами:

```
res = get_nod(2, 1000000)
res2 = get_fast_nod(2, 1000000)
print(res, res2)
```

то увидим тот же самый результат. А если у какой-либо функции убрать применение декоратора, то получим вызов исходной функции без оценки скорости ее работы.

7.15.3. Сохранение имени и описания декорируемой функции

Полученные нами декораторы пока еще не идеальны. Они не сохраняют исходное имя и описание декорируемой функции. О чем здесь речь? Давайте предположим, что имеется следующая функция:

```
def get_max_2(a, b):
    "Функция возвращает максимальное из двух чисел a и b."
    return a if a > b else b
```

У этой функции есть имя get_max_2 и описание – первая строчка в теле функции. Эти данные хранятся в специальных атрибутах объекта-функции:

```
name = get_max_2.__name__
```

```
s_doc = get_max_2.__doc__
```

При их выводе на экран:

```
print('name: ', get_max_2.__name__)
print('doc: ', get_max_2.__doc__)
```

увидим строчки:

```
name:  get_max_2
doc:  Функция возвращает максимальное из двух чисел a и b.
```

Если же эту функцию декорировать, например, следующим простым декоратором:

```
def func_decorator(func):
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        return res

    return wrapper


@func_decorator
def get_max_2(a, b):
    "Функция возвращает максимальное из двух чисел a и b."
    return a if a > b else b
```

то атрибуты __name__ и __doc__ будут связаны с функцией wrapper. Поэтому при их выводе на экран:

```
print('name: ', get_max_2.__name__)
print('doc: ', get_max_2.__doc__)
```

увидим строчки:

```
name:  wrapper
doc:  None
```

Конечно, на работоспособность функции get_max_2 это никак не влияет. Однако если далее по программе используются указанные атрибуты, то работа будет вестись с их некорректными значениями. В ряде случаев это недопустимо. Поэтому возникает задача сохранения исходных имени и описания декорируемой функции. Как это сделать? В самом простом варианте можно вручную внутри func_decorator задать требуемые значения атрибутов __name__ и __doc__ следующим образом:

```
def func_decorator(func):
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        return res

    wrapper.__name__ = func.__name__      # сохраняем имя функции
```

```
    wrapper.__doc__ = func.__doc__       # сохраняем описание функции

    return wrapper
```

После запуска программы увидим искомые значения этих атрибутов:

```
name:  get_max_2
doc:  Функция возвращает максимальное из двух чисел a и b.
```

Но, так как это стандартная операция, то она уже реализована в виде функции update_wrapper из стандартного модуля functools:

```
from functools import update_wrapper
```

Вместо двух строчек копирования значений имени и описания достаточно вызвать эту функцию, передав ей ссылки на функции следующим образом:

```
def func_decorator(func):
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        return res

    update_wrapper(wrapper, func)        # копирование имени и описания
    return wrapper
```

То есть, первый аргумент wrapper – это функция, в которую копируются данные, а второй аргумент func – функция, из которой копируются данные.

Но есть еще более простой и продвинутый способ сохранять свойства __name__ и __doc__. Делается это с помощью декоратора wraps, который импортируется из стандартного модуля functools:

```
from functools import wraps
```

Для копирования атрибутов __doc__ и __name__ достаточно декорировать вложенную функцию wrapper с указанием функции func:

```
def func_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        return res

    return wrapper
```

После выполнения программы увидим, что имя функции и ее описание сохраняются.

7.15.4. Декораторы функций с параметрами

Все ранее рассмотренные декораторы применялись непосредственно к декорируемым функциям без возможности передачи им каких-либо дополнительных значений (аргументов). В то же время такая необходимость

иногда возникает. Например, рассмотренный в предыдущем параграфе декоратор wraps, применяется к функции с передачей ему дополнительного аргумента func. Это пример декоратора с параметром. Давайте посмотрим, как можно создавать такие декораторы.

В качестве примера напишем простой декоратор с одним параметром repeat, который будет указанное число раз вызывать декорируемую функцию и возвращать результат ее последнего вызова. Функция тоже будет простой, которая выводит и возвращает переданную ей строку:

```
def print_string(s):
    print(s)
    return s
```

Как написать декоратор с одним параметром? Первое, что приходит на ум, это добавить этот параметр при определении декоратора, например, следующим образом:

```
def func_repeat(func, repeat=3):
    def wrapper(*args, **kwargs):
        result = None
        for _ in range(repeat):
            result = func(*args, **kwargs)
        return result

    return wrapper
```

И если его применить стандартный образом к функции:

```
@func_repeat
def print_string(s):
    print(s)
    return s
```

а, затем, вызвать:

```
res = print_string("It's a repeat decorator!")
print("result: ", res)
```

то увидим на экране следующие строчки:

```
It's a repeat decorator!
It's a repeat decorator!
It's a repeat decorator!
result:  It's a repeat decorator!
```

Декоратор действительно применяется и три раза вызывает декорируемую функцию. Однако при попытке передать ему именованный аргумент:

```
@func_repeat(repeat=2)
def print_string(s):
    print(s)
```

```
    return s
```

возникнет ошибка:

```
TypeError: func_repeat() missing 1 required positional argument: 'func'
```

говорящая, что пропущен первый необходимый позиционный аргумент, который должен быть ссылкой на декорируемую функцию.

Для корректной реализации такого декоратора следует его обернуть в еще одну внешнюю функцию следующим образом:

```
def decor_repeat(repeat=3):
    def func_repeat(func):
        def wrapper(*args, **kwargs):
            result = None
            for _ in range(repeat):
                result = func(*args, **kwargs)
            return result

        return wrapper
    return func_repeat
```

И применить к функции print_string:

```
@decor_repeat(repeat=2)
def print_string(s):
    print(s)
    return s
```

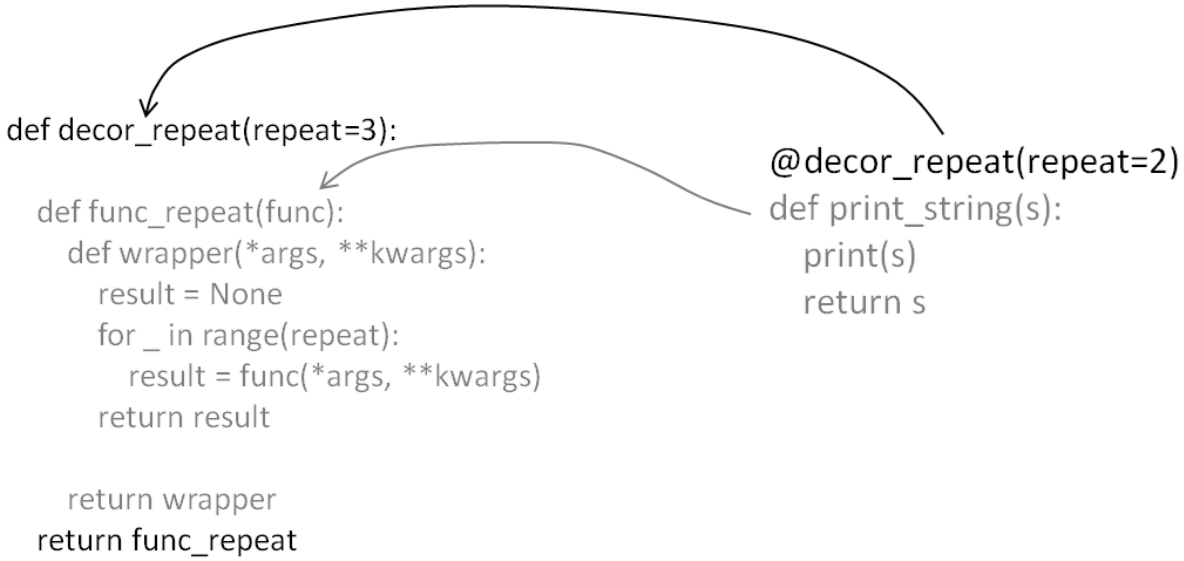
Тогда никаких ошибок не будет и при выполнении программы на экране появятся строчки:

```
It's a repeat decorator!
It's a repeat decorator!
result:  It's a repeat decorator!
```

Как работает такая конструкция? На самом деле все предельно просто. Когда декоратор применяется к функции командой:

```
@decor_repeat(repeat=2)
```

то круглые скобки означают оператор вызова функции. В данном случае функции decor_repeat с аргументом repeat=2. То есть, имеем самый обычный вызов функции. Непосредственно к декоратору он пока не имеет никакого отношения. После вызова функция decor_repeat возвращает ссылку на вложенную функцию func_repeat, к которой уже применяется операция @ для декорирования функции print_string. Соответственно, func_repeat автоматически вызывается с передачей ей ссылки на декорируемую функцию print_string. После своего вызова функция func_repeat возвращает ссылку на вложенную функцию wrapper. Так происходит процесс декорирования.



В явном виде его можно было бы записать так:

```
decorator = decor_repeat(repeat=2)       # создание декоратора
print_string = decorator(print_string)   # применение декоратора
```

Те же самые этапы выполняются и в команде:

```
@decor_repeat(repeat=2)
def print_string(s):
    print(s)
    return s
```

Это практически полный аналог. Получаем декоратор с возможностью передачи ему одного аргумента repeat. Конечно, его можно прописывать, например, и в таком виде:

```
@decor_repeat(5)
def print_string(s):
    print(s)
    return s
```

Разницы особой нет. Или даже так:

```
@decor_repeat()
def print_string(s):
    print(s)
    return s
```

Тогда будет использоваться значение по умолчанию. Но следующая запись не даст ожидаемого эффекта:

```
@decor_repeat
def print_string(s):
    print(s)
    return s
```

После выполнения программы увидим строчку:

```
result:  <function decor_repeat.<locals>.func_repeat.<locals>.wrapper at
0x0000023108309BC0>
```

В целом, я думаю, вы уже понимаете, почему так произошло. Записанный в таком виде декоратор – это аналог команды:

```
print_string = decor_repeat(print_string)
```

который просто возвращает ссылку на внутреннюю функцию func_repeat. И после вызова:

```
res = print_string("It's a repeat decorator!")
```

фактически вызывается функция func_repeat, которая возвращает ссылку на вложенную в нее функцию wrapper. Сама исходная функция print_string не вызывается вообще.

Чтобы последний вариант применения декоратора тоже приводил к желаемым результатам, декоратор decor_repeat нужно немного подкорректировать:

```
def decor_repeat(func_or_repeat=None, *, repeat=3):
    def func_repeat(func):
        def wrapper(*args, **kwargs):
            result = None
            for _ in range(repeat):
                result = func(*args, **kwargs)
            return result

        return wrapper

    if callable(func_or_repeat):    # декоратор вызвали без параметров
        return func_repeat(func_or_repeat)

    if func_or_repeat is not None:  # декоратор вызвали с позиционным
                                    # аргументом
        repeat = func_or_repeat

    return func_repeat        # декоратор вызвали с именованным аргументом
```

Здесь полагается, что внешняя функция decor_repeat в качестве первого позиционного аргумента может принимать либо ссылку на вызываемый объект (то есть, функцию), либо ссылку на число, строку, кортеж и т.п., то есть, не вызываемый объект. Если decor_repeat в параметре func_or_repeat получает ссылку на функцию, то срабатывает условие:

```
    if callable(func_or_repeat):    # декоратор вызвали без параметров
        return func_repeat(func_or_repeat)
```

Здесь стандартная функция callable языка Python возвращает True для вызываемых объектов. Этот вариант описывает поведение команды:

```
@decor_repeat
def print_string(s):
```

```
    ...
```

В этом случае мы самостоятельно вызываем вложенную функцию-декоратор func_decorator с передачей ей ссылки на декорируемую функцию. Она, в свою очередь, возвращает ссылку на функцию wrapper и декорирование выполняется успешно.

Когда декоратор вызывается с позиционным аргументом:

```
@decor_repeat(5)
def print_string(s):
    ...
```

то параметр func_or_repeat принимает значение 5 – это не вызываемый объект, и в функции decor_repeat параметру repeat присваивается это значение 5, т.к. иначе он имел бы значение по умолчанию 3. Затем, просто возвращается функция-декоратор func_decor, которая применяется для декорирования функции.

Последний вариант вызова декоратора:

```
@decor_repeat(repeat=2)
def print_string(s):
    ...
```

отрабатывает стандартным образом, как обычный декоратор с параметрами.

Вот такой прием можно делать, чтобы применять всеми возможными способами декораторы с параметрами. Разумеется, этих параметров может быть несколько, ограничений здесь никаких нет.

ГЛАВА VIII. МОДУЛИ И ПАКЕТЫ

8.1. Импорт стандартных модулей. Команды import и from

Ранее в программах мы с вами выполняли импорт некоторых стандартных модулей, например:

```
import math
import time
```

Это позволяло использовать математические функции и функции для работы с временем. Но что из себя представляют модули math, time и многие другие? В самом простом случае, модуль – это обычный текстовый файл программы на языке Python с расширением «.py». В этом легко убедиться, если в PyCharm нажать и удерживать клавишу Ctrl и щелкнуть по имени модуля, например, math. Откроется файл «math.py» с соответствующим содержимым, в котором прописаны некоторые константы и математические функции.

Давайте теперь посмотрим, что происходит в программе при подключении различных модулей. Пусть, например, импортируются два модуля:

```
import math
import time
```

Чтобы посмотреть все переменные текущего программного модуля, воспользуемся встроенной функцией locals:

```
print(locals())
```

В консоли увидим словарь:

```
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x00000200C0C15F50>, '__spec__': None, '__annotations__': {},
'__builtins__': <module 'builtins' (built-in)>, '__file__':
'C:\\Projects\\python\\course\\main.py', '__cached__': None, 'math':
<module 'math' (built-in)>, 'time': <module 'time' (built-in)>}
```

Последние две записи в нем – это ключи math и time, которые ссылаются на соответствующие импортированные модули. Для более читаемого вида этого словаря можно импортировать еще один модуль:

```
import pprint
```

и затем, вместо функции print, воспользоваться функцией pprint для форматированного вывода информации:

```
pprint.pprint(locals())
```

Получаем на экране следующие строчки:

```
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
```

```
 '__cached__': None,
 '__doc__': None,
 '__file__': 'C:\\Projects\\python\\course\\main.py',
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x000002681BB55F50>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'math': <module 'math' (built-in)>,
 'pprint': <module 'pprint' from 'C:\\Python311\\Lib\\pprint.py'>,
 'time': <module 'time' (built-in)>}
```

Дополнительно появился еще один ключ pprint после импортирования модуля pprint.

Все приведенные здесь ключи (math, pprint, time и т.п.) – это переменные доступные в текущей программе. Например, команда:

```
print(math)
```

выведет на экран строку:

```
<module 'math' (built-in)>
```

которая означает, что переменная math связана с импортированным модулем math. Чтобы посмотреть содержимое этого модуля, можно воспользоваться еще одной встроенной функцией dir:

```
pprint.pprint(dir(math))
```

Увидим следующие начальные строчки:

```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
...
```

Это полный список атрибутов модуля math, включая имена функций и переменных.

Чтобы обратиться к определенной функции нужно написать имя модуля и через точку указать имя атрибута, например:

```
a = math.ceil(1.8)   # обращение и вызов функции ceil
print(a)
print(math.pi)       # обращение к переменной pi
```

По сути, команда:

```
import math
```

автоматически создает в текущей программе пространство имен «math», через которое можно обращаться к любому атрибуту этого модуля. И так с каждым импортируемым модулем.

Приведенные три модуля (math, time, pprint), поставляются вместе с языком Python и составляют его стандартную библиотеку. Полный список модулей этой библиотеки и их подробное описание можно посмотреть на странице официальной документации:

https://docs.python.org/3/library/

А в курсе «Стандартная библиотека Python»:

https://stepik.org/a/259466

приобрести практические навыки работы с многочисленными модулями стандартной библиотеки. Рекомендуется в целом с ней ознакомиться, чтобы представлять возможности уже заложенные в Python, и использовать все богатство ее инструментария в своих программах.

Итак, после выполнения импорта модуля math, у нас в программе появляется переменная math, через которую можно обращаться к атрибутам этого модуля. Однако если далее в программе переменная math будет изменена, например, командой:

```
math = 'математика'
```

то доступ к модулю math будет потерян и дальнейшие попытки обращения к функциям или переменным приведут к ошибкам:

```
a = math.ceil(1.8)   # AttributeError
```

Поэтому нужно аккуратно выбирать имена переменных в своих программах, чтобы они не скрывали существующий функционал.

Если по каким-либо причинам использование переменной math в качестве имени недопустимо, то в момент импорта можно указать другое имя, которое будет связано с этим модулем. Делается это с помощью ключевого слова as, например, следующим образом:

```
import math as mt
```

После этого в программе следует использовать переменную mt в качестве имени модуля. На практике такая подстановка выполняется в двух целях. Во-первых, чтобы избежать возможного конфликта имен в текущем программном модуле и, во-вторых, для удобства, если имя модуля слишком длинное, тогда его можно укоротить придуманным псевдонимом.

Если при импортировании модуля необходимо выбрать лишь часть атрибутов из

всего многообразия, то для этого можно воспользоваться следующей конструкцией:

```
from math import ceil, pi
```

В результате в программу будут импортированы функция ceil и переменная pi. В этом легко убедиться, выполнив команду:

```
pprint.pprint(locals())
```

На экране увидим все доступные переменные:

```
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': None,
 '__file__': 'C:\\Projects\\python\\course\\main.py',
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x000002737D3A5F50>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'ceil': <built-in function ceil>,
 'pi': 3.141592653589793,
 'pprint': <module 'pprint' from 'C:\\Python311\\Lib\\pprint.py'>,
 'time': <module 'time' (built-in)>}
```

Здесь уже нет ключа math, но в программе появились ключи ceil и pi. То есть, к ним теперь можно обращаться напрямую без указания имени модуля math:

```
print(ceil(1.8))
print(pi)
```

Однако при таком импорте легко столкнуться с ситуацией, когда импортируемая функция (или переменная) переопределяется в текущей программе. Например, если прописать свою собственную функцию ceil:

```
def ceil(x):
    print("своя функция ceil")
    return x
```

то именно она и будет вызвана, потому что сначала происходит импорт, а затем объявление функции. Если же объявление функции сделать до импорта, то, наоборот, будет использована библиотечная функция, а пользовательская затрется. То есть, берется то, что определяется последним.

Для разрешения такой ситуации (конфликта имен) у импортируемых элементов также можно указывать псевдонимы через ключевое слово as. Например:

```
from math import ceil as m_ceil, pi
```

Теперь для обращения к библиотечной функции ceil следует прописывать имя

m_ceil:

```
print(m_ceil(1.8))
```

Если после ключевого слова import прописать *, то полученная конструкция выполнит импорт сразу всех атрибутов из указанного модуля. Например, команда:

```
from math import *
```

импортирует все переменные и функции непосредственно в глобальную область текущей программы, что легко может вызвать конфликт имен (потерю части функционала). Поэтому делать такой импорт крайне не рекомендуется. Наперед даже точно нельзя предугадать, какие имена будут присутствовать в импортируемых модулях, так как они могут меняться от версии к версии.

И еще один важный момент. Формально в импорте можно через запятую перечислять имена импортируемых модулей, например, так:

```
import pprint, time, random
```

Но стандарт PEP8 делать этого не рекомендует. Вместо одного такого импорта следует записывать три, каждый с новой строки:

```
import pprint
import time
import random
```

Это правило не касается конструкции «from … import …» Здесь, конечно же, указываются имена через запятую.

Наконец последняя рекомендация – располагать импорты в самом начале программы. Редко от этого правила отступают, но в большинстве случаев импортирование выполняется в первых строчках программы.

8.2. Импорт собственных модулей

На данный момент мы с вами уже знаем, что модули представляют собой обычный текстовый файл с расширением «.py». А раз так, то ничто нам не мешает создать свой собственный модуль. Давайте это сделаем. В текущем рабочем каталоге проекта создадим еще один файл с некоторым именем, например, «mymodule.py». И пропишем в нем следующие строчки:

```
NAME = 'mymodule'

def floor(x):
    print("функция из mymodule")
    return int(x) if x >= 0 else int(x) - 1
```

Здесь определена переменная NAME и функция floor. Именно они образуют атрибуты модуля «mymodule.py». Обратите внимание, что локальные переменные функции, например, ее параметр `x`, в набор атрибутов модуля не попадают, только

глобальные определения.

Теперь можно импортировать этот модуль в программу командой:

```
import mymodule
```

Выведем на экран переменные текущей программы:

```
print(locals())
```

Получим следующий словарь:

```
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x0000020E42946050>, '__spec__': None, '__annotations__': {},
'__builtins__': <module 'builtins' (built-in)>, '__file__':
'C:\\Projects\\python\\course\\main.py', '__cached__': None, 'mymodule':
<module 'mymodule' from 'C:\\Projects\\python\\course\\mymodule.py'>}
```

Видим здесь ключ 'mymodule', который связан с импортированным модулем. Следовательно, к переменной NAME и функции floor можно обратиться с помощью команд:

```
res = mymodule.floor(-5.4)
print(mymodule.NAME)
```

Или же воспользоваться конструкцией:

```
from mymodule import floor
```

для непосредственного импорта любого атрибута из модуля mymodule. Благодаря этому функцию floor можно вызывать напрямую:

```
print(floor(-5.4))
```

Но что будет, если в файле «mymodule.py» дополнительно прописать импорт, например, библиотеки math следующим образом:

```
import math
```

В этом случае в модуле mymodule появляется еще одна переменная с именем math. И при его импорте:

```
import mymodule
```

можно обращаться к функциям математической библиотеки следующим образом:

```
a = mymodule.math.floor(-5.6)
print(a)
```

Получаем иерархию модулей. Сначала идет обращение к mymodule, затем к math, а потом к функции floor математической библиотеки.

Конечно, чтобы не создавать таких цепочек, можно было бы импортировать

библиотеку math командой:

```
from math import *
```

В результате получаем неявное расширение модуля mymodule за счет модуля math. И все функции из math теперь напрямую можно вызывать из mymodule:

```
a = mymodule.floor(-5.6)
b = mymodule.ceil(-5.6)
print(a, b)
```

Однако функция floor была переопределена в mymodule, поэтому вызывается не библиотечный вариант, а переопределенный.

Как мы уже говорили, импорт командой:

```
from math import *
```

делать крайне не рекомендуется, чтобы не смешивать пользовательские и стандартные функции. Лучше либо перечислять необходимые элементы для импорта, например:

```
from math import pi, ceil
```

либо импортировать с пространством имен:

```
import math
```

Давайте детальнее разберемся, как работает импорт модулей. Первый вопрос, откуда Python «знает», где искать импортируемые модули? На самом деле, порядок поиска прописан в специальной коллекции path модуля sys:

```
import sys
pprint.pprint(sys.path)
```

В моем случае после выполнения отображаются следующие пути:

```
['C:\\Projects\\python\\course',
 'C:\\Projects\\python\\course',
 'C:\\Python311\\python311.zip',
 'C:\\Python311\\DLLs',
 'C:\\Python311\\Lib',
 'C:\\Python311',
 'C:\\Users\\Sergey\\venv',
 'C:\\Users\\Sergey\\venv\\Lib\\site-packages']
```

Здесь первые строчки – это каталог с рабочим проектом, а далее, идут дополнительные пути поиска, например, для файлов стандартной библиотеки. Поэтому если переместить файл «mymodule.py» из текущего рабочего каталога, в какую-нибудь вложенную папку, например, с именем folder, то при импорте:

```
import mymodule
```

получим ошибку:

```
ModuleNotFoundError: No module named 'mymodule'
```

так как в путях коллекции path отсутствует подкаталог folder. Но для успешного импорта менять коллекцию path не потребуется. Достаточно, дополнительно указать папку folder, в которой находится модуль mymodule:

```
import folder.mymodule
```

Конечно, и в список path можно добавить путь для поиска модулей:

```
sys.path.append(r"C:\PythonProjects\example\folder")
```

Тогда команда:

```
import mymodule
```

успешно найдет и импортирует mymodule. Но это делается крайне редко, обычно добавляют имена подкаталогов через точку.

Обратите внимание, что в момент выполнения импорта модуль преобразуется интерпретатором языка Python сначала в байт-код, а затем, запускается на исполнение. Это важный момент. Например, если в «mymodule.py» прописать команду:

```
print(NAME)
```

и запустить программу, в которой выполнен импорт модуля mymodule, то в консоли увидим результат срабатывания функции print:

```
mymodule
```

Причем эта команда сработала в момент импорта модуля. Это означает, что если в mymodule прописать следующий цикл:

```
for _ in range(5):
    print(NAME)
```

то он будет выполнен в момент импорта. Это следует иметь в виду, когда формируется программный модуль. Тогда получается, что подключаемые модули должны содержать исключительно определения переменных, функций и других объектов языка Python, но не их вызовы? На самом деле, не обязательно. В каждом модуле есть специальная переменная __name__, которая содержит имя модуля, если он запускается при импорте:

```
print(__name__)
```

и значение "__main__", если модуль запускается как самостоятельная программа. И это свойство часто используют для контроля исполнения модуля. Если в нем нужно прописать программу, выполняемую только при непосредственном запуске модуля, то ее следует поместить в следующее условие:

```
if __name__ == "__main__":
    print("самостоятельный запуск")
else:
    print("запуск при импорте")
```

Такую проверку, например, часто можно увидеть в программах на Python и вы теперь знаете, для чего она нужна.

Давайте посмотрим, что произойдет, если в файле модуля выполнить импорт файла с запускаемой программой. В моем случае – это файл «main.py» и, соответственно, модуль main:

```
import main
```

Чтобы результат был виден, в файл «main.py» добавим команду:

```
print("main.py")
```

а в файл «mymodule.py»:

```
print("mymodule")
```

После запуска видим, что сначала был выполнен файл «main.py», затем, «mymodule.py», а потом, снова «main.py»:

```
main.py
mymodule.py
main.py
```

Но почему выполнение не пошло дальше по цепочке? Дело в том, что модуль импортируется только один раз. Например, если прописать два импорта подряд:

```
import mymodule
import mymodule
```

то увидим только одно сообщение. Это, как раз, и доказывает, что модуль был выполнен только один раз. При необходимости в программе можно делать повторный импорт с помощью функции reload из стандартного модуля importlib, но необходимость этого возникает крайне редко.

8.3. Пакеты (package)

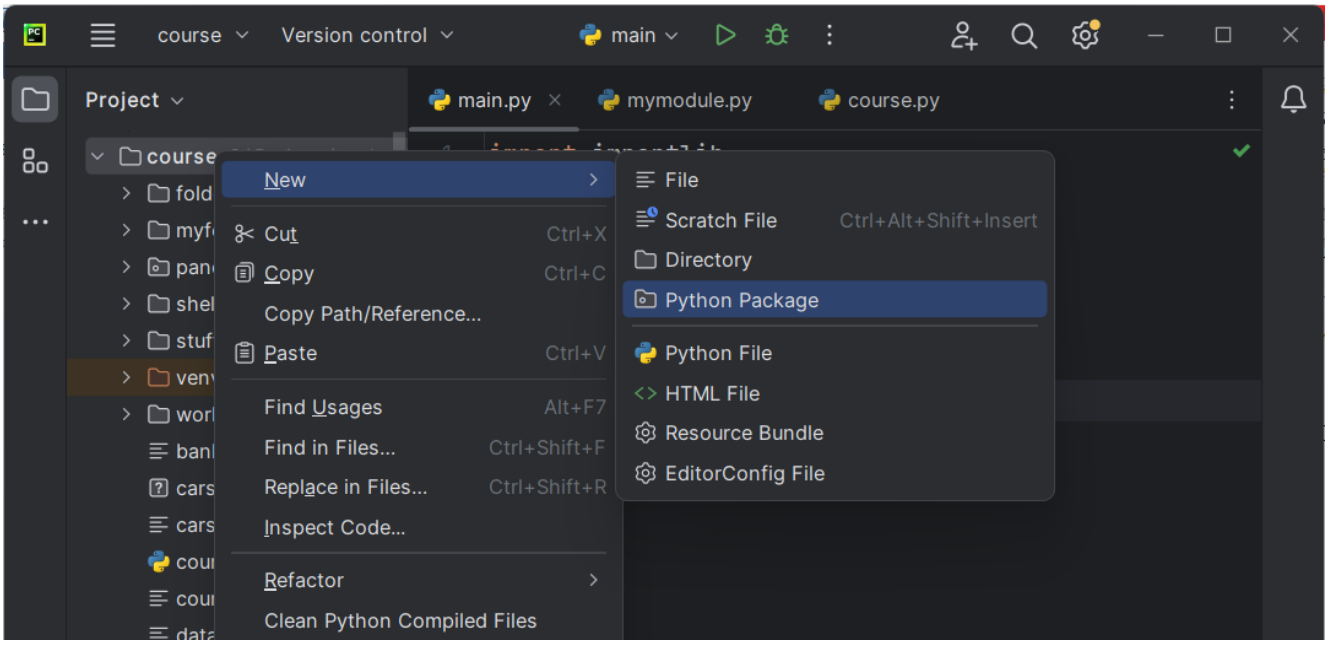
Помимо модулей, которые представляют собой отдельные файлы с текстами программ, в языке Python существуют еще пакеты (package) – это отдельный каталог с набором модулей, которые, как правило, решают некоторую единую задачу.

Давайте в качестве примера создадим пакет из модулей по обучающим курсам:

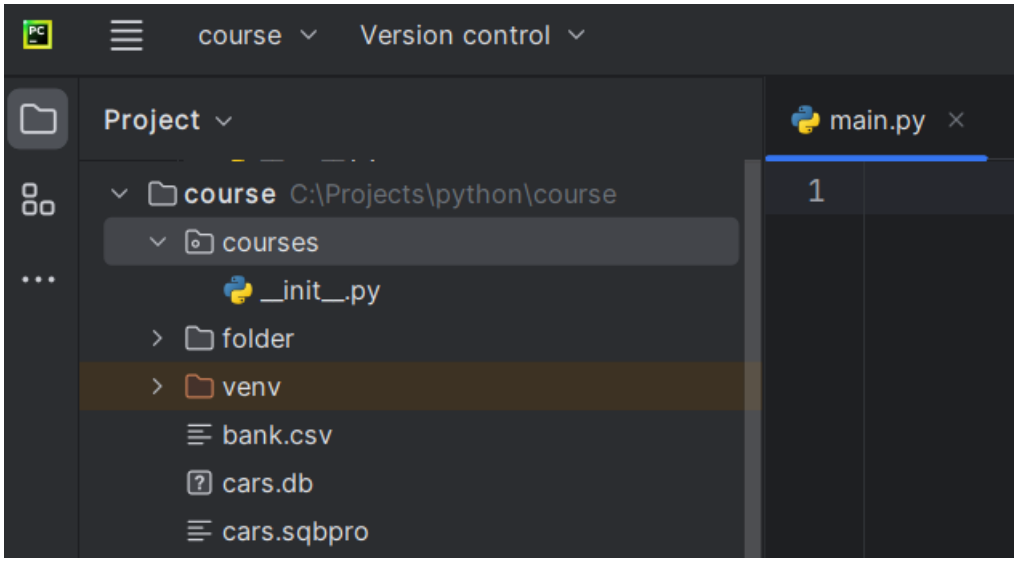
- HTML
- Java
- PHP

- Python

В PyCharm для создания нового пакета нужно щелкнуть правой кнопкой мыши по каталогу с проектом и из выпадающего меню выбрать «New -> Python Package» и в появившемся окне набрать имя пакета (каталога). В нашем примере пусть он называется courses:



В результате в рабочем каталоге появится подкаталог с именем courses, содержащий один обязательный файл «__init__.py». Чуть позже мы узнаем, для чего он нужен.



На данный момент созданный пакет пустой, в нем нет ни одного модуля. Добавим их, то есть, добавим обычные python-файлы в каталог courses. Для примера сформируем их следующим образом:

- имя файла (модуля): **html.py**

```
def get_html():
```

```
    print("курс по HTML")
```

- имя файла (модуля): **java.py**

```
def get_java():
    print("Курс по Java")
```

- имя файла (модуля): **php.py**

```
def get_php():
    pass


def get_mysql():
    pass
```

- имя файла (модуля): **python.py**

```
def get_python():
    print("курс по Python")
```

И, кроме того, в файл __init__.py добавим строчку:

```
NAME = "package courses"
```

Обратите внимание, что для корректной обработки модулей в пакете, все файлы следует создавать с кодировкой UTF-8.

8.3.1. Файл __init__.py

Итак, пакет наполнен содержимым (модулями) и следующим шагом выполним его импорт в исполняемой программе. Делается это абсолютно так же, как и с модулями, только после ключевого слова import прописывается название каталога (пакета), например, следующим образом:

```
import courses
```

С помощью стандартной функции dir выведем на экран содержимое модуля courses:

```
print(dir(courses))
```

Видим следующий список, в котором фигурирует переменная NAME, прописанная непосредственно в файле «__init__.py»:

```
['NAME', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__path__', '__spec__']
```

Выведем значение этой переменной:

```
print(courses.NAME)
```

В консоли отображается строчка:

```
package courses
```

Это говорит о том, что при импорте пакета был автоматически выполнен файл «__init__.py». Остальные же файлы пакета были проигнорированы. В действительности так все и должно работать. Файл «__init__.py» – это инициализатор пакета, в котором прописывается то, что следует импортировать при импорте пакета. Например, можно импортировать заготовленные файлы модулей. Давайте это сделаем. Если в файле «__init__.py» прописать команду:

```
import courses.python

NAME = "package courses"
```

то автоматически будет импортирован модуль python. Обратите внимание, что вначале указан подкаталог courses, так как модули находятся по нестандартному пути (в подкаталоге courses).

Теперь, после запуска основной программы, мы видим, что в пространстве имен пакета courses появилось имя python:

```
['NAME', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__path__', '__spec__', 'courses', 'python']
```

Вызовем из него его функцию get_python командой:

```
courses.python.get_python()
```

Она отработает и выведет на экран строчку:

```
курс по Python
```

Однако прописывать имя пакета python для вызова функции get_python не очень удобно, поэтому нередко в инициализаторе пакетов импорт выполняют с помощью команды:

```
from courses.python import get_python
```

Теперь ту же самую функцию в основной программе можно вызвать так:

```
courses.get_python()
```

Но все же у такого способа импорта модулей в инициализаторе пакета есть один существенный недостаток. Здесь, фактически, указывается полный путь к модулю. Это называется **абсолютным импортом**. Представьте, например, что в будущем, по каким-либо причинам, требуется поменять название пакета. Тогда и все абсолютные импорты также придется переписывать. Поэтому лучше воспользоваться **относительным** способом импортирования. Для этого, вместо имени основного подкаталога courses, ставится точка:

```
from .python import get_python
```

Эта точка означает «использовать текущий каталог». Так гораздо практичнее и,

кроме того, мы теперь совершенно не привязаны к названию пакета.

Чтобы сразу импортировать несколько модулей пакета, все их можно перечислить через запятую следующим образом:

```
from . import html, java, php, python
```

Если же нужно выполнить импорт всех данных из нескольких модулей пакета, то это можно сделать так:

```
from .html import *
from .java import *
from .php import *
from .python import *
```

8.3.2. Коллекция __all__

Ранее мы с вами отмечали, что в программах так лучше не делать. Однако внутри пакетов можно в качестве исключения из общего правила. Почему такой импорт целесообразен? Дело в том, что наполнение внутри пакетов постоянно совершенствуется, появляются новые функции, переменные, классы. И чтобы каждый раз не менять их список в импорте прописывают звездочку. При этом конфликта имен можно избежать, контролируя импортируемые данные. Для этого внутри модуля прописывается специальная переменная __all__ со списком импортируемых данных, если используется оператор «*». Например, в модуле php.py можно прописать:

```
__all__ = ['get_php', 'get_mysql']
```

Тогда в пакет будут импортированы обе функции, а команды вызова имеют вид:

```
courses.get_php()
courses.get_mysql()
```

Но если оставить только какую-нибудь одну:

```
__all__ = ['get_php']
```

то следующий вызов приведет к ошибке:

```
courses.get_mysql()   # AttributeError
```

Коллекцию __all__ можно записывать и в инициализаторе пакета, в файле «__init__.py», например, так:

```
from .html import *
from .java import *
from .php import *
from .python import *

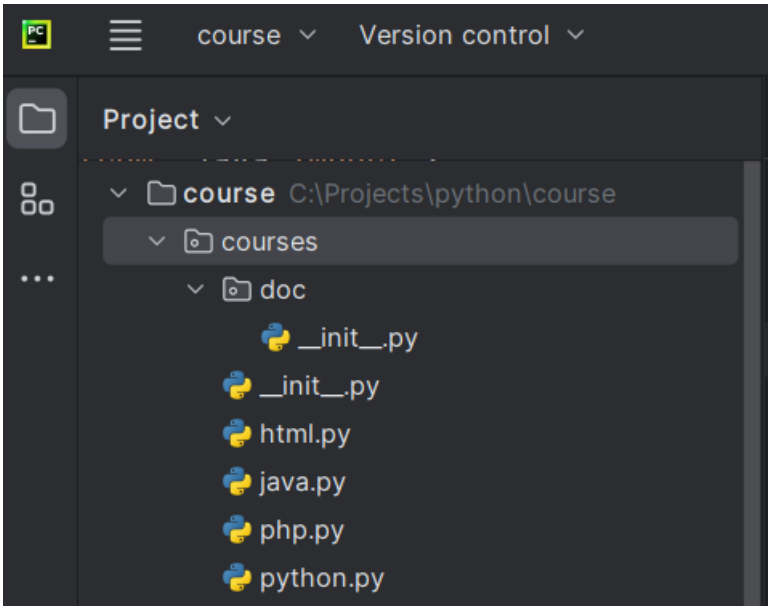
NAME = "package courses"

__all__ = ['python', 'php']
```

Тогда будут импортированы только модули python и php. Переменная NAME также будет отсутствовать при импорте пакета.

8.3.3. Вложенные пакеты

Язык Python позволяет создавать вложенные пакеты, когда один пакет вкладывается в другой. Давайте рассмотрим это на конкретном примере. Создадим еще один вложенный подкаталог внутри каталога courses с именем doc:



И затем разместим в этом вложенном пакете два файла:

- имя файла (модуля): **java_doc.py:**

```
doc = """Документация по языку Java"""
```

- имя файла (модуля): **python_doc.py:**

```
doc = """Документация по языку Python"""
```

После этого в файле инициализатора пакета doc пропишем импорт этих модулей:

```
from .python_doc import *
from .java_doc import *
```

А в инициализаторе основного пакета – импорт вложенного пакета:

```
from .doc import *
```

Скорректируем в этом же файле коллекцию:

```
__all__ = ['python', 'php', "doc"]
```

После всех этих действий в основной программе можно обращаться к переменным вложенного пакета следующим образом:

```
print(courses.python_doc.doc)
```

Мало того, в самих модулях вложенного пакета можно обращаться к модулям внешнего пакета. Для этого при импорте следует указать две точки (переход во внешний каталог) и затем имя модуля, например (в файле python_doc.py):

```
from ..python import get_python
```

Воспользуемся этой функцией для формирования строки описания:

```
doc = """Документация по языку Python: """ + get_python()
```

соответственно, переписав ее в файле python.py, следующим образом:

```
def get_python():
    return "курс по Python"
```

После запуска основной программы, в консоли увидим вывод строки:

```
Документация по языку Python: курс по Python
```

Несмотря на гибкость совместного использования вложенных пакетов, на практике следует вложения делать максимально независимыми, чтобы обращение шло из внешнего пакета, но не наоборот. Благодаря этому вложения, при необходимости, можно спокойно перемещать в другие пакеты и там их использовать.

8.4. Установка сторонних пакетов

Несмотря на то, что стандартная библиотека языка Python содержит множество полезных модулей, они все равно не могут покрыть всех потребностей в программировании. Например, при решении инженерных задач часто используются внешние пакеты, не входящие в стандартную библиотеку:

- NumPy – для работы с многомерными массивами;
- Matplotlib – для отображения графиков;
- ScLearn – содержит различные алгоритмы обработки данных.

Если же необходимо активно работать с графикой, то нередко используют:

- Pygame – создание простой 2D-графики.

А при разработке backend’ов сайтов часто применяются фреймворки:

- FastAPI – облегченный фреймворк для быстрой разработки сайтов;
- Django – более функциональные фреймворк для продвинутых сайтов.

И так далее. Сторонние модули разрабатываются и обновляются постоянно. Для языка Python за долгие годы были разработаны сотни тысяч пакетов для самых разных нужд. В этом одно из преимуществ данного языка – его удобно применять для решения самых разнообразных задач.

Чтобы увидеть, какие пакеты уже установлены для текущего интерпретатора языка Python, нужно открыть командное окно или вкладку «Terminal» в PyCharm,

и набрать команду:

pip list

Отобразится список пакетов с номерами версий. В моем случае начало списка следующее:

```
Package                   Version
------------------------- ------------------
absl-py                   2.1.0
annotated-doc             0.0.4
anyio                     4.14.2
beautifulsoup4            4.12.3
...
```

Ключевое слово pip в команде означает обращение к специальному модулю, отвечающему за установку, удаление и обновление сторонних пакетов для интерпретатора.

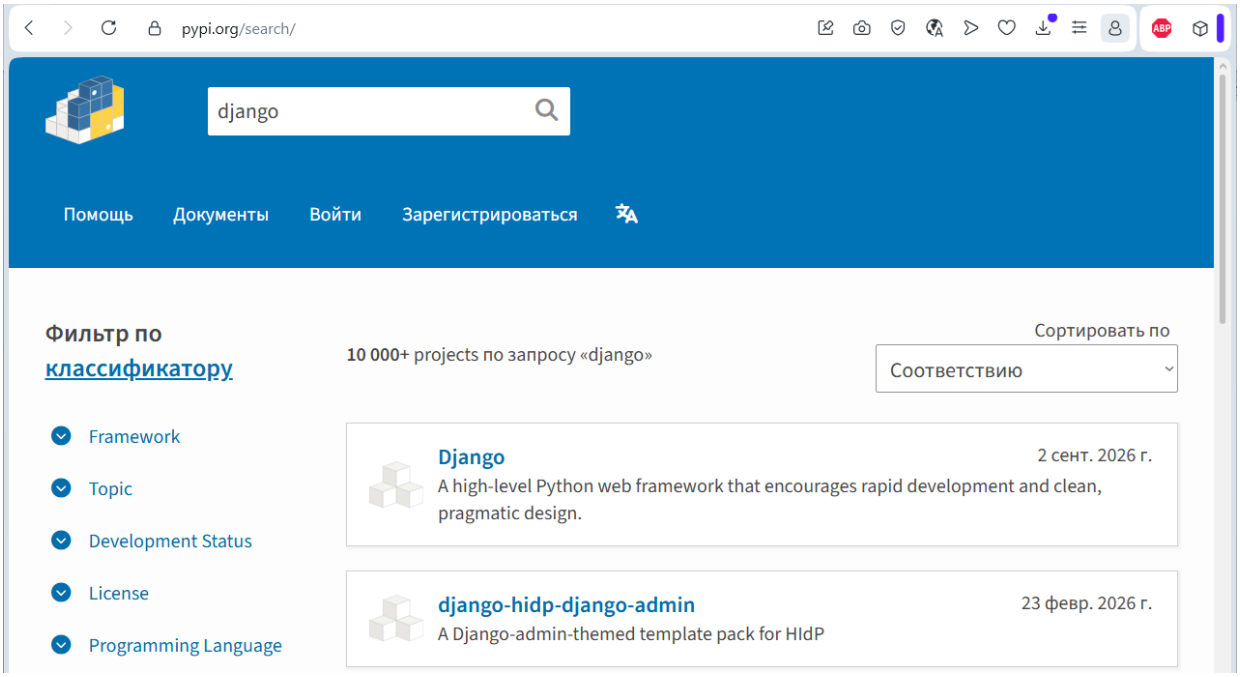
Чтобы установить какой-либо сторонний пакет (модуль) в самом простом случае достаточно знать его название. Сама установка выполняется командой:

pip install <название пакета>

Но как нам узнать, какие пакеты существуют, как они называются и как их применять в программе? Для этого существует специальный сайт (репозиторий) со списком сторонних пакетов и их кратким описанием:

https://pypi.org

Здесь, как в поисковой системе, нужно набрать примерное название пакета и выполнить поиск. Например, если набрать «django», то увидим следующий список из различных версий и компонентов для фреймворка Django:



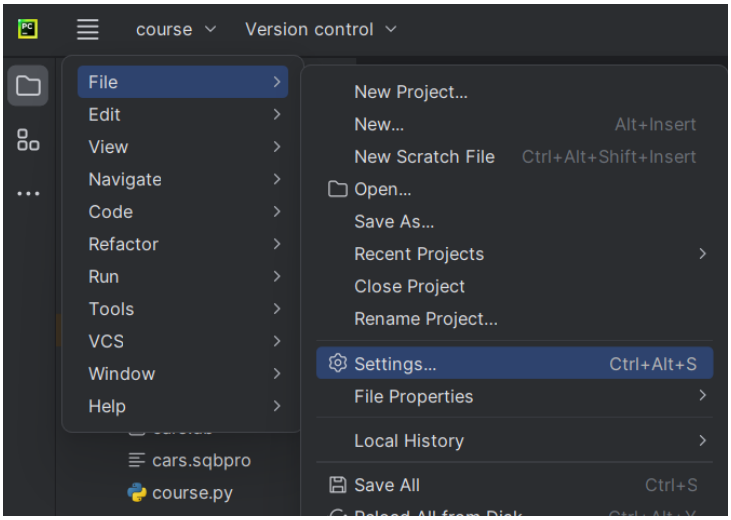
В самом начале, как правило, появляются наиболее релевантные результаты. И, действительно, здесь первой строчкой находится пакет Django – фреймворк для разработки бэкенда сайтов. Если кликнуть по нему, то появится краткое описание, ссылка на подробную документацию и команда для установки этого пакета:

```
pip install Django
```

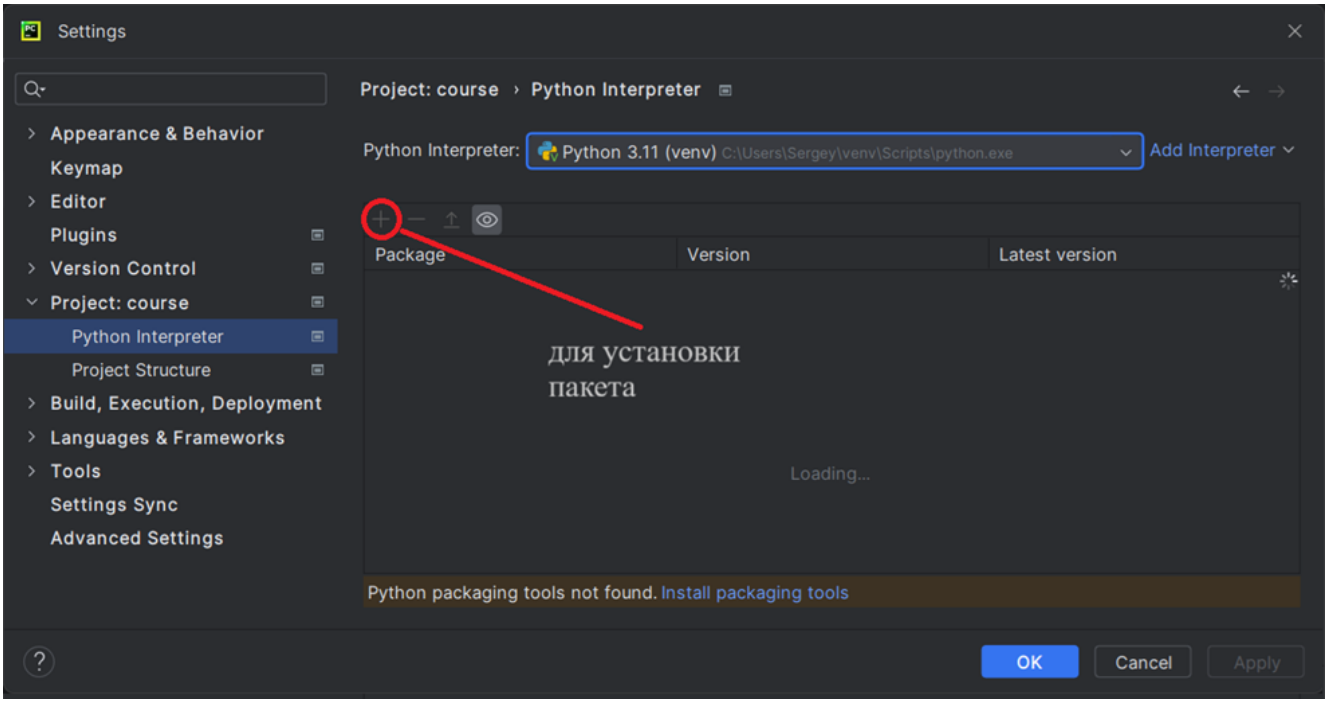
Если выполнить ее в записанном виде, то будет найдена и установлена последняя версия пакета. При необходимости, можно явно указать версию для установки. Для этого после имени пакета ставится два равно и пишется номер версии:

```
pip install Django==6.1.1
```

Другой способ установки пакетов выполняется непосредственно через интерфейс программы PyCharm. Для этого нужно выбрать «File->Settings»:



Затем, в открывшемся окне, выбрать «Python Interpreter»:



Увидим окно с текущим интерпретатором языка Python и набором внешних установленных пакетов, а также их версий. Чтобы добавить новый пакет, выбираем «+» и в открывшемся окне через поиск находим нужный пакет, после чего нажимаем кнопку «Install Package».

К сожалению, на момент написания книги этот функционал в России работает только через VPN. Поэтому приходится выполнять установку пакетов через команду pip в окне «Terminal», как это было описано в самом начале.

Обратите внимание, что установка пакетов выполняется строго для текущего выбранного интерпретатора. Поэтому, если на компьютере установлено несколько версий языка Python, то у каждой из них будет свой набор установленных внешних модулей. Это очень удобно, так как некоторые пакеты привязаны к определенным версиям языка Python, соответственно, разные версии Python могут иметь разные версии установленных пакетов.

Путь, по которому располагаются установленные модули, можно увидеть в специальной коллекции sys.path, о которой мы с вами ранее уже говорили. Благодаря заранее настроенным путям, все установленные внешние пакеты могут быть импортированы в программу без дополнительных указаний каталогов и подкаталогов.

Также с помощью команды pip можно устанавливать сразу несколько модулей. Это называется пакетной установкой. Выполняется она командой:

pip install -r <текстовый файл>

В текстовом файле на каждой отдельной строчке должны быть прописаны имена устанавливаемых модулей, часто с указанием требуемых версий, но можно и без них тогда будет установлена последняя версия пакета. Вот пример подобного файла с именем requirements.txt:

```
absl-py==2.1.0
annotated-doc==0.0.4
anyio==4.14.2
beautifulsoup4==4.12.3
certifi==2024.8.30
charset-normalizer==3.4.0
click==8.4.2
...
```

Этот файл можно сформировать вручную. Однако пакетная установка в основном используется для переноса проекта с одного компьютера на другой. Поэтому, в файле нужно прописывать все установленные внешние зависимости. Чтобы не делать этого вручную, достаточно выполнить команду (здесь имя файла может быть любым, но часто используют requirements.txt):

pip freeze > requirements.txt

После этого в рабочем каталоге проекта появится текстовый файл requirements.txt

с набором всех установленных модулей и пакетов для текущего интерпретатора. Затем этот файл прописывается в команде:

```
pip install -r requirements.txt
```

и указанные зависимости в соответствии с их версиями устанавливаются на компьютер для текущего интерпретатора языка Python (если их ранее не было).

ГЛАВА IX. РАБОТА С ФАЙЛАМИ

До сих пор, при написании программ, мы с вами сохраняли данные через переменные, то есть, непосредственно в памяти устройства. В то же время часто имеется возможность сохранять данные на внешних носителях, к которым относятся HDD и SSD-диски, флэш-память и так далее. Данные в них хранятся в виде файлов, которые сохраняются и после выключения устройства. Что такое файлы, каталоги и для чего они нужны, вы все прекрасно уже знаете, т.к. программы на Python хранятся непосредственно в файлах. Поэтому сразу перейдем к рассмотрению базовых возможностей языка по работе с этими внешними хранилищами.

9.1. Функция open(). Чтение данных из текстового файла

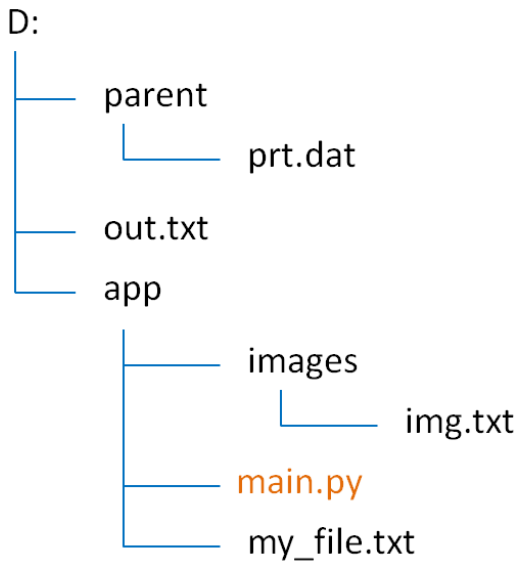
Чтобы работать с каким-либо файлом в программе его нужно открыть, то есть, открыть файловый поток. За это отвечает встроенная функция open, которая имеет следующий синтаксис:

open(file [, mode='r', encoding=None, …])

По умолчанию она открывает указанный файл на чтение данных. Основные ее параметры, следующие:

- file – путь к файлу (вместе с его именем);
- mode – режим доступа к файлу (чтение/запись);
- encoding – кодировка файла.

Чтобы воспользоваться этой функцией, нужно правильно уметь записывать первый аргумент – путь к файлу. Давайте посмотрим различные варианты его определения. Представим, что файл с текстом программы «main.py» находится в рабочем каталоге app:



Тогда, чтобы обратиться к внешнему файлу «my_file.txt», путь можно записать в виде:

```
"my_file.txt"
```

Или так:

```
"d:\\app\\my_file.txt"
```

или с использованием слешей:

```
"d:/app/my_file.txt"
```

Последние два варианта представляют собой абсолютный путь к файлу, то есть, полный путь, начиная с указания диска. Причем обычно используют слеш (/) в качестве разделителя: так короче писать и такой путь будет корректно восприниматься как под ОС Windows, так и Linux. Первый же вариант – это относительный путь (относительно каталога с исполняемым файлом).

Теперь, предположим, что требуется обратиться к файлу «img.txt». Это можно сделать через относительный путь:

```
"images/img.txt"
```

или через абсолютный:

```
"d:/app/images/img.txt"
```

Для доступа к файлу «out.txt» пути записываются следующим образом:

```
"../out.txt"
"d:/out.txt"
```

Обратите внимание, здесь две точки означают переход к родительскому каталогу, то есть, выход из подкаталога app на один уровень вверх.

И, наконец, для доступа к файлу «prt.dat» пути можно прописывать так:

```
"../parent/prt.dat"
"d:/ parent/prt.dat"
```

Вот принцип, по которому указываются пути к файлам. Для примера откроем текстовый файл с именем «my_file.txt», который находится в том же каталоге, что и файл с исполняемой программой. В этом случае путь можно записать, просто указав имя файла:

```
file = open("my_file.txt")
```

В результате переменная file будет ссылаться на файловый объект, через который и происходит работа с файлом. Обратите внимание, что при указании несуществующего файла, например:

```
file = open("my_file2.txt")  # FileNotFoundError (если файл не существует)
```

возникнет ошибка FileNotFoundError. Поэтому маршруты к файлам следует прописывать аккуратно. Позже мы увидим, как можно обрабатывать такие

ошибки, чтобы программа не прерывалась при неверно указанном имени файла.

Итак, функция open по умолчанию открывает файл на чтение. Это значит, что в данном режиме мы можем только считывать информацию из файла, но не записывать. Как прочитать данные из файла? Для этого у файлового объекта существует несколько методов. Например, если выполнить метод read:

```
data = file.read()
print( data )
```

то будет прочитан весь файл целиком и результат возвращен в виде одной строки.

Обратите внимание, если при чтении данных из файла получаются какие-то кракозябры, например:

```
вЂ” РЎРєР°Р¶Рё-РєР°, РґРЏРґР, РІРµРґСЊ РЅРµ РґР°СЂРѕРј
РњРѕСЃРєРІР°, СЃРїР°Р»РµРЅРЅР°СЏ РїРѕР¶Р°СЂРѕРј,
Р¤СЂР°РЅС†СѓР·Сѓ РѕС‚РґР°РЅР°?
Р’РµРґСЊ Р±С‹Р»Рё Р¶ СЃС…РІР°С‚РєРё Р±РѕРµРІС‹Рµ,
Р”Р°, РіРѕРІРѕСЂСЏС‚, РµС‰С‘ РєР°РєРёРµ!
РќРµРґР°СЂРѕРј РїРѕРјРЅРёС‚ РІСЃСЏ Р РѕСЃСЃРёСЏ
РџСЂРѕ РґРµРЅСЊ Р‘РѕСЂРѕРґРёРЅРѕ!
```

то это явный признак несовпадения кодировки текста в файле и используемой кодировки при его чтении. В приведенном примере метод read читает данные в кодировке windows-1251, а файл имеет кодировку UTF-8. Чтобы кодировки совпали в функции open нужно явно указать ее через именованный аргумент encoding следующим образом:

```
file = open("myfile.txt", encoding="utf-8")
```

Теперь все сработает корректно и на экране увидим вполне читаемые символы:

```
— Скажи-ка, дядя, ведь не даром
Москва, спаленная пожаром,
Французу отдана?
Ведь были ж схватки боевые,
Да, говорят, еще какие!
Недаром помнит вся Россия
Про день Бородина!
```

Но, что если требуется прочитать из файла всего несколько символов, а не все содержимое? Для этого достаточно в методе read первым аргументом указать максимальное количество читаемых символов, например, так:

```
print(file.read(4))
```

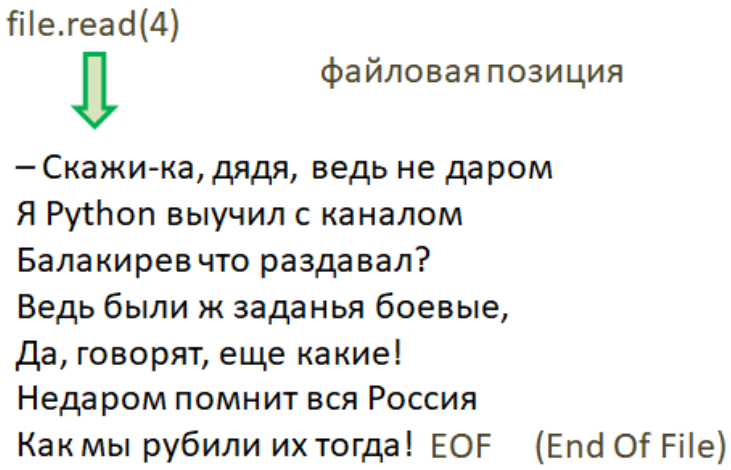
Из файла будут считаны первые четыре символа:

```
— Ск
```

Если мы снова вызовем метод:

```
print(file.read(4))
```

то будут прочитаны следующие четыре символа. Почему так происходит? Откуда функция read «знает», с каких символов продолжать чтение данных? Все дело в так называемой файловой позиции (зеленая стрелка на рисунке ниже), которая указывает, с какого места производить следующее считывание информации. Благодаря ей мы можем последовательно читать данные, просто вызывая метод read. А когда доходим до конца файла, то возвращается специальный символ EOF, означающий конец файла.



Положение файловой позиции можно менять. Для этого существует специальный метод:

file.seek(offset[, from_what])

Например, следующая команда:

```
file.seek(0)
```

устанавливает позицию в начало данных. Поэтому следующие строчки:

```
print(file.read(4))
file.seek(0)
print(file.read(4))
```

будут считывать одни и те же первые символы из файла:

```
– Ск
– Ск
```

Если же нужно определить текущую позицию в файле, то можно воспользоваться методом tell следующим образом:

```
pos = file.tell()   # pos = 8
print(pos)
```

Обратите внимание, что метод tell возвращает положение файловой позиции, выраженной в байтах, а не в символах. В чем разница? Например, кодовая таблица UTF-8 все русские символы кодирует двумя байтами (и не только их).

Поэтому при чтении первых четырех символов методом `read(4)` файловая позиция смещается не на четыре, а на восемь байт:

```
data = file.read(4)
pos = file.tell()
print(pos)          # pos = 8
```

При другой кодировке файла, например, Windows-1251, каждому символу соответствует один байт. И та же самая программа выдаст значение 4. Это следует иметь в виду при работе с файлами.

Следующий полезный метод readline позволяет построчно считывать информацию из текстового файла. Например:

```
file = open("my_file.txt", encoding="utf-8")

s = file.readline()
print(s)
```

На экране увидим первую строчку:

```
— Скажи-ка, дядя, ведь не даром
```

Здесь концом строки считается символ переноса ‘\n’, либо конец файла. Причем этот спецсимвол переноса будет также присутствовать в строке. В этом легко убедиться если применить к строке функцию repr, которая показывает все специальные символы в строке:

```
print(repr(s))    # '— Скажи-ка, дядя, ведь не даром\n'
```

Соответственно, если несколько раз вызвать метод readline, то каждый раз будет читаться новая строка. Например:

```
file = open("my_file.txt", encoding="utf-8")

print( file.readline() )
print( file.readline() )
print( file.readline() )
```

На экране увидим:

```
— Скажи-ка, дядя, ведь не даром

Москва, спаленная пожаром,

Французу отдана?
```

Здесь пустые строки появляются из-за наличия символа ‘\n’ в конце каждой строки.

Если требуется последовательно прочитать все строчки из файла, то для этого можно воспользоваться циклом for, применив его к файловому объекту file,

следующим образом:

```
file = open("my_file.txt", encoding="utf-8")

for line in file:
    print( line, end="" )
```

Этот пример показывает, что объект file является итерируемым и на каждой итерации возвращает очередную строку.

Чтобы прочитать все строчки из файла, можно воспользоваться методом:

```
s = file.readlines()
```

Получим список из строк:

```
['— Скажи-ка, дядя, ведь не даром\n', 'Москва, спаленная пожаром,\n',
'Французу отдана?\n', 'Ведь были ж схватки боевые,\n', 'Да, говорят, еще
какие!\n', 'Недаром помнит вся Россия\n', 'Про день Бородина!']
```

Однако этот метод следует использовать с осторожностью, т.к. для больших файлов могут получаться очень длинные списки, которым потребуется много памяти устройства, а в некоторых случаях даже не хватить.

Итак, мы с вами рассмотрели основные методы считывания информации из текстового файла. Осталось только добавить, что, как только мы завершили работу с файлом, его следует закрыть (закрыть файловый поток). Для этого используется метод close:

```
file.close()
```

Конечно, прописывая эту строчку, мы не увидим никакой разницы в работе программы. Но, во-первых, закрывая файл, происходит освобождение памяти, которая была связана с этим файловым потоком и, во-вторых, не возникнет проблем с потерей данных при их записи в файл. Поэтому всегда следует придерживаться простого правила: после завершения работы с файлом, его следует закрыть.

9.2. Обработка исключения FileNotFoundError. Менеджер контекста

Прежде чем идти дальше и рассматривать другие файловые методы, затронем важный вопрос обработки ошибки FileNotFoundError при попытке открыть несуществующий файл. Избежать ее невозможно, так как программист наперед не может знать, в каких условиях будет работать программа и вполне возможна ситуация, когда ранее доступный файл становится недоступным, например, из-за случайного его удаления, или изменении имени и прав доступа. То есть, при работе с файлами нужно уметь обрабатывать исключение FileNotFoundError, чтобы программа продолжала работать, даже если файл недоступен.

Для обработки подобных ошибок (или, как говорят, исключений) существует специальная группа операторов:

о которых мы подробно еще будем говорить в части объектно-ориентированного программирования на Python. Но, несмотря на то, что это выходит за рамки базового курса, давайте посмотрим, как использовать эти операторы при работе с файлами. Иначе, ваши программы будут заведомо содержать серьезную уязвимость при обращении к файлам.

Формально, операторы try / except / finally имеют, следующий синтаксис:

```
try:
        блок операторов
        критического кода
except [исключение]:
        блок операторов
        обработки исключения
finally:
        блок операторов
        всегда исполняемых
        вне зависимости от
        возникновения исключения
```

При работе с файлами ошибку FileNotFoundError генерирует функция open, а значит, ее вызов следует поместить в блок try, равно, как и методы чтения данных из файла. Получим следующий текст программы:

```
try:
    file = open("my_file.txt", encoding='utf-8')
    s = file.readlines()
    print(s)
    file.close()
except FileNotFoundError:
    print("Невозможно открыть файл")
```

Смотрите, здесь функция open находится внутри блока try, поэтому, если возникнет исключение FileNotFoundError, то выполнение программы перейдет в блок except и отобразится строка «Невозможно открыть файл». Иначе, будут читаться все строки из файла с последующим их отображением. В конце обязательно файл закрывается методом close.

Однако и в такой реализации не все гладко. В момент считывания информации из файла (в методе readlines) также может возникнуть исключение из-за невозможности получения данных из файла. Поэтому, совсем надежно, можно было бы реализовать эту программу следующим образом:

```
try:
    file = open("my_file.txt", encoding='utf-8')

    try:
        s = file.readline()
        print(s)
```

```
    finally:
        file.close()

except FileNotFoundError:
     print("Невозможно открыть файл")
except:
    print("Ошибка при работе с файлом")
```

Здесь прописан еще один вложенный блок try, который будет учитывать все возможные исключения и при их возникновении обязательно сработает блок finally, где выполняется закрытие файла. Это важная команда, так как любой ранее открытый файл (функцией open) следует закрывать, даже при возникновении исключений. И наличие блока finally гарантирует закрытие файла, так как срабатывает при любых исходах. Однако внутренний блок try / finally не отлавливает исключений, поэтому они будут передаваться внешнему блоку try, где выполняется их обработка. В программе второй блок except записан без указания исключений. В результате, он будет реагировать на любые не обработанные ошибки, то есть, в нашем случае на любые ошибки, кроме FileNotFoundError.

9.2.1. Менеджер контекста для файлов

Не будем сейчас углубляться в работу блоков try / except / finally. Приведенного материала будет пока вполне достаточно. Вместо этого, рассмотрим менеджер контекста применительно к файлам, который заменяет собой блок try / finally с автоматическим закрытием файловых потоков.

Что такое менеджер контекста? В языке Python существует специальный оператор with, который, можно воспринимать как аналог конструкции try / finally и в случае работы с файлами записывается следующим образом:

```
try:
    with open("my_file.txt", encoding='utf-8') as file:
        s = file.readlines()
        print(s)

except FileNotFoundError:
     print("Невозможно открыть файл")
except:
    print("Ошибка при работе с файлом")
```

Смотрите, здесь функция open вызывается внутри оператора with, а переменная на файловый объект формируется с помощью ключевого слова as. Далее, через переменную file можно вызывать любые файловые методы. Разумеется, все вызовы должны быть записаны внутри менеджера контекста, так как после его завершения файл закрывается автоматически. Именно поэтому здесь нет явного вызова метода close.

При работе с менеджером контекста следует иметь в виду, что он по умолчанию не обрабатывает никаких ошибок (исключений). Все они передаются

вышестоящему блоку try, где и отлавливаются. Но в любом случае, произошли ошибки или нет, файл будет автоматически закрыт. В этом легко убедиться, если в блоке finally вывести флаг закрытия файла (атрибут closed):

```
try:
    with open("my_file.txt", encoding='utf-8') as file:
        s = file.readlines()
        print(s)

except FileNotFoundError:
     print("Невозможно открыть файл")
except:
    print("Ошибка при работе с файлом")
finally:
    print(file.closed)
```

После запуска программы увидим значение True, то есть, файл был закрыт. Даже если произойдет критическая ошибка то, файл все равно будет закрыт. В этом удобство использования менеджера контекста with при работе с файлами.

9.3. Запись данных в текстовый файл

Продолжаем рассмотрение файловых методов и на очереди методы записи данных в файл.

Конечно, первое, что нужно сделать – это открыть файл на запись. Напомню, что по умолчанию он открывается на чтение в режиме "r" (первая буква слова read – чтение). Чтобы открыть файл на запись необходимо вторым аргументом функции open указать "w" (первая буква слова write – запись):

```
file = open("out.txt", "w")
```

Для того чтобы поместить в открытый файл какую-либо текстовую информацию можно воспользоваться методом write, например, так:

```
file.write("Hello World!")
file.close()
```

После запуска программы будет создан файл «out.txt» (если его ранее не было) со строкой «Hello World!». Причем этот файл будет располагаться в том же каталоге, что и исполняемый файл. Также после записи всей нужной информации в файл, его обязательно следует закрыть! Иначе часть информации может не сохраниться. Как мы уже отмечали, метод close нужно вызывать всегда после завершения работы с файлом.

Давайте еще раз запустим эту же программу и посмотрим, как изменится содержимое файла «out.txt». Оно никак не поменялось. Это означает, что при открытии файла на запись, прежние данные удаляются и снова записывается та же строка. Это очень важный момент, который всегда следует иметь в виду – открывая файл на запись в режиме "w", его прежнее содержимое автоматически

удаляется!

Конечно, при открытии файла на запись и работы с ним могут возникать различные файловые ошибки, поэтому здесь также имеет смысл применять оператор try/except совместно с менеджером контекста, который автоматически закроет файловый поток:

```
try:
    with open("out.txt", "w") as file:
        file.write("Hello World!")
except:
    print("Ошибка при работе с файлом")
```

Об этом всегда следует помнить, при работе с файлами, чтобы программа работала надежно даже в нештатных ситуациях.

Давайте теперь посмотрим, что будет если вызвать метод write несколько раз подряд для одного и того же открытого файла:

```
try:
    with open("out.txt", "w") as file:
        file.write("Hello1")
        file.write("Hello2")
        file.write("Hello3")
except:
    print("Ошибка при работе с файлом")
```

После запуска видим, что в файле появились три строчки, идущие друг за другом. То есть, здесь, как и при чтении, объект file записывает информацию, используя файловую позицию, и автоматически перемещает ее при записи новых данных. Поэтому все эти строки в файле следуют друг за другом.

Если требуется записывать строчки в столбик, то в конце каждой из них достаточно прописать символ переноса строки:

```
try:
    with open("out.txt", "w") as file:
        file.write("Hello1\n")
        file.write("Hello2\n")
        file.write("Hello3\n")
except:
    print("Ошибка при работе с файлом")
```

И так можно записывать в файл произвольный объем данных (пока хватает размера внешнего хранилища). Однако если требуется в уже существующий файл с данными добавить (дописать) какую-либо информацию, то для этого существует еще один режим доступа, определяемый аргументом "a" (первая буква от слова append – добавление). Если ее прописать в вышеприведенной программе:

```
try:
    with open("out.txt", "a") as file:
        file.write("Hello1\n")
```

```
        file.write("Hello2\n")
        file.write("Hello3\n")
except:
    print("Ошибка при работе с файлом")
```

то после запуска, в файл будут добавлены еще три строки, и всего получится шесть строк.

Конечно, когда файл открывается на запись в режиме "w" или дозапись в режиме "a", то читать информацию из него уже не получится. Например, если в программе вызвать метод read:

```
try:
    with open("out.txt", "a") as file:
        data = file.read()        # io.UnsupportedOperation: not readable
        file.write("Hello1\n")
        file.write("Hello2\n")
        file.write("Hello3\n")
except:
    print("Ошибка при работе с файлом")
```

то возникнет ошибка, которая автоматически будет обработана в блоке except. Но если режим доступа к файлу несколько модифицировать и записать его с плюсом "a+" или "w+", то появится возможность как записывать данные, так и считывать их. Например:

```
try:
    with open("out.txt", "a+") as file:
        file.seek(0)
        data = file.read()
        print(data)

        file.write("Hello1\n")
        file.write("Hello2\n")
        file.write("Hello3\n")
except:
    print("Ошибка при работе с файлом")
```

Обратите внимание, что при открытии файла в режиме записи его файловый указатель устанавливается на последнюю позицию. Поэтому для считывания уже существующих в нем данных, указатель следует переместить в самое начало методом `seek(0)`. А вот запись данных всегда осуществляется в конец файла, так как для записи используется другая файловая позиция и она указывает на конец файла.

Конечно, помимо метода write, существуют и другие методы записи данных. Они аналогичны ранее рассмотренным методам чтения. Например, для записи сразу нескольких строк удобно использовать метод writelines. Например:

```
try:
    with open("out.txt", "w") as file:
        file.writelines(["Hello1\n", "Hello2\n", "Hello3\n"])
```

```
except:
    print("Ошибка при работе с файлом")
```

Данный метод часто используется, когда в процессе обработки текста получается список из строк и его требуется записать в файл.

9.4. Чтение и запись в бинарном режиме доступа

До сих пор мы с вами говорили о чтении и записи информации в текстовом режиме доступа. Однако есть еще один – бинарный режим работы с файлами. Что это за режим? Это когда данные из файла считываются один в один без какой-либо обработки и пропусков. Например, изображения, аудио, видео, таблицы excel и многое другое следует записывать и считывать именно в таком режиме, так как целостность их данных гарантируется только при бинарном режиме доступа к файлам.

Чтобы открыть какой-либо файл в бинарном режиме на чтение или запись, в конце добавляется символ ‘b’. Например, чтобы открыть файл на запись в таком режиме, прописывается "wb" или "ab". Соответственно, для чтения – строка "rb".

В самом простом случае в этом режиме можно открыть какой-либо файл изображения следующим образом:

```
try:
    with open("panda_ny.jpg", "rb") as file:
        data = file.read()
        print(len(data))
except FileNotFoundError:
    print("Файл не найден.")
```

В этом примере файл «panda_ny.jpg» занимает 53448 байт и это число отображается на экране:

```
53448
```

Если в приведенной программе попытаться прочитать файл в текстовом режиме, указав режим "r", то получим ошибку:

```
UnicodeDecodeError: 'charmap' codec can't decode byte ...
```

Она связана с тем, что в файле встретился символ, который невозможно декодировать в соответствии с таблицей Unicode. В бинарном же режиме доступа никакого декодирования не выполняется и метод read возвращает обычную байтовую строку:

```
b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x00\x00\x01\x00\ ...'
```

При записи данных в бинарном режиме все то же самое. Если попытаться поместить в него обычную строку:

```
try:
    with open("panda.dat", "wb") as file:
```

```
        data = "hello"
        file.write(data)

except FileNotFoundError:
    print("Файл не найден.")
```

то возникнет ошибка:

```
TypeError: a bytes-like object is required, not 'str'
```

говорящая, что ожидается байтовая строка, а не обычная. Байтовые строки в Python обозначаются с символом ‘b’ в ее начале, например, так:

```
try:
    with open("panda.dat", "wb") as file:
        data = b"hello"
        file.write(data)

except FileNotFoundError:
    print("Файл не найден.")
```

В этом случае никаких ошибок записи не возникнет, и в файл запишутся указанные символы. То есть, в бинарном режиме доступа чтение и запись осуществляется исключительно на уровне байтовых строк (наборов байт).

9.5. Модуль pickle

Нередко бинарный режим доступа используется для сохранения и считывания программных объектов, таких как списки, кортежи, словари и т. д. Для этого они предварительно переводятся в байтовые строки, а при чтении, наоборот, из байтовых строк в исходные объекты языка Python. Этот процесс называется **сериализацией** и **десериализацией** объектов. Осуществляется он с помощью функций модуля pickle стандартной библиотеки. Давайте посмотрим на конкретных примерах, как это происходит.

9.5.1. Функции pickle.dumps() и pickle.loads()

Пусть в программе определены следующие объекты с набором данных:

```
d1 = (5, 10, -2)
d2 = [8, 4, "hello", {1, 2, 3}]
d3 = {'key1': -1, 'key2': 2}
```

Тогда, с помощью функции pickle.dumps, можно получить их представления в виде байтовых строк:

```
d1_bytes = pickle.dumps(d1)
d2_bytes = pickle.dumps(d2)
d3_bytes = pickle.dumps(d3)
```

Например, для кортежа d1, имеем:

```
b'\x80\x04\x95\x0c\x00\x00\x00\x00\x00\x00\x00K\x05K\nJ\xfe\xff\xff\xff\x87
```

```
\x94.'
```

Как видите, функция dumps работает с произвольными типами данных. И на практике сериализировать можно практически любой объект, за некоторыми исключениями. Например, нельзя сериализовать дескрипторы файлов, сокеты, подключения к базам данных и другие, состояния которых определяется в процессе выполнения программы.

Обратное преобразование набора байт в исходный объект можно выполнить с помощью функции pickle.loads. Например:

```
ld1 = pickle.loads(d1_bytes)
ld2 = pickle.loads(d2_bytes)
ld3 = pickle.loads(d3_bytes)

print(ld1, ld2, ld3, sep="\n")
```

Увидим отображение восстановленных объектов:

```
(5, 10, -2)
[8, 4, 'hello', {1, 2, 3}]
{'key1': -1, 'key2': 2}
```

То есть, это снова будут кортежи, списки, словари, строки и т.д. Причем объекты ld1, ld2, ld3 – это новые созданные объекты из соответствующих байтовых последовательностей. В этом легко убедиться, выполнив команду:

```
print(ld1 is d1, ld2 is d2, ld3 is d3)  # False False False
```

Но состояния (данные) этих объектов будут равны:

```
print(ld1 == d1, ld2 == d2, ld3 == d3)  # True True True
```

У вас может возникнуть вопрос, откуда модуль pickle «знает», как выполнять сериализацию объектов самых разных типов? В действительности, он лишь использует универсальный протокол преобразования текущего состояния объекта в набор байт, а сам алгоритм сериализации прописан отдельно для каждого типа данных.

9.6.1. Функции pickle.dump() и pickle.load()

Конечно, довольно часто сериализованные данные необходимо помещать в какой-либо поток, например, в файл или передавать по сети. Для этого удобно использовать другую пару функций:

pickle.dump() и pickle.load()

Для примера запишем объекты d1, d2, d3 в файл в бинарном режиме, получим:

```
with open('myfile.pkl', 'wb') as f:
    pickle.dump(d1, f)
    pickle.dump(d2, f)
```

```
    pickle.dump(d3, f)
```

Если теперь прочитать этот файл также в бинарном режиме:

```
with open('myfile.pkl', 'rb') as f:
    data = f.read()
    print(data)
```

то увидим результат сериализации объектов, байтовую строку:

```
b'\x80\x04\x95\x0c\x00\x00\x00\x00\x00\x00\x00K\x05K\nJ\xfe\xff\xff\xff\x87
\x94.\x80\x04\x95\x1b\x00\x00\x00\x00\x00\x00\x00]\x94(K\x08K\x04\x8c\x05he
llo\x94\x8f\x94(K\x01K\x02K\x03\x90e.\x80\x04\x95\x1a\x00\x00\x00\x00\x00\x
00\x00}\x94(\x8c\x04key1\x94J\xff\xff\xff\xff\x8c\x04key2\x94K\x02u.'
```

Для получения исходных объектов данные следует читать функцией load, например, так:

```
with open('myfile.pkl', 'rb') as f:
    ld1 = pickle.load(f)
    print(ld1)
```

После запуска программы, увидим:

```
(5, 10, -2)
```

То есть, функция load прочитала первую порцию данных, относящихся к объекту d1, и восстановила его в виде кортежа. В результате получили копию, кортеж ld1.

Чтобы прочитать все данные, достаточно несколько раз вызвать функцию load:

```
with open('myfile.pkl', 'rb') as f:
    ld1 = pickle.load(f)
    ld2 = pickle.load(f)
    ld3 = pickle.load(f)
    print(ld1, ld2, ld3, sep='\n')
```

В консоли увидим содержимое восстановленных объектов:

```
(5, 10, -2)
[8, 4, 'hello', {1, 2, 3}]
{'key1': -1, 'key2': 2}
```

Обратите внимание, функция load «знает», где завершить чтение порции байтовых данных для каждого объекта. Поэтому сериализованные данные можно последовательно записывать в поток, а затем последовательно считывать.

Будьте осторожны! Модуль pickle, несмотря на его удобство, следует использовать в своих программах с большой осторожностью, т.к. имеет потенциальную уязвимость. Дело в том, что при десериализации объектов происходит обращение к определенным методам этих объектов и их выполнение. В штатной ситуации они выполняют реконструкцию объекта. Но никто не мешает заложить в них вредоносный код, который автоматически выполнится при десериализации загруженной байтовой строки. Поэтому, если на программном

уровне нет возможности контролировать процесс воссоздания объектов, то лучше использовать другие способы сохранения данных, такие как JSON, CSV и некоторые другие.

Более подробно файловые операции разбираются в курсе «Стандартная библиотека Python»:

https://stepik.org/a/259466

В ее рамках дополнительно рассматриваются темы:

- Модуль os. Функции работы с файлами и каталогами.
- Файловые функции модуля os.path.
- Способы сканирования каталогов. Модули glob и fnmatch.
- Функции работы с файлами модуля shutil. Работа с архивами.
- Потоки io.BytesIO, io.StringIO и io.TextIOWrapper.
- Модули json, csv, shelve.

ГЛАВА X. НЕКОТОРЫЕ ПОЛЕЗНЫЕ ФУНКЦИИ ЯЗЫКА PYTHON

Язык Python содержит множество встроенных функций. С некоторыми из них мы с вами уже знакомы. Это, например, функции round, abs, len, sum, min, max, input, print, id и некоторые другие. В этой главе рассмотрим еще несколько распространенных функций map, filter, zip, all, any, sorted, опираясь только на текущие знания языка Python.

10.1. Функции type() и isinstance()

С функцией type мы с вами уже немного знакомы. Она возвращает тип данных указанного объекта. Например, пусть имеются следующие объявления различных типов данных:

```
vector = [10, 40, 20]
start = ('begin', 'start')
digit = 5
digit_f = 7.8
s = "It's a string"
```

Применяя к ним функцию type, получим результаты:

```
type(vector)   # <class 'list'>
type(start)    # <class 'tuple'>
type(digit)    # <class 'int'>
type(digit_f)  # <class 'float'>
type(s)        # <class 'str'>
```

Всюду видим обозначение ‘class’, которое говорит о том, что все объекты являются классами. О классах речь пойдет во второй части этого пособия, здесь достаточно воспринимать их как аналог слова «тип данных».

В чем может быть практическая польза функции type? Предположим, необходимо реализовать функцию, которая бы выделяла только целые числа из переданной коллекции, и возвращала полученные данные в виде списка:

```
def get_digits(data):
    res = []
    for x in data:
        if type(x) == int:
            res.append(x)

    return res
```

Как видите, здесь с помощью функции type проверяется тип каждого элемента коллекции data, и если тип равен int, то элемент добавляется в результирующий список res.

Вызовем эту функцию следующим образом:

```
a = [1, 2, True, '5', 0.5, 'abs', -6]
res = get_digits(a)
```

```
print(res)
```

После запуска программы, увидим результат:

```
[1, 2, -6]
```

Действительно, были выделены только целые числа типа int.

Теперь расширим диапазон выделяемых элементов и будем помещать в итоговый список элементы типа int и float, то есть, целочисленные и вещественные. Проверка типов в этом случае может быть записана так:

```
def get_digits(data):
    res = []
    for x in data:
        if type(x) == int or type(x) == float:
            res.append(x)

    return res
```

Однако часто в подобных случаях используют другую конструкцию проверки типов, а именно, такую:

```
def get_digits(data):
    res = []
    for x in data:
        if type(x) in (int, float):   # проверка типа int или float
            res.append(x)

    return res
```

Это намного удобнее, нагляднее и короче. Все типы данных, прописанные в кортеже, будут выбираться для добавления элемента в список res.

Здесь следует иметь в виду, что функция type проверяет строгое соответствие заданному типу. Например, булевый тип данных (значения True/False), в действительности образованы от целочисленного типа int. Тем не менее, функция type четко их различает между собой. В этом ее особенность, и в ряде случаев, преимущество перед другим способом проверки типов, которая реализуется с помощью функции isinstance.

Функция isinstance выполняет не такую строгую проверку на соответствие объекта определенному типу данных, как это делает функция type, и имеет следующий синтаксис:

isinstance(<переменная>, <тип данных>)

Она возвращает True, если переменная (объект) соответствует указанному типу данных, и False в противном случае. Давайте рассмотрим особенность ее работы.

Пусть, например, в программе объявлена булева переменная:

```
b = True
```

Тогда вызов функции:

```
result = isinstance(b, bool)  # True
```

вернет значение True, т.к. переменная b действительно соответствует типу bool. Однако если вместо bool записать int:

```
result = isinstance(b, int)   # True
```

то тоже получим True. Это связано с тем, что тип bool в языке Python образован от типа int и эту взаимосвязь показывает функция isinstance. Напомню, что строгую проверку типов можно выполнить с помощью только что рассмотренной функции type. Во втором случае она вернет False.

Если же вместо int в функции isinstance прописать любой другой тип данных, например, float (тоже числовой):

```
result = isinstance(b, float)   # False
```

то получим уже значение False, т.к. тип bool никак не связан с вещественным типом float. Этот момент следует иметь в виду при использовании функции isinstance. В ряде случаев, действительно, требуется делать проверку с учетом взаимосвязи типов и тогда используется isinstance. Если же нужна строгая проверка на конкретные типы данных, то следует использовать функцию type.

С помощью функции isinstance также можно делать проверки сразу на несколько типов данных. Предположим, нужно определить относится ли число к целому или вещественному типу данных. Для этого достаточно записать кортеж с проверяемыми типами int и float, например, следующим образом:

```
a = 5.5
res = isinstance(a, (int, float))  # True
```

Эта команда эквивалентна записи:

```
res = isinstance(a, int) or isinstance(a, float)
```

Но первый вариант короче и удобнее в использовании.

10.2. Функция map()

Следующая очень распространенная и весьма полезная функция map позволяет применять указанную функцию к элементам итерируемой последовательности. Проще всего понять принцип ее работы на конкретном примере. Предположим, необходимо следующий список, состоящий из строк:

```
digits = ['1', '2', '3', '5', '7']
```

преобразовать в аналогичный список, но состоящий из чисел:

```
result = [1, 2, 3, 5, 7]
```

По сути, для этого нужно к каждому элементу списка digits применить функцию int и на выходе сформировать требуемую последовательность. Как раз для этого предназначена функция map. Для решения поставленной задачи ее следует записать в виде:

```
b = map(int, digits)
```

Первым аргументом передается ссылка на функцию int, которая будет последовательно применяться к каждому элементу списка digits, а вторым аргументом – непосредственно список из строк. Конечно, вторым аргументом можно записывать любой итерируемый объект, но в нашем примере – это обычный список. В результате функция map возвращает итератор. То есть, переменная b – это объект, который можно перебирать функцией next:

```
<map object at 0x000002111B3BE140>
```

Давайте применим функцию next к объекту b следующим образом:

```
print(next(b))
print(next(b))
```

В консоли увидим первые два преобразованных значения:

```
1
2
```

Или же, все элементы итератора b можно перебрать циклом for:

```
for x in b:
    print(x, end=" ")
```

Получим строку из целых чисел:

```
1 2 3 5 7
```

Но нам бы хотелось получить список из целых значений, а не просто делать их перебор. Для этого можно воспользоваться уже знакомой функцией list следующим образом:

```
result = list(b)
```

Или, объединяя две команды, получаем следующую конструкцию:

```
result = list(map(int, digits))
```

В итоге переменная result будет ссылаться на список, состоящий из целых чисел:

```
[1, 2, 3, 5, 7]
```

В этих примерах функция list автоматически перебрала итератор, неявно вызывая функцию next, и сформировала соответствующие значения списка.

Конечно, этот же результат можно было бы получить, используя генератор

списка, следующим образом:

```
a = [int(x) for x in ['1', '2', '3', '5', '7']]
```

Но в этом случае все значения сразу сохраняются в памяти в виде списка, тогда как функция map возвращает итератор, и значения формируются по одному в процессе вызова функции next.

А вот эквивалентный генератор:

```
a = (int(x) for x in ['1', '2', '3', '5', '7'])
```

дает практически тот же самый результат, что и функция map. Но каждый раз записывать такую конструкцию не очень удобно, гораздо проще это делать с помощью функции map. В этом ее основное предназначение – упростить написание программного кода.

Помимо функции list к итератору b можно применять другие функции, которые в качестве аргумента ожидают итерированный объект, например:

```
res = sum(b)   # res = 18
```

Но, если следом попытаться перебрать итератор b еще раз:

```
res2 = sum(b)  # res2 = 0
```

то получим значение 0, так как итератор перебирется только один раз. Это всегда следует помнить при использовании функции map.

Также с итератором b можно использовать функции max и min, например:

```
print(max(b))    # 7
```

или (вместо предыдущей функции, чтобы дважды не перебирать b):

```
print(min(b))    # 1
```

По аналогии работают и другие функции, которые применимы к итерируемым последовательностям.

Конечно, вместо функции int в функции map можно указать любую другую, которая принимает один аргумент и возвращает некоторое значение. Например, применим к списку городов:

```
cities = ["Москва", "Астрахань", "Самара", "Уфа", "Смоленск", "Тверь"]
```

функцию len, получим:

```
b = map(len, cities)
print(list(b))
```

На экране увидим список из длин строк:

```
[6, 9, 6, 3, 8, 5]
```

Также можно применять к строкам их методы. Например, переведем все элементы списка cities в верхний регистр:

```
b = map(str.upper, cities)
```

Получим результат:

```
['МОСКВА', 'АСТРАХАНЬ', 'САМАРА', 'УФА', 'СМОЛЕНСК', 'ТВЕРЬ']
```

Обратите внимание, как указан метод upper. Сначала записан класс str и через точку его метод upper, который возвращает новую строку со всеми заглавными буквами.

Также в функции map можно указывать свои собственные функции. Главное, чтобы функция принимала один аргумент и возвращала определенное значение. Например, объявим функцию symbols, которая принимает строку и на ее основе формируется список из символов:

```
def symbols(s):
    return list(s.lower())
```

Запишем ее в функции map:

```
b = map(symbols, cities)
```

(Обратите внимание, функция записывается без круглых скобок, так как передается ссылка на нее). После запуска программы увидим наборы вложенных списков:

```
[['м', 'о', 'с', 'к', 'в', 'а'], ['а', 'с', 'т', 'р', 'а', 'х', 'а', 'н',
'ь'], ['с', 'а', 'м', 'а', 'р', 'а'], ['у', 'ф', 'а'], ['с', 'м', 'о', 'л',
'е', 'н', 'с', 'к'], ['т', 'в', 'е', 'р', 'ь']]
```

Конечно, если функция выполняет какую-либо простую операцию, то часто прописывают лямбда-функцию непосредственно в map. В нашем случае это можно сделать так:

```
b = map(lambda s: list(s.lower()), cities)
```

Результат будет прежним, а программа стала более простой и понятной.

Или можно преобразовать строки, записав их символы в обратном порядке:

```
b = map(lambda s: s[::-1], cities)
```

Получим:

```
['авксоМ', 'ьнахартсА', 'арамаС', 'афУ', 'кснеломС', 'ьревТ']
```

Довольно часто функцию map используют для обработки вводимых с клавиатуры данных. Например, если пользователь вводит в одну строчку через пробел целые числа, то для получения списка введенных значений удобно воспользоваться функцией map следующим образом:

```
data_map = map(int, input().split())
data = list(data_map)
```

Здесь конструкция `input().split()` возвращает список из строк введенных чисел, а затем, к каждой строке применяется функция int. После этого функция list преобразует генератор data_map в список из чисел.

Также часто можно встретить запись вида:

```
x, y = map(float, input().split())
```

Она предполагает ввод в одну строку через пробел двух чисел, которые сразу распаковываются в переменные x и y. Это бывает очень удобно.

Наконец, если данные следуют не через пробел, а через какой-то другой символ, например, запятую, то можно записать следующую команду для их считывания:

```
data = tuple(map(float, input().split(",")))
```

Здесь вместо функции list используется аналогичная функция tuple, которая формирует кортеж из введенных значений. В частности, при вводе строки:

```
0.5, -7.8, 10,0.01
```

получим на выходе кортеж из вещественных чисел:

```
(0.5, -7.8, 10.0, 0.01)
```

10.3. Функция filter()

Следующая функция функции filter:

filter(func, *iterables)

предназначена для фильтрации (отбора) элементов указанного итерированного объекта. Эта функция, также как и map, возвращает итератор для получения итоговых значений.

Первым аргументов в filter передается ссылка на некоторую функцию func, которая должна иметь один параметр. Если возвращаемое значение func соответствует булевому True, то текущий элемент возвращается функцией filter, а иначе отбрасывается.

Пусть, например, имеется список из целых чисел:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Требуется выбрать из него только четные значения. Функция filter в этом случае может быть записана следующим образом:

```
b = filter(lambda x: x % 2 == 0, a)
print(b)
```

После запуска программы, видим, что переменная b ссылается на объект-итератор

filter:

```
<filter object at 0x0000018D9AEF9AE0>
```

который можно перебирать с помощью функции next:

```
print(next(b))   # 2
print(next(b))   # 4
```

В результате в консоли увидим первые два четных значения из списка.

Также итератор b можно перебрать (но только один раз) с помощью цикла for:

```
for x in b:
    print(x, end=" ")
```

Будут отображены все выделенные четные значения из исходного списка:

```
2 4 6 8 10
```

И, конечно же, можно сформировать новый список с помощью функции list (либо кортеж (tuple), либо множество (set), либо другую коллекцию данных):

```
c = list(b)   # [2, 4, 6, 8, 10]
```

Иногда алгоритм обработки текущего значения может быть нетривиальным и записать его в виде лямбда-функции довольно сложно. Например, проверить является ли число простым или нет. Напомню, что простым называется любое натуральное число (начиная от 2), которое делится только на себя и на единицу. Такую проверку удобнее вынести в отдельную функцию:

```
def is_prime(n):
    if n < 2:       # Числа меньше 2 не являются простыми
        return False

    if n == 2:      # Число 2 - простое число
        return True

    if n % 2 == 0:  # Чётные числа больше 2 не являются простыми
        return False

    # Проверяем остальные числа на простоту
    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2

    return True
```

А затем указать ее в filter:

```
b = filter(is_prost, a)
```

```
print(tuple(b))
```

На выходе получим кортеж из простых чисел, выделенных из исходного списка:

```
(2, 3, 5, 7)
```

Функцию filter можно применять с любыми типами данных, например, со строками. Пусть имеется кортеж с названиями городов:

```
cities = ("Москва", "Рязань1", "Смоленск", "Тверь2", "Томск")
```

Требуется выделить только те, которые записаны буквами (без цифр и других символов). Для этого можно воспользоваться стандартным строковым методом isalpha следующим образом:

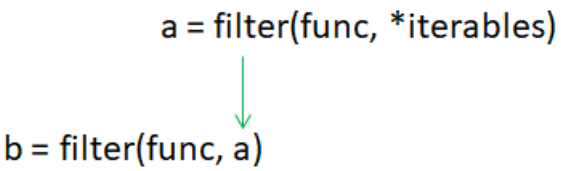
```
b = filter(str.isalpha, cities)

for x in b:
    print(x)
```

В результате получим:

```
Москва Смоленск Тверь Томск
```

Так как функция filter возвращает итератор, то в качестве второго аргумента можно указывать еще один вызов функции filter. Получаем вложенные вызовы по следующей схеме:



Например, вложенная функция filter выбирает только числовые значения из коллекции, а внешняя – только четные из них. Такой алгоритм можно реализовать следующим образом:

```
data = {4, 0.5, -43, 'a', 'b', (10, 20), 10, 22, False}

digits = filter(lambda x: type(x) in (int, float), data)
result = filter(lambda x: x % 2 == 0, digits)

print(tuple(result))
```

После запуска программы увидим:

```
(4, 10, 22)
```

Здесь первая функция filter выделяет из множества data только числовые значения, т.е. те, у которых тип int или float, остальные отбрасывает. Вторая функция filter использует первую, как итератор для получения чисел, и оставляет только четные значения. В результате получаем три числа: 4, 10, 22.

10.4. Функция zip()

Третья, часто используемая функция, – это функция:

zip(iter1 [, iter2 [, iter3] ...])

Она перебирает переданные итерируемые объекты и на каждом шаге возвращает кортеж из значений соответствующих элементов. Перебор ограничивается самой короткой из переданных коллекций.

Чтобы все это было понятно, давайте в качестве простого примера применим функцию zip к следующим двум спискам:

```
a = [1, 2, 3, 4]
b = [5, 6, 7, 8, 9, 10]

z = zip(a, b)
print(z)
```

Так как функция zip возвращает итератор, то после запуска программы увидим соответствующий объект:

```
<zip object at 0x00000226DCC06000>
```

Для перебора его элементов, как всегда, следует воспользоваться функцией next:

```
print(next(z))   # (1, 5)
print(next(z))   # (2, 6)
```

Видим, что значения первого кортежа – это первые значения списков a и b. Значения второго кортежа – вторые значения элементов тех же исходных списков. И так далее, пока не доходим до конца первого списка a, так как он самый короткий.

Давайте переберем все пары итератора z через цикл for:

```
for x in z:
    print(x)
```

Получим на выходе следующие кортежи:

```
(1, 5)
(2, 6)
(3, 7)
(4, 8)
```

Последний содержит числа 4 и 8. Процесс остановился при достижении самого короткого списка a. Обратите внимание, что итератор, возвращенный функцией zip, можно перебрать только один раз. Например, повторный вызов оператора цикла for:

```
for x in z:
    print(x)
```

не сработает ни разу, т.к. итератор z ранее уже полностью перебирался.

Как всегда, элементы, возвращаемые функцией zip, удобно помещать в упорядоченные коллекции, например, в кортеж:

```
result = tuple(zip(a, b))
```

А затем обходить несчетное число раз. Конечно, в этом случае увеличивается расход памяти, так элементы хранятся в кортеже, а не генерируются «на лету» с помощью итератора. Но для коротких последовательностей это не так принципиально.

Функция zip может работать с любыми итерируемыми (перебираемыми) объектами, в том числе, и со строками. Например:

```
a = [1, 2, 3, 4]
b = {5, 6, 7, 8, 9, 10}
c = "python"

res = tuple(zip(a, b, c))
print(res)
```

В консоли увидим:

```
((1, 5, 'p'), (2, 6, 'y'), (3, 7, 't'), (4, 8, 'h'))
```

Напомню, что в операторе for можно сразу выполнять распаковку коллекций в отдельные переменные, например, следующим образом:

```
for v1, v2, v3 in zip(a, b, c):
    print(v1, v2, v3)
```

Также все кортежи из итератора можно сразу извлечь с помощью оператора присваивания с распаковкой следующим образом:

```
v1, v2, v3, v4 = zip(a, b, c)
print(v1, v2, v3, v4)
```

Получим:

```
(1, 5, 'p') (2, 6, 'y') (3, 7, 't') (4, 8, 'h')
```

Здесь неявно вызывается функция next для получения каждого следующего кортежа с присвоением их отдельным переменным.

Конечно, на практике такую команду лучше не использовать, т.к. при изменении числа возвращаемых кортежей получим ошибку из-за несоответствия числа переменных числу возвращаемых данных. В универсальном виде предыдущую команду можно записать следующим образом:

```
z = zip(a, b, c)
lz = list(z)
```

Если распаковать этот список:

```
print(*lz)
```

то получим множество отдельных кортежей:

```
(1, 5, 'p') (2, 6, 'y') (3, 7, 't') (4, 8, 'h')
```

И если их снова обработать с помощью еще одной функции zip:

```
t1, t2, t3 = zip(*lz)
print(t1, t2, t3, sep="\n")
```

то в результате будут сформированы три кортежа из соответствующих элементов этих четырех:

```
(1, 2, 3, 4)
(5, 6, 7, 8)
('p', 'y', 't', 'h')
```

То есть, исходные коллекции были преобразованы в кортежи одинаковой длины и составлены из соответствующих значений элементов.

Этот пример показывает возможность использования операции распаковки «*» для передачи в функцию zip множества итерируемых объектов, сохраненных в какой-либо другой коллекции. На практике этот прием нередко используется.

10.5. Сортировка с помощью метода sort() и функции sorted()

Ранее мы с вами уже рассматривали работу метода sort для сортировки списков и функцию sorted для любых коллекций. Здесь еще раз акцентрируем внимание на особенностях и различиях в их работе, а также познакомимся с дополнительным параметром key, через который можно управлять процессом сортировки.

Я напомню, что для любого списка:

```
a = [4, 3, -10, 1, 7, 12.5]
```

элементы которого можно сравнивать на «больше» и «меньше», применим метод:

```
a.sort()
```

для сортировки этого списка. В результате получим отсортированные данные по возрастанию (неубыванию) значений:

```
[-10, 1, 3, 4, 7, 12.5]
```

Но если в этом списке окажется элемент, который нельзя сравнивать с другими элементами, например, строка:

```
a = [4, 3, -10, 1, 7, 12.5, "abc"]
```

то при вызове метода:

```
a.sort()
```

возникнет ошибка:

```
TypeError: '<' not supported between instances of 'str' and 'int'
```

говорящая, что числа и строки сравнивать между собой нельзя. Это всегда следует иметь в виду при сортировке коллекций.

Если требуется выполнить сортировку по убыванию значений, то в методе sort достаточно прописать именованный аргумент reverse со значением True:

```
a = [4, 3, -10, 1, 7, 12.5]
a.sort(reverse=True)
```

Получим результат:

```
[12.5, 7, 4, 3, 1, -10]
```

Во всех этих примерах метод sort менял непосредственно сам список. Но, что если требуется выполнить сортировку, не меняя текущую коллекцию? Например, сформировать новый отсортированный список, не меняя прежний? Для этого удобно использовать функцию sorted, которая применима к любым коллекциям, а не только спискам. Например, команда:

```
a = [4, 3, -10, 1, 7, 12.5]
res = sorted(a)
```

возвратит новый список res с отсортированными значениями исходного списка. Причем сам список a остается неизменным.

Чтобы отсортировать коллекцию по убыванию значений, также используется именованный аргумент reverse со значением True:

```
res = sorted(a, reverse=True)
```

Получим:

```
[12.5, 7, 4, 3, 1, -10]
```

Аналогичным образом с помощью функции sorted можно выполнять сортировку других коллекций, например, кортежей или строк. Предположим, имеется следующий кортеж, содержащий строки:

```
names = ("Волга", "Лена", "Дон", "Енисей")
```

Объект names не имеет метода sort. Этот метод присутствует только у списков. Поэтому команда:

```
names.sort()   # AttributeError
```

приведет к ошибке. Но функцию sorted вполне можно использовать, так как она не меняет текущий объект, а возвращает новый список:

```
res = sorted(names)
```

Получим список res с отсортированными строками:

```
['Волга', 'Дон', 'Енисей', 'Лена']
```

В этом преимущество функции sorted перед методом sort. Она позволяет выполнять сортировку любых коллекций, не только списков. В частности, с ее помощью можно выполнить сортировку символов строк. Например, команда:

```
res = sorted("python")
```

Вернет список с набором символов отсортированных по возрастанию:

```
['h', 'n', 'o', 'p', 't', 'y']
```

Или, то же самое можно сделать для словаря:

```
d = {'river': "река", 'house': "дом", 'tree': "дерево", 'road': "дорога"}
res = sorted(d)
```

На выходе получаем список res с ключами, отсортированными по возрастанию:

```
['house', 'river', 'road', 'tree']
```

Почему здесь фигурируют только ключи без их значений? Дело в том, что словарь – это итерируемый объект, который по умолчанию возвращает ключи. Например, чтобы получить список отсортированных значений словаря, то следует дополнительно вызвать метод values:

```
res = sorted(d.values())
```

Получим:

```
['дерево', 'дом', 'дорога', 'река']
```

А если прописать метод items:

```
res = sorted(d.items())
```

то получим список из отсортированных кортежей в формате «ключ-значение»:

```
[('house', 'дом'), ('river', 'река'), ('road', 'дорога'), ('tree',
'дерево')]
```

Если теперь полученный кортеж res указать в функции dict:

```
d_sort = dict(res)
```

то будет сформирован новый словарь с отсортированными по кортежам данными:

```
{'house': 'дом', 'river': 'река', 'road': 'дорога', 'tree': 'дерево'}
```

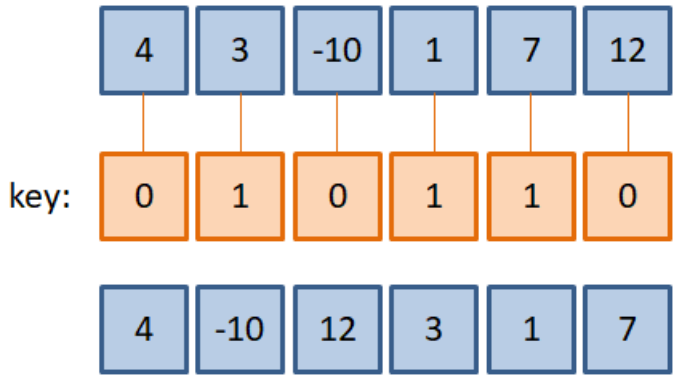
10.5.1. Параметр key метода sort() и функции sorted()

По умолчанию сортировка коллекций выполняется по значениям ее элементов.

Например:

```
lst = [4, 3, -10, 1, 7, 12]
b = sorted(lst)                  # [-10, 1, 3, 4, 7, 12]
```

Однако вместо этих значений можно использовать другие, по которым будет осуществляться сортировка исходных данных. О чем здесь речь? Предположим, в списке lst нужно в процессе сортировки расположить сначала четные значения, а затем нечетные.



Чтобы выполнить такую манипуляцию в функции sorted удобно воспользоваться параметром key, которому передается ссылка на функцию, формирующую альтернативные значения каждого текущего элемента (ключи). А затем выполнить сортировку по этим ключам, а не по исходным данным. Например, для решения поставленной задачи разделения элементов на четные и нечетные, пропишем аргумент key со следующей лямбда-функцией:

```
b = sorted(lst, key=lambda x: x % 2)
```

Параметр `x` лямбда-функции – это текущее значение сортируемого элемента коллекции. А то, что она возвращает, становится значением соответствующего ключа. То есть, для четных значений имеем 0, а для нечетных 1. Сортируя эти ключи элементов списка по возрастанию, получаем искомый результат:

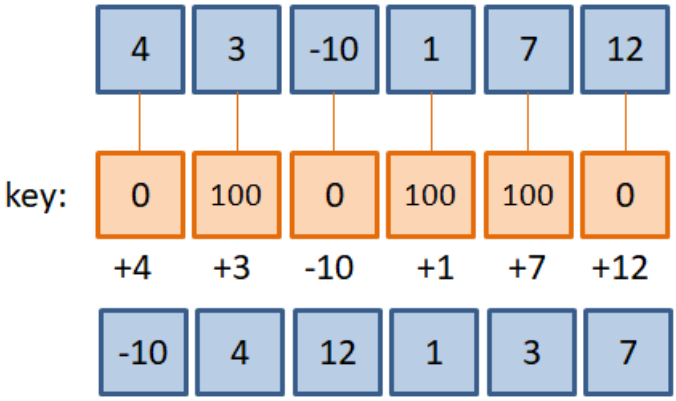
```
[4, -10, 12, 3, 1, 7]
```

Как видите, все достаточно просто. Тот же самый аргумент key можно указать и в методе sort для списка:

```
lst.sort(key=lambda x: x % 2)
```

Работает он абсолютно так же.

Давайте теперь немного усложним задачу и сделаем не просто разделение на четные и нечетные, а еще и отсортируем каждую группу чисел по возрастанию. Для этого значения ключей будем увеличивать на значения самих элементов и, кроме того, нечетным величинам дополнительно будем прибавлять число 100, чтобы гарантированно разделить четные и нечетные значения (полагаем, что в нашей коллекции числа не первышают значение 100). Это исключительно учебный пример, показывающий возможности сортировки элементов по ключам.



Получим следующую реализацию:

```
lst = [4, 3, -10, 1, 7, 12]
b = sorted(lst, key=lambda x: x if x % 2 == 0 else 100 + x)
```

Здесь в лямбда-функции использована тернарная условная операция, которая для четных величин возвращает их значение, а для нечетных – значение плюс 100. Получаем результат:

```
[-10, 4, 12, 1, 3, 7]
```

Видим, что теперь дополнительно выполняется сортировка каждой группы чисел. Вот в целом принцип работы параметра key метода sort и функции sorted.

Дополнительно рассмотрим несколько более практичных примеров использования данного параметра. Предположим, имеется список городов:

```
lst = ["Москва", "Тверь", "Смоленск", "Псков", "Рязань"]
```

которые нужно упорядочить по длине имени. Воспользуемся функцией sorted и в аргументе key укажем стандартную функцию len:

```
names_sort = sorted(lst, key=len)
```

Получим следующий результат:

```
['Тверь', 'Псков', 'Москва', 'Рязань', 'Смоленск']
```

Или можно сделать сортировку по последнему символу слова:

```
res = sorted(lst, key=lambda x: x[-1])
```

Получим результат:

```
['Москва', 'Псков', 'Смоленск', 'Тверь', 'Рязань']
```

И так далее. Также аргумент key нередко применяют для сортировки сложных структур данных. Допустим, имеется кортеж, содержащий вложенные кортежи с информацией по книгам:

```
books = (
    ("Евгений Онегин", "Пушкин А.С.", 200),
    ("Муму", "Тургенев И.С.", 250),
```

```
    ("Мастер и Маргарита", "Булгаков М.А.", 500),
    ("Мертвые души", "Гоголь Н.В.", 190)
)
```

Требуется его отсортировать по цене (последнее значение в каждом вложенном кортеже). С помощью параметра key это можно так:

```
book_sorted = sorted(books, key=lambda x: x[2])
```

На выходе получаем отсортированный по ценам список:

```
[('Мертвые души', 'Гоголь Н.В.', 190), ('Евгений Онегин', 'Пушкин А.С.',
200), ('Муму', 'Тургенев И.С.', 250), ('Мастер и Маргарита', 'Булгаков
М.А.', 500)]
```

И так через параметр key можно реализовывать самые разные алгоритмы сортировки данных.

10.6. Функции all() и any()

Следующие две распространенные функции:

- all() проверяет, что все элементы коллекции соответствуют заданному критерию;
- any() проверяет, что хотя бы один элемент коллекции соответствует заданному критерию.

10.6.1. Функция all()

Рассмотрим их работу на конкретных примерах. Предположим, в программе объявлен список lst, состоящий из булевых значений:

```
lst = [True, True, True, True]
```

Требуется определить, что все они принимают значение True. Для этого удобно воспользоваться функцией all, которая в качестве аргумента принимает итерируемый объект (в нашем примере список):

```
res = all(lst)    # res = True
```

Так как все значения списка lst равны True, то функция all возвращает True. Если хотя бы одно (любое) значение изменить на False, то функция all вернет False:

```
res = all([True, True, False, True])   # res = False
```

Функция all применима и к коллекциям, содержащим произвольные данные, например, такие:

```
a = [0, 1, 2.5, "", "python", [], [1, 2], {}]
```

Тогда, при выполнении команды:

```
res = all(a)    # res = False
```

все элементы списка будут сначала преобразованы в булевы значения (с помощью функции bool), а затем выполнена проверка, что все они должны быть равны True. Если это так, то функция all вернет True, а иначе False.

Ранее мы с вами уже говорили, что любые значения можно привести к булевому значению, используя функцию bool. В частности, все «пустое» означает False:

```
bool(0)    # False
bool("")   # False
bool([])   # False
bool({})   # False
```

А все «не пустое» приводится к True:

```
bool(1)         # True
bool("python")  # True
```

Так как список a содержит 0, пустую строку, пустой список и словарь, то в нем будут появляться значения False. Соответственно, итоговый результат работы функции all также будет равен False.

Для лучшего понимания работы функции all ее можно повторить на уровне булевых вычислений. Если вначале объявить некую переменную (в программе all_res) со значением True:

```
all_res = True
for x in a:
    all_res = all_res and bool(x)

print(all_res)
```

А затем в цикле для каждого элемента применять операцию and, то в результате переменная all_res сохранит начальное значение True только в том случае, если не встретится ни одного значения False. Так в деталях работает функция all.

10.6.2. Функция any()

Вторая функция any работает похожим образом, но возвращает True, только в том случае, если в анализируемой коллекции встретилось хотя бы одно значение True. Например, для того же списка a функция any вернет значение True:

```
a = [0, 1, 2.5, "", "python", [], [1, 2], {}]
res = any(a)    # res = True
```

А вот если передать список со всеми значениями False:

```
res = any([False, False, False, False])   # res = False
```

то получим значение False. Это единственный случай, когда функция any возвращает False, во всех остальных ситуациях получим True.

Повторить работу функции any через булевы операции можно следующим образом. Начальное значение переменной any_res приравняем False, а затем в

цикле на каждой итерации ее значение будем корректировать с помощью операции or, получим:

```
any_res = False
for x in a:
    any_res = any_res or bool(x)

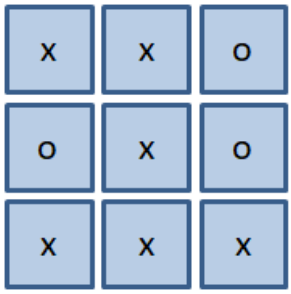
print(any_res)
```

Если в коллекции a встретится хотя бы одно значение True, то оператор or вернет True и оно сохранится в переменной any_res до конца работы цикла, какие бы данные дальше не появлялись.

10.6.3. Практический пример использования функции all()

Давайте рассмотрим более практичный пример использования функции all. Представим, что мы делаем игру «Крестики-нолики» и хотели бы на каждом шаге определять, есть ли выигрышная позиция, например, у крестиков, или нет? Для простоты сделаем это следующим образом. Все поле из девяти клеток опишем одномерным списком с некоторыми значениями, например, такими:

```
P = ['x', 'x', 'o', 'o', 'x', 'o', 'x', 'x', 'x']
```



Для проверки выигрышной ситуации по строкам, можно воспользоваться функцией all, следующим образом:

```
row_1 = all(map(lambda x: x == 'x', P[:3]))
row_2 = all(map(lambda x: x == 'x', P[3:6]))
row_3 = all(map(lambda x: x == 'x', P[6:]))
print(row_1, row_2, row_3)
```

Увидим значения:

```
False False True
```

Как это работает? Смотрите, здесь в функции all используется вложенный вызов функции map, которая формирует значение True для «крестиков», и False для «ноликов». Без использования map все величины списка P были бы преобразованы в True. Также здесь записаны срезы для выделения каждой строки, чтобы функция map применялась к ним независимо. Выходные величины функции map обрабатываются функцией all с формированием итогового результата: True – есть выигрышная комбинация; False – нет выигрышной

комбинации.

В приведенной реализации у нас получилось небольшое дублирование кода: три раза записана одна и та же анонимная функция. Это легко поправить, задав свою собственную, либо так:

```
def true_x(a):
    return a == 'x'
```

либо более простой конструкцией:

```
true_x = lambda x: x == 'x'
```

А затем укажем ее в функции map:

```
row_1 = all(map(true_x, P[:3]))
row_2 = all(map(true_x, P[3:6]))
row_3 = all(map(true_x, P[6:]))
```

По аналогии можно сделать проверку выигрыша по столбцам:

```
col_1 = all(map(true_x, P[::3]))
col_2 = all(map(true_x, P[1::3]))
col_3 = all(map(true_x, P[2::3]))
print(col_1, col_2, col_3)
```

Увидим результаты:

```
False True False
```

Для практики проверку по диагоналям попробуйте сделать самостоятельно.

10.6.4. Практический пример использования функции any()

Для второй функции any рассмотрим такой простой пример. Предположим, разрабатывается игра «Сапер», в которой игровое поле представлено в виде одномерного списка длиной N x N элементов:

```
N = 10
P = [0] * (N*N)
```

Если в списке P появляется хотя бы одна мина (отметим ее звездочкой):

```
P[4] = '*'
```

то игрок проигрывает. Чтобы узнать, наступил ли игрок на мину, удобно воспользоваться функцией any следующим образом:

```
loss = any(map(lambda x: x == '*', P))
```

После запуска программы получим значение True, то есть, игрок проиграл. А если мину убрать, то получим значение False.

ГЛАВА XI. ПОГРУЖЕНИЕ В ЧИСЛА. АННОТАЦИИ ТИПОВ. КОНСТРУКЦИЯ match/case

11.1. Форматы записи чисел

Мы с вами в программах неоднократно прописывали целые и вещественные значения в виде:

```
a = 500
b = 0.01
```

Это наиболее распространенная запись. Однако язык Python поддерживает и другие форматы записи чисел:

Формат	Префикс	Пример записи
Двоичный (binary)	0b или 0B	0b1111 (число 15)
Шестнадцатеричный (hex)	0x или 0X	0xff20 (число 65312)
Восьмеричный (octal)	0o или 0O	0O755 (число 493)
Экспоненциальный	Запись с ‘e’ или ‘E’	15e-1 (число 1.5)

О них сейчас и пойдет речь. Но вначале отмечу, что начиная с версии Python 3.6, допустимо при записи чисел использовать символ нижнего подчеркивания в качестве визуального разделителя разрядов. Например, когда записывается какое-либо большое значение:

```
value = 147356486
```

то сразу сложно определить его разряды (десятки, тысячи, миллионы и т.д.). Но если это же число переписать в виде:

```
value = 147_356_486
```

то хорошо видно, что это 147 миллионов. Причем так можно записывать и вещественные числа. Например:

```
volume = 100_543.102_453
```

И любые другие в разных форматах.

11.1.1. Экспоненциальная форма записи чисел

Давайте теперь рассмотрим возможности записи чисел в различных форматах. И начнем с экспоненциальной формы, когда числа записываются через степень десятки. Например, число:

```
a = 5e2    # 500.0
```

или, что то же самое:

```
a = 5E2    # 500.0
```

определяет десятичное значение 500. Здесь запись «e2» означает 10^2, то есть, 100 и оно умножается на 5, получаем 500. Причем, обратите внимание, число 500

представляется как вещественное, а не как целое. Экспоненциальная запись всегда формирует вещественные числа.

Если же указать отрицательную степень десятки, например, так:

```
b = 1e-2   # 0.01
```

то получим значение 0.01. Здесь запись «e-2» означает 10^{-2}, то есть, 0.01 с последующим умножением на 1. Причем единичка вначале строго обязательна – это формат записи чисел: сначала идет число, которое умножается на степень десятки.

Для чего может понадобиться такая форма записи? Часто в инженерных задачах оперируют очень большими или очень малыми числами. Например, нам нужно указать шаг сходимости градиентного алгоритма как 10^{-8}. Записывать это число в классическом виде:

```
lr = 0.00000001
```

не очень удобно, да и легко ошибиться в количестве нулей. А вот в экспоненциальной форме все удобно и наглядно:

```
lr = 1e-8
```

И то же самое с очень большими числами, например, число Авогадро (примерно $6,02 \cdot 10^{23}$):

```
avg = 6.02e23
```

Прописывать такое число в обычном (десятичном) виде было бы очень неудобно. А в экспоненциальной форме выглядит все намного лучше. Причем, обратите внимание, вначале записано вещественное значение 6.02, а затем степень десятки. Это тоже вполне допустимая запись.

11.1.2. Двоичный формат записи чисел

В аналах вычислительной техники все числа представляются в двоичном формате, то есть, составлены всего из двух символов (цифр) 1 и 0. Например, число:

1101 = 13 (в десятичной системе)

Как же числа можно описывать всего двумя цифрами и как находить их соответствия с привычной нам десятичной записью? Вкратце я напомню, что в десятичной системе счисления все цифры умножаются на степени десяток, в зависимости от позиции их написания. Например:

$$123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 = 100 + 20 + 3$$

При этом нам достаточно иметь десять различных цифр (от 0 до 9) для представления любого числа. Когда переходим к двоичной системе счисления, то

все числа кодируются двумя символами 0 и 1. Они еще называются **битами**. В результате получаем такие виды записей:

$$101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$$

$$1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13$$

$$10001101 = 1 \cdot 2^7 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 = 141$$

Здесь сразу приведен их перевод в десятичную систему счисления, учитывая, что все позиции цифр – это соответствующие степени двойки.

Чтобы в Python записывать числа сразу в двоичной системе, необходимо вначале записать префикс 0b или 0B, после которого должны следовать цифры 0 и 1. Например:

```
a = 0b001
b = 0B1101
c = 0b10001101
d = -0b1111
```

При такой записи все числа получаются целыми (тип int).

Конечно, первая реакция человека, впервые столкнувшегося с двоичным представлением чисел, зачем это нужно? Куда привычнее использовать десятичную запись. Однако бывают задачи, в которых требуется включить или выключить отдельный бит числа, или проверить является ли текущий бит равным 1 (то есть включенным) или 0 (выключенным). Здесь двоичное представление очень помогает визуализировать процесс вычислений.

11.1.3. Шестнадцатеричный формат записи чисел

Следующая часто используемая система счисления – шестнадцатеричная. Здесь уже используется 16 различных обозначений. Для этого берут десять цифр из десятичной системы и еще первые шесть букв латинского алфавита:

0-9, A, B, C, D, E, F

В результате числа принимают такой вид:

1A, FB, C2DE

Для их перевода в привычную десятичную систему, используется тот же принцип, что в двоичной и в десятичной. Например:

$$1A = 1 \cdot 16^1 + A \cdot 16^0 = 26$$

$$FB = F \cdot 16^1 + B \cdot 16^0 = 240 + 11 = 251$$

$$C2DE = C \cdot 16^3 + 2 \cdot 16^2 + D \cdot 16^1 + E \cdot 16^0 = 49886$$

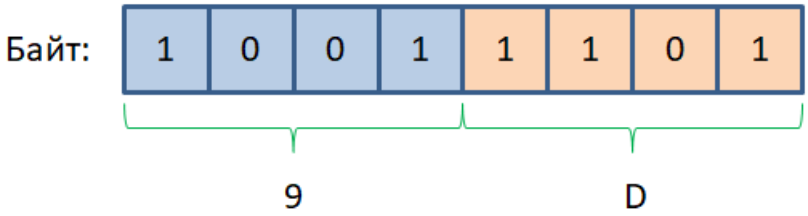
Здесь буква A соответствует десятичному значению 10, поэтому в первом случае получаем значение 26. Буква F = 15, B = 11, поэтому для второго числа получаем

значение 251. А последнее число расшифруйте самостоятельно.

В чем ценность шестнадцатеричной системы счисления? Она получила широкое распространение благодаря удобному представлению байтовых данных. Один байт – это восемь бит. Если разделить байт на две части, то получим по четыре бита. В эти четыре бита можно записать одно из шестнадцати значений, то есть, одно из значений шестнадцатеричной системы счисления:

$$0 – F$$

Получается, что два символа шестнадцатеричной системы, как раз описывают один байт. Причем делают это независимо друг от друга.



При определенной тренировке, глядя на шестнадцатеричную запись, можно легко перевести ее в десятичную и двоичную системы.

Для записи значений в шестнадцатеричном формате перед числом следует поставить префикс «0x» или «0X». Например:

```
a = 0x1A
```

Причем можно использовать не только заглавные, но и малые латинские буквы:

```
b = 0xde
c = 0xaa3f
```

Везде будут получаться целые положительные значения. Для определения отрицательных чисел достаточно вначале прописать унарный минус, например:

```
d = -0x342
```

11.1.4. Восьмеричный формат записи чисел

Последняя восьмеричная система, которую мы рассмотрим, работает по аналогии с предыдущими, только используется основание восемь. Для описания восьмеричных чисел используются первые восемь цифр (от 0 до 7). Например:

$$27 = 2 \cdot 8^1 + 7 \cdot 8^0 = 23$$

$$54 = 5 \cdot 8^1 + 4 \cdot 8^0 = 44$$

$$775 = 7 \cdot 8^2 + 7 \cdot 8^1 + 5 \cdot 8^0 = 509$$

Для записи чисел в восьмеричном формате следует использовать префикс «0o» или «0O». Например:

```
a = 0o27
```

```
b = 0o54
c = -0o775
```

После объявлений с этими числами можно выполнять все те же самые операции, что и с обычными, десятичными числами. Тем более, что все они автоматически приводятся к единому формату, принятым в вычислительной технике. Форматы записи чисел используются исключительно для их визуального определения в текстах программ. На этапе исполнения все они принимают единый, стандартный вид.

11.2. Битовые операции И, ИЛИ, НЕ, XOR

Практически все языки программирования позволяют работать с отдельными битами чисел. Что такое биты мы с вами ранее упоминали – это значения 0 и 1, которыми описывается число в двоичной системе счисления. И язык Python позволяет оперировать ими, используя битовые операции И, ИЛИ, НЕ и XOR. Давайте рассмотрим принцип каждой из них на конкретных примерах.

11.2.1. Битовая операция НЕ

Начнем с самой простой битовой операции НЕ (по-английски NOT), которая выполняет инверсию бит в соответствии со следующей таблицей истинности:

x	НЕ
0	1
1	0

Как она применяется непосредственно к числам? Пусть в программе объявлена целочисленная переменная:

```
a = 121
```

Битовое (двоичное) представление этого числа можно получить с помощью функции bin:

```
bits = bin(a)   # '0b1111001'
```

Эта функция возвращает строку с набором бит. В данном случае, получим:

```
'0b1111001'
```

Применим к этому числу битовую операцию НЕ. Для этого в Python используется операция ~ (тильда), которая записывается перед числом или переменной:

```
b = ~a
```

Получится отрицательное число. Чтобы наглядно увидеть эффект инверсии бит, преобразуем число b сначала в аналогичное положительное, а затем применим функцию bin:

```
w_bits = 8  # максимальное количество для описания числа
ub = b & ((1 << w_bits) - 1)  # преобразование в беззнаковое значение
```

```
bits = bin(ub)  # '0b10000110'
```

Видим, что операция ~ действительно выполнила инверсию исходного числа:

```
bits = bin(a)   # '0b01111001'
```

Вот принцип работы битовой операции НЕ. Мы здесь не будем вдаваться в детали кодирования целых чисел на уровне бит – это другая, отдельная тема. Главное понять, что делает операция НЕ и как она записывается на языке Python.

11.2.2. Битовая операция И

Следующая битовая операция И (по-английски AND) применяется уже к двум операндам (переменным или числам) и имеет следующую таблицу истинности:

x1	x2	И
0	0	0
0	1	0
1	0	0
1	1	1

Записывается эта операция с помощью символа & (амперсанд).

Например, пусть имеются два целых числа:

```
flags = 5
mask = 4
```

Для них битовая операция И запишется в виде:

```
res = flags & mask   # res = 4
```

В результате, переменная res будет равна 4. Почему получился такой результат? В соответствии с таблицей истинности, результирующие биты будут следующими (применяем таблицу для И независимо к каждому столбцу):

flags	0	0	0	0	0	1	0	1
mask	0	0	0	0	0	1	0	0
res	0	0	0	0	0	1	0	0

Полученный набор бит переменной res как раз соответствует десятичному значению 4.

У вас здесь может возникнуть вопрос. Зачем все это надо? В действительности существуют задачи, где требуется, например, проверять: включен какой-либо бит числа (то есть установлен в 1) или нет. Это можно сделать с помощью битовой операции И следующим образом:

```
if flags & mask == mask:
     print("Включен 2-й бит числа")
else:
     print("2-й бит выключен")
```

Здесь операция И обнулит в переменной flags все биты, кроме тех, что совпадают

в обеих переменных. Так как mask содержит только один включенный бит – второй (счет с нуля), то условие выполнится, только если в переменной flags этот бит тоже включен. Чтобы убедиться, что все работает, достаточно присвоить переменной flags значение 1:

```
flags = 1
```

и снова запустить программу. Должно появиться сообщение, что 2-й бит выключен. Это пример того, как с помощью битовой операции И можно проверять включены ли отдельные биты числа.

Другое распространенное применение битовой операции И связано с выключением (обнулением) отдельных битов. Делается это следующим образом:

```
flags = 13              # 1101
mask = 5                # 0101
flags = flags & ~mask   # 1000
```

Как работает данная конструкция? Сначала выполняется инверсия бит маски (переменная mask), так как операция НЕ имеет более высокий приоритет, чем операция И. В результате маска будет содержать биты:

```
~mask                   # 1010
```

А следующая операция «битовое И» оставит в переменной flags без изменения только те биты, которым соответствуют биты 1 в маске, а все остальные будут обнулены. За счет этого происходит выключение 2-го и 0-го битов переменной flags.

Кстати, последнюю команду можно переписать короче:

```
flags &= ~mask
```

Это будет одно и то же.

11.2.3. Битовая операция ИЛИ

Следующая битовая операция ИЛИ (по-английски OR) записывается с помощью символа «|» и имеет такую таблицу истинности:

x1	x2	ИЛИ
0	0	0
0	1	1
1	0	1
1	1	1

Обычно эту операцию применяют, когда нужно включить отдельные биты в переменной. Делается это следующим образом (пример):

```
flags = 8             # 1000
mask = 5              # 0101
flags = flags | mask  # 1101
```

Операция ИЛИ как бы собирает все единички из обеих переменных и получается итоговое значение, которое можно сравнить со своеобразным сложением (объединением).

flags	0	0	0	0	1	0	0	0
mask	0	0	0	0	0	1	0	1
flags	0	0	0	0	1	1	0	1

Кстати, в этом случае действительно получилось 8+5=13. Но так выходит далеко не всегда. Например, если:

```
flags = 9
flags |= mask
```

то результат тоже будет 13, так как операция ИЛИ включает бит вне зависимости от того, был ли он уже включен или нет.

11.2.4. Битовая операция исключающее ИЛИ (XOR)

Последняя базовая операция работы с битами – это исключающее ИЛИ (по-английски XOR). Она записывается символом «^» и имеет следующую таблицу истинности:

x1	x2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Из нее видно, что операция XOR позволяет переключать биты числа, то есть, если они были равны 0, то станут 1 и, наоборот, если были 1, то станут 0. Проверим это на следующем примере:

```
flags = 9               # 1001
mask = 1                # 0001
flags = flags ^ mask    # 1000
```

В результате такой операции переменная flags будет равна 8, так как 0-й бит будет выключен:

flags	0	0	0	0	1	0	0	1
mask	0	0	0	0	0	0	0	1
flags	0	0	0	0	1	0	0	0

Однако если повторить последнюю команду:

```
flags ^= mask           # 1001
```

то нулевой бит снова включится и переменная flags примет прежнее значение 9. Вот так, с помощью операции XOR можно переключать отдельные биты переменных.

Интересной особенностью операции XOR является отсутствие потерь данных при

ее работе. Что это значит? Смотрите, какую бы маску мы не взяли, дважды примененная маска дает исходное значение:

```
flags = 9       # 0000 1001
mask = 111      # 0110 1111
flags ^= mask   # 0110 0110
flags ^= mask   # 0000 1001
```

Этот эффект часто используют для шифрования информации. Например, так делается в архиваторе zip. В нем пароль – это маска, которая накладывается на заархивированные данные. Чтобы восстановить их, необходимо ввести пароль и повторить наложение этой же маски.

Итак, мы рассмотрели с вами основные битовые операции, которые можно записывать в виде следующих команд:

Полная форма	Краткая форма	Приоритет
a = a & b	a &= b	2 (И)
a = a | b	a |= b	1 (ИЛИ)
a = a ^ b	a ^= b	1 (XOR)
~a		3 (НЕ)

В этой таблице приоритет тем выше, чем выше значение. То есть, самый высокий (выполняется в первую очередь) у битовой операции НЕ, затем И, а операции ИЛИ и XOR имеют равный приоритет.

11.2.5. Операции смещения бит

Помимо четырех основных битовых операций язык Python имеет еще две часто используемые операции смещения бит:

- >> – смещение бит вправо;
- << – смещение бит влево.

Принцип их работы очень прост. Предположим, имеется переменная:

```
x = 160
```

Ее битовое представление:

```
bin(x) # 10100000
```

С помощью операции >> можно сдвинуть все ее биты вправо, например, на одну позицию. Делается это командой:

```
x = x >> 1
```

Теперь битовое представление числа:

```
bin(x) # 01010000
```

с десятичным значением 80. То есть, сдвигая биты вправо на один бит, получаем

деление числа на 2.

Если сдвинуть биты сразу на две позиции вправо:

```
x = x >> 2      # x = 80:4 = 20
```

то это будет эквивалентно делению на 4. Если еще раз сдвинуть биты на две позиции вправо:

```
x = x >> 2      # x = 5    00000101
```

получим значение 5. Снова имеем полноценное деление числа на 4. Но что будет, если еще раз сдвинуть биты вправо на одну позицию? Число 5 нацело не делится на два, что тогда в итоге получим? Выполним команду:

```
x = x >> 1      # x = 2    00000010
```

Видим результат 2, то есть, дробная часть как бы отбрасывается. В действительности, дробной части здесь не существовало, это всего лишь такая интерпретация операции смещения бит вправо.

По аналогии можно выполнять смещения бит влево, например:

```
x = x << 1      # x = 4    00000100
```

Получим значение 4, что эквивалентно умножению на два. Или, так:

```
x = x << 3      # x = 32   00100000
```

Это эквивалентно умножению числа 4 на 8 (два в кубе). И так далее. Операции смещения бит влево и вправо выполняют целочисленное умножение и деление кратное двум. Причем эти операции умножения и деления работают значительно быстрее, чем традиционные арифметические операции умножения и деления. Поэтому разработчики различных алгоритмов для маломощных компьютеров стараются составить вычисления так, чтобы они базировались на сдвиговых операциях, исключая прямое умножение и деление.

11.3. Модуль random

Нередко в различных приложениях возникает необходимость генерации случайных чисел. Конечно, ни о какой настоящей случайности здесь речь не идет, значения вычисляются по определенному алгоритму, который имитирует случайность чисел. Именно поэтому они называются **псевдослучайными**.

Подробно о модуле random можно почитать на странице официальной документации:

https://docs.python.org/3/library/random.html

Здесь представлено множество различных функций генерации псевдослучайных значений. Мы с вами затронем некоторые из них, которые наиболее часто используются на практике.

Сразу отмечу, что функции модуля random не следует использовать в задачах криптозащиты и подобных приложениях. Злоумышленнику не составит большого труда воспроизвести эти псевдослучайные значения, так как они вычисляются по известным алгоритмам. Это обстоятельство следует хорошо запомнить и применять функции модуля random исключительно в научных, игровых и других не критичных приложениях.

11.3.1. Функции генерации псевдослучайных величин с разными законами распределений

Для использования модуля random в своих программах его нужно вначале импортировать. Часто это делается командой:

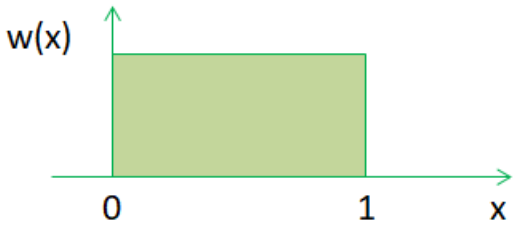
```
import random
```

И далее, из пространства имен random выбираются самые разные функции для работы со случайными значениями.

Первая функция, которую мы рассмотрим, будет функция random:

```
a = random.random()
print(a)              # 0.7878235020679635
```

Она при каждом вызове возвращает псевдослучайное значение из диапазона [0.0; 1.0). Обратите внимание, что числа, которые генерируются этой функцией, подчиняются равномерному закону распределения, то есть, в диапазоне [0.0; 1.0) они могут принимать любое значение с равной вероятностью.



Другая похожая функция uniform(a, b) также генерирует псевдослучайные значения по равномерному закону, но уже в диапазоне от a до b:

```
a = random.uniform(1, 5)
```

В случаях, когда требуется моделировать целочисленные случайные значения с равномерным распределением, можно использовать или функцию:

```
a = random.randint(-3, 7)   # [-3; 7]
```

или функцию randrange:

```
a = random.randrange(5)
b = random.randrange(-3, 10, 2)
```

Обратите внимание, что функция randint модуля random возвращает целые числа из диапазона [a; b], включая обе границы. А функция randrange выполняет

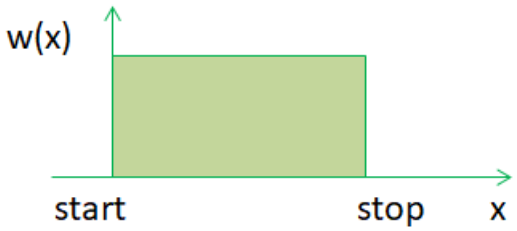
генерацию, не включая правую границу. То есть, используя команду:

```
d = random.randrange(5)
```

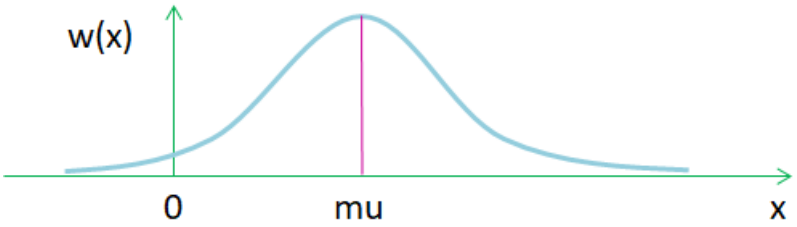
будут формироваться псевдослучайные значения от 0 до 5 (не включая 5) с шагом 1. А вызов функции:

```
b = random.randrange(-3, 10, 2)
```

возвращает псевдослучайные числа от -3 до 10 (не включая 10) с шагом 2.



Однако часто на практике требуются случайные величины с другим, гауссовским (нормальным) законом распределения. Этот закон распределения имеет вид, показанный на рисунке ниже, и определяется двумя параметрами: математическим ожиданием (средним значением) и среднеквадратическим отклонением (мерой разброса относительно математического ожидания).



Особенность этих случайных величин в том, что они чаще появляются в области математического ожидания и реже на более дальних расстояниях. Такому закону распределения часто подчиняются многие реальные события: колебания цен на нефть, цены на товары и услуги, шумы при радиопередачах, погрешности измерений различных величин и т. п. Это распределение окружает нас повсюду! И, конечно же, не редко требуется его моделировать. Для этого в Python существует функция gauss с двумя параметрами: mu – математическое ожидание; sigma – среднеквадратическое отклонение. Например, ее можно вызвать так:

```
a = random.gauss(0, 3.5)
```

В результате будут генерироваться псевдослучайные числа с нулевым средним и среднеквадратическим отклонением 3.5.

11.3.2. Функции для работы с последовательностями

Все ранее рассмотренные функции генерируют и возвращают по одному случайному значению. Однако имеется весьма полезный набор функций для обработки последовательностей. Давайте рассмотрим их на конкретных

примерах.

Предположим, имеется список:

```
lst = [4, 5, 0, -1, 10, 76, 3]
```

из которого случайным образом нужно выбрать один элемент. Для этого можно воспользоваться функцией:

```
a = random.choice(lst)
print(a)
```

После запуска увидим случайно выбранное значение из списка lst.

Следующая функция shuffle перемешивает элементы списка случайным образом:

```
random.shuffle(lst)
print(lst)
```

Причем меняет сам список, поэтому работает только с изменяемыми коллекциями.

Следующая функция sample возвращает новый список с указанным числом неповторяющихся элементов, выбранных случайным образом. Например, вызов:

```
a = random.sample(lst, 3)
print(a)                    # [0, 5, 4] (как пример)
```

возвратит список из трех разных элементов, выбранных случайным образом. Разумеется, максимальное число элементов не может превышать размер самого списка lst.

11.3.3. Функция seed()

Все рассмотренные ранее функции будут генерировать разные наборы псевдослучайных чисел при каждом запуске программы. Например, выполняя каждый раз следующие строчки:

```
a = [random.randint(0, 10) for i in range(20)]
print(a)
```

будем видеть разные случайные значения в списке `a`. Часто именно так и должен работать генератор случайных величин. Но иногда требуется, чтобы каждый запуск программы приводил к одной и той же последовательности случайных чисел. Например, для повторяемости результатов экспериментов на разных компьютерах. Для этого нужно зафиксировать, так называемое, «зерно» генератора случайных значений. Это делается с помощью функции seed, например, следующим образом:

```
random.seed(123)
```

Здесь число 123 – это как раз и есть значение «зерна». Оно задает некое начальное состояние датчика псевдослучайных чисел, поэтому, запуская программу снова и

снова, будем каждый раз видеть одни и те же псевдослучайные значения в списке.

Если выбрать другое значение зерна, то изменится и последовательность псевдослучайных чисел. Этот прием удобно использовать для формирования сколь угодно больших последовательностей чисел, сохраняя только значение «зерна» датчика псевдослучайных значений.

11.4. Аннотации базовыми типами

При оформлении текстов программ язык Python позволяет прописывать у переменных тип данных, который в них ожидается. Это называется **аннотация типов**. Причем аннотация – это всего лишь элемент оформления программного кода, предназначенный для лучшего ее понимания программистом. На процесс выполнения аннотирование никак не влияет. Что же такое аннотация типов и зачем она нужна?

Я напомню, что в Python тип переменной (ссылки) определяется типом объекта, на который она ссылается, то есть, динамически. Отсюда и пошло название **динамическая типизация**. Например:

```
cnt = 0             # тип int
msg = "hello"       # тип str
lst = [1, 2, 3]     # тип list
```

Благодаря такому подходу одним и тем же переменным можно присваивать разные типы данных, например:

```
cnt = -5.3          # тип float
msg = 0             # тип int
```

Здесь важно помнить, что переменные – это ссылки на другие объекты, и сами по себе не хранят присваиваемых данных. Однако нередко в программе переменные создаются для вполне конкретных целей и работают с вполне определенными типами. И было бы хорошо иметь возможность подписывать ключевые переменные, указывая тип данных, с которым они должны работать. Как раз это делается с помощью аннотирования. Например, положим, что переменная cnt работает исключительно с целочисленными значениями. Тогда ее аннотирование типом int можно записать так:

```
cnt: int
```

Обратите внимание, что сама по себе аннотация типа не создает переменную. Она будет создана после первого присваивания ей значения, например, следующей строчкой:

```
cnt = 0
```

Что в итоге изменилось? С программной точки зрения, ничего. Однако если в PyCharm записать команду:

```
cnt = -5.3          # тип float
```

то интегрированная среда подсветит желтым цветом эту конструкцию, как потенциально ошибочную. Это есть результат работы **аннотации типов** совместно со **статическим анализатором**, который проверяет переменные на соответствие указанным типам. Кроме того, при наведении курсора на подсветку, увидим сообщение:

Expected type 'int', got 'float' instead

которое гласит, что ожидается тип int, а вместо него используется float.

Еще раз обратите внимание, что программа выполнится без ошибок, так как аннотация типов носит чисто информационный характер и на ход выполнения программы никак не влияет. В общем случае, она служит следующим целям:

- удобство восприятия стороннего кода;
- удобство редактирования кода, когда IDE «подсказывает» атрибуты указанного типа переменных;
- отслеживание некоторых явных ошибок на уровне несоответствия типов.

Вообще тема аннотаций гораздо объемнее, чем может показаться на первый взгляд. Поэтому мы затронем лишь базовые моменты, которые чаще всего используются на практике. А во всех деталях с этой темой можно ознакомиться на странице официальной документации:

https://docs.python.org/3/library/typing.html

Вернемся к аннотированию переменной cnt. Допускается сразу в одной строчке записывать желаемый тип данных и выполнять инициализацию переменной, например, так:

```
cnt: int = 0
```

Если здесь прописать тип данных, отличающийся от int:

```
cnt: int = 0.5
```

то значение 0.5 будет подсвечено желтым, но программа, по-прежнему, выполнится без ошибок. И так мы можем делать с любыми переменными и любыми типами данных, включая и свои собственные (например, классы).

11.4.1. Аннотация типов в функциях

Конечно, для отдельных переменных аннотация применяется относительно редко. Обычно ее можно встретить при объявлении параметров функций. Давайте посмотрим, как это делается.

Предположим объявляется функция для умножения некоторого числа на два:

```
def mul2(x):
```

```
    return x * 2
```

Эту функцию можно вызывать с разными типами аргументов, например:

```
res1 = mul2(5)
res2 = mul2(4.8)
```

Даже можно указывать не только числа, но, например, список или строку:

```
res3 = mul2([5])
res4 = mul2("5")
```

При этом программа никаких ошибок выдавать не будет, но вместо умножения числа на два, увидим размножение элементов списка или символов строк. Если предполагается в качестве аргументов этой функции передавать только числа, то это можно описать с помощью аннотирования параметров, например, так:

```
def mul2(x: int):
    return x * 2
```

Теперь при вызове:

```
res = mul2("5")
```

интегрированная среда подсветит аргумент "5", как потенциально не соответствующий тип данных. Конечно, программа по-прежнему выполнится без ошибок, но теперь программист видит, что с аргументом функции что-то не так и может оперативно исправить ситуацию.

Если же в аргументе функции прописать целое число:

```
res = mul2(5)
```

то подсветка пропадет, т.к. типы совпадают.

Еще один плюс аннотирования – это подсказки интегрированной среды в виде набора атрибутов, присущих тому или иному типу. Например, если оставить только объявление функции без аннотирования:

```
def mul2(x):
    return x * 2
```

то, записывая внутри функции «x.», мы увидим лишь общие атрибуты. Если же добавить аннотацию:

```
def mul2(x: float):
    return x * 2
```

то при той же записи «x.» появятся атрибуты для типа float. Это дополнительное удобство, которое дает анализатор IDE (интегрированной среды).

Все аннотированные параметры функции попадают в специальную коллекцию __annotations__. Так, для mul2 команда:

```
print(mul2.__annotations__)
```

отобразит в консоли следующий словарь:

```
{'x': <class 'float'>}
```

Если в функцию mul2 добавить еще один параметр:

```
def mul2(x: float, y):
    return x * y
```

то словарь __annotations__ не изменится, так как второй параметр не аннотирован. Но после его аннотирования:

```
def mul2(x: float, y: int):
    return x * y
```

увидим в коллекции __annotations__ два ключа с соответствующими значениями:

```
{'x': <class 'float'>, 'y': <class 'int'>}
```

Причем аннотации совсем не мешают определять параметры со значениями по умолчанию. По-прежнему можно использовать запись вида:

```
def mul2(x: float, y: int = 2):
    return x * y
```

Соответственно при передаче только одного (первого) аргумента, значение параметра y будет принимать значение 2, записанное по умолчанию.

Однако у функции еще есть возвращаемое значение, которое тоже может иметь строго определенный тип. Чтобы аннотировать этот элемент, после описания параметров функции нужно прописать два символа -> (стрелочка) и указать возвращаемый тип. Например:

```
def mul2(x: float, y: int) -> float:
    return x * y
```

В результате коллекция __annotations__ будет содержать специальный ключ ‘return’ со значением float:

```
{'x': <class 'float'>, 'y': <class 'int'>, 'return': <class 'float'>}
```

Если же функция ничего не возвращает (на самом деле такие функции возвращают None), то указывается значение None:

```
def show_x(x: float) -> None:
    print(f"x = {x}")
```

Это простые примеры аннотирования параметров и возвращаемого значения функций базовыми типами языка Python.

11.4.2. Модуль typing и типы Union, Optional, Any, Final

Если внимательно посмотреть на функцию mul2:

```
def mul2(x: float, y: int) -> float:
    return x * y
```

то ее параметры `x` и `y` вполне могут принимать как тип int, так и тип float. Получается при аннотировании нужно уметь прописывать сразу несколько типов данных. Для решения этой и некоторых других задач были разработаны специальные типы, доступные в модуле typing:

```
from typing import Union, Optional, Any, Final
```

Давайте рассмотрим их по порядку. Первый тип Union позволяет комбинировать несколько разных типов в один, образуя единый **составной тип**. Например, с его помощью можно скомбинировать два типа int и float следующим образом:

```
def mul2(x: Union[int, float], y: Union[int, float]) -> Union[int, float]:
    return x * y
```

То есть в квадратных скобках через запятую прописываются типы, которые ожидаются у параметров `x` и `y`, а также у возвращаемого значения функции.

Сразу отмечу, что начиная версии Python 3.10, объединение типов в аннотациях можно записывать с использованием символа «|»:

```
def mul2(x: int | float, y: int | float) -> int | float:
    return x * y
```

И сейчас это рекомендуемый подход. Но вернемся к прежней записи:

```
Union[int, float]
```

Здесь, фактически, определяет новый объект – новый тип данных. А раз это так, то его можно присвоить какой-либо переменной следующим образом:

```
digit = Union[int, float]
```

Или, используя новый синтаксис:

```
digit = int | float
```

А затем использовать переменную digit при аннотировании:

```
def mul2(x: digit, y: digit) -> digit:
    return x * y
```

В ряде случаев такой подход делает текст программы более читабельным. Хотя, усердствовать в этом не стоит, так как программисту придется искать определение digit и помнить, что оно из себя представляет (какой набор типов).

Следующее определение Optional позволяет формировать строго определенный тип данных. Рассмотрим его работу на конкретном примере. Если записать

команду:

```
String = Optional[str]
```

Это будет эквивалентно команде:

```
String = str | None
```

То есть, тип str автоматически объединяется с типом None. Такая комбинация (тип | None) довольно часто применяется на практике, поэтому отдельно для нее был определен объект Optional.

Пусть, например, объявляется функция show_x, которая имеет параметр descr со значением по умолчанию None. Ее аннотирование можно записать в виде:

```
def show_x(x: float, descr: Optional[str]=None) -> None:
    if descr:
        print(f"{descr} {x}")
    else:
        print(f"x = {x}")
```

Если при вызове функции show_x передать один аргумент:

```
show_x(55.6768)
```

то сработает блок else с выводом строки:

```
x = 55.6768
```

А если указать два аргумента:

```
show_x(55.6768, 'x:')
```

то сработает условие и на экране увидим:

```
x: 55.6768
```

Если же во втором аргументе указать не строку, а число:

```
show_x(55.6768, 10)
```

то интегрированная среда подсветит его из-за несоответствия ожидаемого типа данных. То есть, Optional введен для удобства, чтобы не прописывать два типа, один из которых None.

Следующий тип Any буквально означает любой тип данных. Он используется, если параметр, или переменная, или возвращаемое значение функции может принимать любой тип. Или же, попросту, сложно указать какой-либо конкретный тип данных, т.к. он неизвестен. Записывается такая нотация следующим образом:

```
def show_x(x: Any, descr: Optional[str]=None) -> None:
    if descr:
        print(f"{descr} {x}")
    else:
```

```
        print(f"x = {x}")
```

Наконец последний тип Final появился в версии Python 3.8 и служит для отметки констант (неизменяемых переменных или параметров). Например, если прописать:

```
MAX_VALUE: Final = 1000
```

то при попытке далее изменить это значение:

```
MAX_VALUE = 2000
```

интегрированная среда подсветит такую переменную с сообщением, что идет попытка изменения константы. При этом программа, как всегда, отработает без ошибок.

11.5. Аннотирование различных коллекций

На данный момент мы с вами уже знаем, как аннотировать отдельные переменные и функции простыми базовыми типами:

int, float, str, bool и т. п.

Давайте теперь посмотрим, как делать аннотацию для коллекций языка Python:

list, tuple, dict, set

11.5.1. Аннотация списков

Первое, что приходит в голову, при аннотации переменных или параметров функций типом list – это прописать его так, как это мы делали в предыдущем параграфе:

```
lst: list = [1, 2, 3]
```

И, действительно, так вполне можно делать. Но, обратите внимание, в этой аннотации нет никаких указаний на тип элементов списка. Поэтому можно прописать и числа, и строки, и вообще все, что угодно:

```
lst_str: list = [1, '2', (5, 6), '3']
```

Интегрированная среда не выдаст никаких предупреждений. А вот если вместо списка записать, например, кортеж:

```
lst: list = (1, 2, 3)
```

то увидим подсветку с предупреждением. То есть, тип list в аннотации отвечает только за определение списка. Типы элементов, при этом, могут быть любыми.

Но что если нам нужно дополнительно в аннотации указать и тип элементов списка? Начиная с версии Python 3.9, это можно делать с помощью все того же базового типа list следующим образом:

```
lst: list[int] = [1, 2, 3]
```

Здесь в квадратных скобках указывается тип элементов списка (они все полагаются целочисленными).

Если в этой аннотации прописать другой тип, например, str:

```
lst: list[str] = [1, 2, 3]
```

то увидим подсветку кода. Но, как только в списке оказывается хотя бы один строковый элемент:

```
lst: list[str] = ['1', 2, 3]
```

подсветка пропадает. Это особенность обработки аннотации в PyCharm. Забегая вперед, скажу, что для языка Python есть довольно полезный модуль mypy, с помощью которого можно строже оценивать корректность программы на соответствие записанным аннотациям. О модуле mypy речь пойдет ниже.

Вернемся к аннотации списка list[str]. Записывать ее в таком виде можно, начиная с версии Python 3.9 и выше. Если версия вашего интерпретатора ниже, то для достижения того же результата придется импортировать из модуля typing следующие типы:

```
from typing import List, Tuple, Dict, Set
```

а затем вместо list прописывать List:

```
lst: List[str] = [1, 2, 3]
```

Эта конструкция работает абсолютно также, как и типизация через list. Но List и другие подобные типы считаются устаревшими и в новых версиях их лучше не использовать.

11.5.2. Аннотация кортежей

С кортежами в целом все также, как и со списками. Единственное ключевое отличие – это независимое указание типов для всех элементов кортежа. Например:

```
address: tuple[int, str] = (1, 'proproprogs.ru')
```

Возможно здесь у вас возникает вопрос. Почему у списка указывается один тип для всех элементов, а у кортежа для каждого? Как вы помните, кортеж – это неизменяемый тип данных. В частности, мы не можем добавлять или удалять элементы. Поэтому их число полагается известным и фиксированным. Часто кортежи в Python используются для группировки каких-либо данных в единый объект. Например, информация по книге может быть представлена в виде кортежа:

```
book: tuple[str, str, int]
book = ('Балакирев С.М.', 'Добрый, добрый Python', 2026)
```

Здесь наперед известно, какие данные и в каком порядке предполагается хранить. Поэтому прописать аннотацию для всех его элементов не представляет особого труда. В дальнейшем она будет служить хорошей подсказкой программисту.

Если же требуется аннотировать кортеж с произвольным числом элементов, то это можно сделать так:

```
elems: tuple[int, ...]
```

И тогда все следущие строчки:

```
elems = (1,)
elems = (1, 2)
elems = (1, 2, 3)
```

будут корректны с точки зрения этой аннотации.

Я думаю, вы уже догадались, что для версий языка Python ниже 3.9 следует вместо tuple использовать тип Tuple, чтобы иметь возможность дополнительно указывать тип данных для отдельных элементов:

```
book: Tuple[str, str, int]
book = ('Балакирев С.М.', 'Аннотация типов', 2022)
```

Но, начиная с версии 3.9, рекомендуется использовать стандартный тип tuple с квадратными скобками.

11.5.3. Аннотация словарей и множеств

Аналогичным образом выполняется аннотация словарей. Так как словари – это изменяемый тип данных и число их элементов наперед не определено, то при аннотировании указываются лишь тип ключа и тип значения. Например:

```
words: dict[str, int] = {'one': 1, 'two': 2}
```

(для Python ниже 3.9):

```
words: Dict[str, int] = {'one': 1, 'two': 2}
```

Эта аннотация ведет себя также, как и при аннотировании списков.

И то же самое для множеств:

```
persons: set[str] = {'Сергей', 'Михаил', 'Наталья'}
```

(для Python ниже 3.9):

```
persons: Set[str] = {'Сергей', 'Михаил', 'Наталья'}
```

Здесь также указывается единый тип для всех элементов множества. Обратите внимание, что подсветка с предупреждением появляется, когда все элементы множества или словаря не соответствуют указанному типу. Но модуль mypy (о котором речь впереди) позволяет находить все эти несоответствия.

11.5.4. Комбинирование типов

Разумеется все приведенные способы аннотирования можно использовать не только для отдельных переменных, но и при объявлении функций. Например, так:

```
def get_positive(digits: list[int]) -> list[int]:
    return list(filter(lambda x: x > 0, digits))


print(get_positive([1, -2, 3, 4, -5]))
```

Здесь полагается, что все элементы списка будут целочисленными. Если же список может содержать произвольные числа (и целые и вещественные), то для этого нужно использовать объединенный тип. До версии Python 3.10 это делалось с помощью знакомого нам объекта Union из модуля typing, например, следующим образом:

```
def get_positive(digits: list[Union[int, float]]) -> list[Union[int,
float]]:
    return list(filter(lambda x: x > 0, digits))
```

А, начиная с версии Python 3.10, эту же аннотацию можно определить так:

```
def get_positive(digits: list[int | float]) -> list[int | float]:
    return list(filter(lambda x: x > 0, digits))


print(get_positive([1, -2, 3, 4, -5]))
```

И это рекомендуемый вариант.

Давайте еще несколько усложним задачу для задачи аннотирования. Предположим, что по умолчанию параметр digits должен принимать значение None. Мы уже знаем, что для этого следует использовать объект Optional модуля typing. Имеем:

```
def get_positive(digits: Optional[list[Union[int, float]]] = None) ->
list[Union[int, float]]:
    return list(filter(lambda x: x > 0, digits))
```

Или, начиная с версии Python 3.10, можем записать так:

```
def get_positive(digits: Optional[list[int | float]] = None) -> list[int |
float]:
    return list(filter(lambda x: x > 0, digits))
```

И даже еще проще:

```
def get_positive(digits: list[int | float | None] = None) -> list[int |
float]:
    return list(filter(lambda x: x > 0, digits))
```

Можно использовать любой вариант, но нужно стремиться к читаемости

программного кода. В ряде случаев для наглядности можно воспользоваться алиасами (переменными на типы). Например, если объявить следующий алиас:

```
digit = int | float
```

то затем можно прописать его в аннотации:

```
def get_positive(digits: Optional[list[digit]] = None) -> list[digit]:
    return list(filter(lambda x: x > 0, digits))
```

Или в таком виде:

```
def get_positive(digits: list[digit | None] = None) -> list[digit]:
    return list(filter(lambda x: x > 0, digits))
```

Так текст программы воспринимается гораздо лучше. Разумеется можно задавать и более глубокие, вложенные алиасы, например:

```
ListDigits = list[digit]
```

А затем использовать при аннотировании:

```
def get_positive(digits: Optional[ListDigits] = None) -> ListDigits:
    return list(filter(lambda x: x > 0, digits))
```

Но увлекаться этим не стоит. Иначе программисту постоянно придется искать определения алиасов и смотреть, что за типы в них прописаны. Во всем должна быть мера.

11.5.5. Аннотации для вызываемых объектов (Callable)

Модуль typing содержит еще одно определение Callable для аннотирования вызываемых объектов, например, функций. Он полезен, когда в параметр функции передается ссылка на другую функцию (вызываемый объект). Давайте посмотрим на примере, как можно выполнять аннотирование данных такого типа.

Пусть имеется функция, которая из переданной коллекции целых чисел выбирает строго определенные по некоторому критерию:

```
def get_digits(flt: Callable[[int], bool], lst: Optional[list[int]] = None)
-> list[int]:
    if lst is None:
        return []

    return list(filter(flt, lst))
```

Здесь первый параметр отмечен как вызываемый тип (Callable), у которого ожидается целочисленный аргумент int и булево возвращаемое значение bool.

Объявим еще одну функцию, которую будем передавать в первый параметр функции get_digits:

```
def even(x):
```

```
    return x % 2 == 0
```

Эта функция возвращает True для четных чисел и False для нечетных. Вызовем теперь функцию get_digits со ссылкой на функцию even, получим:

```
res = get_digits(even, [1, 2, 3, 4, 5, 6, 7])
print(res)
```

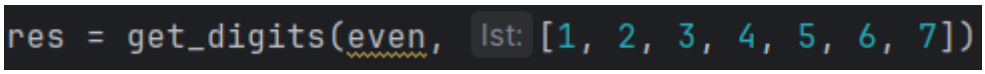
После запуска программы увидим результат:

```
[2, 4, 6]
```

Теперь, для примера, изменим функцию even так, чтобы она возвращала не булево значение, а строку:

```
def even(x):
    return str(x % 2 == 0)
```

Видим, что первый аргумент в функции get_digits подсвечен желтым цветом из-за несоответствия возвращаемого типа указанного в аннотации:



Так используется Callable для аннотирования вызываемых объектов. В общем случае он задается согласно следующему синтаксису:

Callable[[TypeArg1, TypeArg2, …], ReturnType]

Например, если нужно аннотировать функцию, у которой нет параметров и она ничего не возвращает:

```
def hello_callable():
    print("Hello Callable")
```

то объект Callable для нее будет иметь вид:

```
Callable[[], None]
```

11.6. Аннотации классами. Инструмент mypy проверки аннотирования

Язык Python может оперировать не только стандартными объектами типа str, list, dict, set и т.п., но и формировать новые. Для этого в программах используются классы с возможностью конструирования их объектов. Подробно об этом мы с вами будем говорить во второй части этого пособия. Но, забегая вперед, рассмотрим способы аннотирования такими типами данных (классами).

Вначале запишем простую аннотацию переменной стандартным типом dict:

```
tr: dict = {'car': 'машина'}
```

В действительности dict – это класс, который описывает работу со словарем. А имя класса воспринимается, как название типа. Значит, при аннотации можно указывать любые классы в том числе и самый базовый (фундаментальный)

`object`, от которого неявно происходят все классы в Python 3. Например:

```
x: object = None
x = "123"
x = 123
```

Эта аннотация позволяет прописывать любые типы данных, унаследованные от object, и интегрированная среда, при этом, не выдаст никаких предупреждений. А это практически все типы данных в языке Python. Поэтому строки, числа и любые другие стандартные объекты можно присваивать переменной x без нарушения данной аннотации.

Или такой пример. Пусть в программе объявляются два класса Geom и Line, причем класс Line наследуется от класса Geom:

```
class Geom: pass
class Line(Geom): pass
```

После этого аннотируем переменную g типом Geom:

```
g: Geom
```

Если теперь присвоить переменной g объект класса Line:

```
g = Line()
```

то интегрированная среда не будет подсвечивать этот код, так как, данные соответствуют аннотации. Очевидно, это связано с тем, что объект типа Line наследуется от типа Geom.

А если убрать наследование класса Line от класса Geom:

```
class Line: pass
```

то появляется подсветка, т.к. теперь типы Line и Geom независимы между собой.

Все эти примеры показывают, как именно аннотация типов учитывает иерархию наследования классов при проверке корректности присваиваемых данных.

11.6.1. Отличие между object и typing.Any

Возможно, здесь у вас возникает вопрос. А в чем отличия между типом object и типом Any из модуля typing? Если все типы данных в Python наследуются от object, значит это эквивалентно использованию типа Any? Почти. Но есть один нюанс. Его можно сформулировать так:

Тип Any совместим с любым другим типом, а тип object – ни с каким другим.

Давайте проясним смысл этой фразы на конкретном примере. Допустим, имеется некая переменная аннотированная типом Any:

```
a: Any = None
```

А, затем определяется еще одна переменная, аннотированная типом str:

```
s: str
```

После этого корректным считается присвоение первой переменной второй:

```
s = a
```

Подсветки кода не возникнет. Но если вместо Any прописать object:

```
a: object = None
s: str = a            # появляется подсветка
```

то подсветка появляется, т.к. тип object несовместим с типом str. То есть, с точки зрения приведенной аннотации будет неверным присваивать переменной типа str переменную типа object. Потому что str наследуется от object, а не наоборот. А вот если бы типы были записаны в другом порядке:

```
a: str = '123'
s: object
s = a
```

то никаких предупреждений не возникало бы.

Когда же прописывается Any, то совершенно не имеет значения, какой тип будет присвоен этой переменной, и куда она далее будет присваиваться:

```
a: Any = '123'
s: object
s = a
```

Any – это некий эквивалент отключения проверки типов, то есть, поведение статического анализатора по умолчанию. Однако, прописывая Any, мы явно указываем, что переменная может ссылаться на произвольный тип данных.

11.6.2. Инструмент mypy

До сих пор мы непосредственно в интегрированной среде PyCharm смотрели на подсветку кода и понимали, что нарушено, включая аннотацию типов. Но доверять целиком и полностью анализу интегрированной среды не стоит. Вполне могут возникать ситуации, когда нарушения типов не подсвечиваются и мы с вами это уже видели на примере аннотирования элементов списка. Чтобы строже оценивать корректность программы с точки зрения аннотации типов, часто на практике используют инструмент mypy. Им также удобно пользоваться, если под рукой нет интегрированной среды, а неточности как-то требуется выявлять. Подробнее о mypy можно почитать на странице официальной документации:

https://mypy.readthedocs.io/en/stable/

Так как mypy – это внешнее приложение, которое не поставляется вместе с языком Python, то вначале его нужно установить. Делается это стандартной командой:

pip install mypy

После этого в терминале будет доступна команда:

mypy <имя файла>

Например, если программа записана в файле «main.py», а сам файл хранится в текущем рабочем каталоге, то команду mypy можно записать в виде:

mypy main.py

Для анализа сразу нескольких файлов все их следует прописывать через пробел.

После запуска mypy выдает в терминале информацию о найденных нарушений типизаций. Например:

```
main.py:3: error: Incompatible types in assignment (expression has type
"object", variable has type "str")  [assignment]
Found 1 error in 1 file (checked 1 source file)
```

Если же никаких ошибок нет, то в терминале появится строчка:

```
Success: no issues found in 1 source file
```

11.6.3. Аннотация типами Type и TypeVar

Давайте снова вернемся к рассмотрению аннотирования и, предположим, что в программе объявлены два класса Geom и Point2D, причем класс Point2D наследуется от класса Geom:

```
class Geom: pass
class Point2D(Geom): pass
```

Пусть также имеется некая функция, которая создает экземпляры переданных ей классов, унаследованных от Geom:

```
def factory_point(cls_geom):
    return cls_geom()
```

В этой функции полагается, что параметр cls_geom ссылается на сам класс Geom (на тип данных), а не объект класса. Спрашивается, как выполнить аннотирование типа данных, а не значений определенного типа, как это было до сих пор? Используя текущие знания, первое, что приходит в голову – это все же указать тип Geom:

```
def factory_point(cls_geom: Geom) -> Geom:
    return cls_geom()
```

Однако интегрированная среда в этом случае подсветит фрагмент cls_geom. Это произошло по той причине, что записанная аннотация `:Geom` подразумевает передачу в параметр cls_geom объектов класса Geom, а не самого класса. Этот момент важно понимать при записи аннотаций: в них подразумеваются объекты тех типов, которые указываются. Но как тогда поправить эту ситуацию и

прописать тип для класса? По сути, тип для типа? Для этого в модуле typing существует специальное определение Type, которое в нашем примере можно использовать следующим образом:

```
def factory_point(cls_geom: Type[Geom]) -> Geom:
    return cls_geom()
```

Теперь никаких подсветок кода, связанных с нарушением типов, не будет. То есть, здесь в аннотации указывается, что параметр cls_geom ссылается непосредственно на класс Geom, а не на его объекты. Далее переменная cls_geom, используется для создания объект этого класса, который возвращается функцией.

Давайте теперь воспользуемся этой функцией и вызовем ее так:

```
geom = factory_point(Geom)
point = factory_point(Point2D)
```

Как видим, все аннотации соблюдены. Но если дополнительно аннотировать переменные geom и point соответствующими типами:

```
geom: Geom = factory_point(Geom)
point: Point2D = factory_point(Point2D)  # подсветка с предупреждением
```

то во второй строчке появляется подсветка кода. Очевидно это из-за того, что явно указывается ожидаемый тип Point2D, а в определении функции прописан тип Geom. И, так как Geom – это базовый класс для Point2D, то возникает конфликт аннотаций.

Для исправления таких ситуаций в Python можно описывать общие типы с помощью еще одного определения TypeVar из модуля typing, например, так:

```
T = TypeVar("T", bound=Geom)
```

Здесь объявляется универсальный тип с именем T и указывается, что он должен быть или классом Geom или любым его дочерним классом. Далее этот тип прописывается при аннотировании в самой функции:

```
def factory_point(cls_geom: Type[T]) -> T:
    return cls_geom()
```

Он автоматически вычисляется при вызове функции. Когда передается класс Geom, то T будет соответствовать этому типу, а когда передается Point2D (унаследованный от Geom), то тип T будет соответствовать Point2D. И так далее. В этом смысл универсальных типов при формировании аннотаций.

Для полноты картины сразу отмечу, что класс TypeVar можно использовать и в таких вариациях:

```
T = TypeVar("T")              # T - произвольный тип без ограничений
T = TypeVar("T", int, float) # T - тип связанный только с int и float
```

Подробнее обо всем этом можно почитать в официальной документации. Мы же

глубоко в эту тему погружаться не будем. Для начала, приведенной информации вполне достаточно.

11.6.4. Аннотации внутри классов

В предыдущем параграфе мы видели, как прописываются аннотации, используя различные классы. Но ее можно записывать и внутри самих классов. В целом все делается практически так же, как и в случае с переменными и функциями.

Давайте для примера распишем класс Point2D следующим образом:

```
class Point2D:
    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y
```

Обратите внимание, что параметр self не принято аннотировать. Также метод __init__ обычно возвращает None.

Воспользоваться этим классом можно следующим образом:

```
p = Point2D(10.5, 20)
```

Видим подсветку первого аргумента, т.к. он не соответствует целому типу. Но программа отработает без ошибок. Однако если затем воспользоваться объектом p и, например, атрибуту x присвоить другой тип данных:

```
p.x = '10'
```

то интегрированная среда никак на это не отреагирует. А вот инструмент mypy найдет все эти несоответствия. После выполнения команды (в терминале):

mypy main.py

увидим строчки:

```
main.py:7: error: Argument 1 to "Point2D" has incompatible type "float";
expected "int"  [arg-type]
main.py:8: error: Incompatible types in assignment (expression has type
"str", variable has type "int")  [assignment]
Found 2 errors in 1 file (checked 1 source file)
```

Однако если аннотировать атрибуты x, y класса Point2D явно:

```
class Point2D:
    x: int
    y: int

    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y
```

то интегрированная среда PyCharm будет реагировать на все несоответствия

присваиваемых типов данных переменным x, y.

Еще одна особенность аннотации типов внутри классов заключается в использовании типа текущего класса. Например, если добавить в класс Point2D метод copy, который создает копию объекта текущего класса:

```
class Point2D:
    x: int
    y: int

    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y

    def copy(self) -> Point2D:
        return Point2D(self.x, self.y)
```

то просто так записать имя Point2D в аннотации внутри этого же класса не получится, возникнет ошибка. Существует два способа разрешить эту ситуацию. В первом (устаревшем) подходе имя типа Point2D записывается в кавычках:

```
    def copy(self) -> 'Point2D':
        return Point2D(self.x, self.y)
```

Но можно сделать лучше. Если импортировать из модуля __future__ объект annotations:

```
from __future__ import annotations
```

то после этого можно убрать кавычки у имени класса, все будет корректно работать:

```
    def copy(self) -> Point2D:
        return Point2D(self.x, self.y)
```

Следует отметить, что начиная с версии Python 3.14, прописывать команду:

```
from __future__ import annotations
```

не нужно, она используется по умолчанию.

11.6.5. Когда следует использовать аннотацию типов

На данный момент мы с вами рассмотрели основные возможности аннотирования переменных, параметров, функций и классов языка Python. Конечно, это далеко не все, что существует, но в большинстве случаев приведенного материала будет вполне достаточно. И последнее, на что следует сделать акцент, – это вопрос, когда и для чего следует использовать аннотацию типов?

Конечно, если вы пишите небольшие, короткие программы, которые достаточно легко охватить взглядом, то, как таковая, аннотация мало чего дает. Программист сможет и так без труда понять, какие данные куда передаются. Поэтому лишняя запись может только ухудшить восприятие кода. На мой взгляд, для небольших

программ достаточно аннотировать ключевые функции или даже совсем ее не делать, как это было в самых ранних версиях языка. Если же создается средний или крупный проект, состоящий из множества модулей, то здесь грамотное аннотирование заметно упрощает понимание кода сторонними программистами. Да и сами авторы своих программ благодаря написанной аннотации смогут спустя какое-то время быстро восполнить важные нюансы их работы. Здесь аннотирование действительно играет важную роль. Однако во всем должна быть мера. Аннотациями можно легко засорить текст программы и получить обратный эффект. Всегда следует помнить, что цель аннотирования – это упрощение восприятия текста программы. И, как только, аннотации начинают мешать – это верный признак усмирить свой энтузиазм и вернуться непосредственно к написанию кода.

11.7. Введение в конструкцию match/case

В Python 3.10 появилась новая конструкция под названием match/case, которая позволяет достаточно гибко анализировать переменные на соответствие шаблонов и для найденного совпадения выполнять некоторые заданные операции. Оператор match/case имеет следующий синтаксис:

```
match <переменная>:
    case <шаблон_1>:
        операторы
    …
    case < шаблон_n>:
        операторы
    case _:
        иначе (default)
```

Чтобы понять, как работает эта конструкция, давайте вначале рассмотрим очень простой примем ее использования:

```
cmd = "top"

match cmd:
    case "top":
        print("вверх")
    case "left":
        print("влево")
    case "right":
        print("вправо")
    case _:
        print("другое")

print("проверки завершены")
```

Программа начинается с объявления переменной cmd со значением "top". Далее она указывается в операторе match, внутри которого записаны блоки case. После каждого case записана определенная строка. Так вот, как только значение переменной cmd оказывается равным одному из значений после case, то

выполняются операторы внутри соответствующего блока и далее выполнение программы переходит к следующей команде, записанной после оператора match. В нашем примере – это функция:

```
print("проверки завершены")
```

То есть, как только выполняется один из блоков case, оператор match завершает свою работу. Соответственно, внутри match обязательно должен быть записан хотя бы один оператор case. А общее число блоков case может быть любым. Конечно, при этом, всегда следует помнить о читаемости программы и удобстве ее последующего редактирования.

Внутри каждого блока case должен быть хотя бы один оператор. Например, запись вида:

```
match cmd:
    case "top":
    case "left":
    case "right":
        print("вверх, влево или вправо")
    case _:
        print("другое")
```

приведет к синтаксической ошибке. Если нужно в одном блоке case учесть сразу несколько констант, это можно сделать с помощью операции ‘|’ следующим образом:

```
cmd = "top"

match cmd:
    case "top" | "left" | "right":
        print("вверх, влево или вправо")
    case _:  # wildcard
        print("другое")

print("проверки завершены")
```

Обратите внимание на последний блок case с символом подчеркивания. Это, так называемый, «wildcard» символ. В Python он заменяет неиспользуемую переменную. В частности, здесь блок case с переменной _ будет отрабатывать всегда, если не сработал ни один из предыдущих блоков. То есть, это некий аналог условного оператора else в операторе if. Например, если переменная:

```
cmd = "top2"
```

то в консоли отобразятся строчки:

```
другое
проверки завершены
```

То есть, здесь отработал последний блок case.

Те из вас, кто знаком с другими языками программирования, например, C++ или Java, возможно сейчас смотрят на эту программу и видят аналог конструкции switch/case для проверки переменных на равенство заданных констант. Но не спешите с выводами. В действительности оператор match/case гораздо более гибкий, чем switch/case и гибкость определяется тем, что после case прописывается не просто константа, а шаблон проверки. На данный момент был использован простейший шаблон сравнения переменной cmd на равенство указанным после case значениям. Но вариаций здесь гораздо больше и сейчас об этом пойдет речь.

11.7.1. Шаблоны сравнений оператора case

Вместо конкретных значений после оператора case можно записывать и переменные. Например, так:

```
cmd = "top"

match cmd:
    case command:
        print(f"команда: {command}")
```

При выполнении блока case будет создана переменная command (если ранее ее не было в программе) со ссылкой на то же значение, что и переменная cmd, то есть, на строку "top". Поэтому при выполнении программы на экране увидим сообщение:

```
команда: top
```

Фактически, в блоке case выполняется присваивание:

```
command = cmd
```

и выводится значение переменной command в консоль. Причем этот блок case будет выполняться всегда при любых значениях переменной cmd, потому что такой шаблон не делает никаких проверок, он просто создает переменную command и присваивает ей значение переменной, указанной после оператора match. Поэтому если ниже прописать еще один блок case, например:

```
match cmd:
    case command:
        print(f"команда: {command}")
    case "top":
        print("top")
```

то возникнет синтаксическая ошибка, из-за того, что первый блок case уже перехватывает все возможные варианты и добавлять новые после него просто не имеет смысла. Подобные блоки с общим шаблоном допустимо записывать в последнюю очередь, например, так:

```
match cmd:
    case "top":
```

```
        print("top")
    case command:
        print(f"команда: {command}")
```

Теперь, после запуска программы, увидим строчку:

```
top
```

то есть, был выполнен первый блок case.

Если в этой программе вместо переменной command прописать нижнее подчеркивание, то получим уже знакомый нам блок default, который перехватывает все, что не вошло в предыдущие блоки:

```
match cmd:
    case "top":
        print("top")
    case _:
        print(f"другая команда")
```

11.7.2. Шаблоны проверки типов

Давайте теперь усложним шаблон и в первом блоке case будем перехватывать все строковые команды, то есть, те, которые записаны в виде строки. Сделать это можно следующим образом:

```
cmd = "top"

match cmd:
    case str() as command:
        print(f"строковая команда: {command}")
    case _:
        print(f"другая команда")
```

Смотрите, после case записано str(). Мы привыкли такую запись воспринимать как создание или пустой строки или преобразование объекта в строку. Здесь же это не что иное, как проверка переменной cmd строковому типу. То есть, если переменная cmd является строкой, то только в этом случае выполнится данный блок case и будет создана переменная command, которая, как и ранее, ссылается на значение переменной cmd.

Обратите еще раз внимание на конструкцию «str() as command». Вначале идет проверка строкового типа переменной cmd, и если это так, то переменная, записанная после ключевого слова as, будет ссылаться на проверяемую строку. И если нам нужна только проверка на строковый тип данных, то фрагмент «as command» можно не прописывать. Например:

```
match cmd:
    case str():
        print(f"строковая команда")
    case _:
```

```
        print(f"другая команда")
```

Или вместо «as command» можно записать:

```
match cmd:
    case str(command):
        print(f"строковая команда: {command}")
    case _:
        print(f"другая команда")
```

Это будет одно и то же, но выглядит несколько путанно, так как в Python такая конструкция выполняет преобразование объекта в строку. Поэтому нагляднее использовать ключевое слово as.

По аналогии можно делать проверки на другие типы данных, например:

```
cmd = 10

match cmd:
    case str() as command:
        print(f"строковая команда: {command}")
    case int() as command:
        print(f"целочисленная команда: {command}")
    case bool() as command:
        print(f"булева команда: {command}")
    case _:
        print(f"другая команда")
```

При выполнении этой программы будет выполнен второй блок case, так как переменная cmd целочисленная:

```
целочисленная команда: 10
```

Если переменной cmd присвоить булево значение:

```
cmd = True
```

и запустить программу, то снова отработает второй блок case, а не третий, как этого можно было бы ожидать:

```
целочисленная команда: True
```

Почему так произошло? Я думаю, многие из вас уже догадались, что проверка на тип данных в операторах case выполняется по принципу функции isinstance, то есть, с учетом цепочки наследования типов. В частности, булевый тип наследуется от целочисленного, поэтому isinstance для них возвращает True.

Разрешить эту проблему достаточно просто. Так как шаблоны в операторах case проверяются по порядку (сверху-вниз), то сначала следует поставить проверку на булевый тип, а потом на целочисленный:

```
match cmd:
    case str() as command:
        print(f"строковая команда: {command}")
```

```
    case bool() as command:
        print(f"булева команда: {command}")
    case int() as command:
        print(f"целочисленная команда: {command}")
    case _:
        print(f"другая команда")
```

Тогда все проверки на тип данных будут проходить корректно. По аналогии можно делать проверку на любые типы данных, которые имеются в языке Python.

11.7.3. Guard (защитник)

Давайте сделаем еще один шаг усложнения шаблонов и предположим, что для целочисленных команд дополнительно нужно проверять, чтобы они были в диапазоне от 0 до 9 включительно. Для этого после определения переменной command можно прописать ключевое слово if и указать дополнительное условие. В нашем случае оно будет выглядеть так:

```
match cmd:
    case str() as command:
        print(f"строковая команда: {command}")
    case bool() as command:
        print(f"булева команда: {command}")
    case int() as command if 0 <= command <= 9:
        print(f"целочисленная команда: {command}")
    case _:
        print(f"другая команда")
```

Теперь блок `case int() as command if 0 <= command <= 9` отработает только в том случае, если переменная cmd целочисленная и значение переменной command (которая ссылается на переменную cmd) лежит в диапазоне [0; 9]. Причем, обратите внимание, оператор if будет отрабатывать после команды int(), то есть, после проверки на целочисленный тип данных и после формирования переменной command. Поэтому если проверка дошла до if, то мы точно знаем, что определена переменная command и она имеет целочисленный тип. Проверка шаблона в case всегда выполняется строго слева-направо.

В конструкции match/case оператор if получил название guard (то есть, защитник). Условия в этом операторе можно прописывать по тем же правилам, что и в условных операторах if языка Python.

По аналогии можно записать guard и для строковых команд, например, так:

```
cmd = "c_top"

match cmd:
    case str() as command if len(command) < 10 and command[0] == 'c':
        print(f"строковая команда: {command}")
    case bool() as command:
        print(f"булева команда: {command}")
    case int() as command if 0 <= command <= 9:
```

```
        print(f"целочисленная команда: {command}")
    case _:
        print(f"другая команда")
```

Если же какая-либо переменная может принимать несколько разных типов, например, int или float, то их можно записывать через оператор ‘|’ следующим образом:

```
case int() | float() as command if 0 <= command <= 9: ...
```

И так в шаблонах оператора case можно определять любые требуемые условия.

Возможно у вас здесь возникает все тот же вездесущный вопрос, зачем это надо? В языке Python уже есть условные операторы if/elif/else и через них можно делать все те же самые проверки? Все верно. Конструкция match/case не дает нам ничего принципиально нового. Она лишь может упростить написание программного кода (при ее грамотном использовании), а также повысить читаемость текста программы. Это, наверное, главные причины, по которым данные операторы были добавлены в язык Python. Как мы дальше увидим, шаблоны в операторах case позволяют очень гибко, просто и удобно обрабатывать коллекции данных и выделять из них определенные значения.

11.8. Конструкция match/case с упорядоченными коллекциями данных

Давайте представим, что на вход оператора match передается кортеж или список. Как для этих типов данных выстраиваются шаблоны в операторах case? Для простоты зададим кортеж из трех элементов, содержащий информацию о книге, например, такой:

```
cmd = ("Балакирев С.М.", "Python", 3000.78)
```

Здесь первые две строки – это автор и название книги, а последнее число – цена книги. Если сейчас прописать в операторе case одну переменную с проверкой на тип данных, например, так:

```
match cmd:
    case tuple() as book:
        print(f"кортеж: {book}")
    case _:
        print("непонятный формат данных")
```

то после запуска программы отобразится следующая строка:

```
кортеж: ('Балакирев С.М.', 'Python', 3000.78)
```

Пока все работает предсказуемым образом: формируется новая переменная book, которая ссылается на кортеж с данными о книге. Но, так как кортеж относится к коллекциям с упорядоченными данными, то его элементы можно распаковывать непосредственно в шаблонах, например, следующим образом:

```
match cmd:
    case author, title, price:
```

```
        print(f"Книга: {author}, {title}, {price}")
    case _:
        print("непонятный формат данных")
```

Фактически, здесь используется уже знакомая нам конструкция распаковки последовательностей вида:

```
author, title, price = cmd
```

После запуска программы, увидим:

```
Книга: Балакирев С.М., Python, 3000.78
```

Причем записанный шаблон с распаковкой сработает только в том случае, если кортеж содержит ровно три элемента. Например, если добавить четвертое значение – год издания:

```
cmd = ("Балакирев С.М.", "Python", 3000.78, 2026)
```

то после запуска программы увидим строчку:

```
непонятный формат данных
```

которая говорит нам о том, что первый блок case не сработал. Это произошло по той причине, что в нем явно прописаны три элемента, а в переданном кортеже оказалось четыре. В результате сработал последний блок case.

Но как можно было бы прописать шаблон, чтобы вне зависимости от числа его элементов выбирать только первые три? Очень просто. Снова вспоминаем правила распаковки последовательностей, где мы с вами использовали операцию «*» для упаковки всех оставшихся элементов. Получаем конструкцию вида:

```
match cmd:
    case author, title, price, *_:
        print(f"Книга: {author}, {title}, {price}")
    case _:
        print("непонятный формат данных")
```

Здесь в «*_» будут попадать все элементы, начиная с четвертого. Причем этот шаблон сработает даже если в кортеже будет ровно три элемента. В этом случае коллекция «*_» будет пустой.

Точно так же можно обрабатывать и списки. Например, если вместо кортежа записать список:

```
cmd = ["Балакирев С.М.", "Python", 3000.78, 2026]
```

то, записанная выше конструкция match/case, отработает аналогичным образом.

Если в шаблоне дополнительно требуется ограничить размер кортежа или списка определенным числом элементов, то для этого можно использовать guard, так же, как мы это делали для отдельных переменных. Например, ограничение длины до

шести элементов запишется в виде:

```
match cmd:
    case author, title, price, *_ if len(cmd) < 6:
        print(f"Книга: {author}, {title}, {price}")
    case _:
        print("непонятный формат данных")
```

Для улучшения визуального восприятия шаблона в нем допустимо использовать круглые или квадратные скобки, например, следующим образом:

```
case (author, title, price, *_) if len(cmd) < 6:
```

Или так:

```
case [author, title, price, *_] if len(cmd) < 6:
```

Обратите внимание, здесь эти скобки не создают ни кортежей, ни списков. Это группирующие скобки, чтобы часть шаблона воспринималось как единое целое. Позже мы увидим, для чего они нужны.

Давайте теперь предположим, что в шаблоне дополнительно нужно делать проверку на тип данных элементов списка cmd. В этом случае для каждой отдельной переменной достаточно указать ее тип через ключевое слово as, например, так:

```
case [str() as author, str() as title, float() as price, *_] if len(cmd) < 6:
```

Или сделать такую проверку только у некоторых переменных, например, у цены:

```
case [author, title, float() as price, *_] if len(cmd) < 6:
```

Мало того, если переменная может принимать несколько разных типов, то они также перечисляются через символ ‘|’, например:

```
case [author, title, float() | int() as price, *_] if len(cmd) < 6:
```

Если же нам нужно указать какие-либо ограничения на длину строк автора и заголовка, то в guard можно записать более сложное условие, например:

```
case [str(author), str(title), price, *_] if len(cmd) < 6 and len(author) <
50 and len(title) < 100:
```

И так далее, то есть, здесь можно выполнять все те же самые действия, которые мы рассматривали ранее.

Предположим теперь, что информация о книге может быть представлена в нескольких форматах, например, в таких:

```
cmd = ["Балакирев С.М.", "Python", 2000.78]
cmd = [1, "Балакирев С.М.", "Python", 2000.78, 2022]
```

Требуется в обоих случаях выделять автора, заголовок и цену, а все остальное отбрасывать. Как сформировать обработку таких данных? Конечно, здесь можно

было бы прописать два отдельных шаблона. Но это привело бы к некоторому дублированию кода, что не очень хорошо. Лучше определить один шаблон, объединяющий эти два варианта. Сделать это можно следующим образом:

```
match cmd:
    case (author, title, price) | (_, author, title, price, _):
        print(f"Книга: {author}, {title}, {price}")
    case _:
        print("непонятный формат данных")
```

Здесь использованы группирующие круглые скобки с указанием в них порядка следования данных в анализируемой коллекции. На местах пропускаемых элементов записан символ подчеркивания, а символ '|' означает «или». То есть, общий шаблон сработает, если сработает хотя бы одна группирующая скобка.

Обратите внимание, что количество и имена выделяемых переменных author, title, price в обеих группах должно совпадать. Это строго обязательно. Здесь нельзя, например, во второй группе дополнительно прописать год издания:

```
case (author, title, price) | (_, author, title, price, year):
```

Возникнет синтаксическая ошибка. Должны быть одни и те же переменные во всех группах одного и того же шаблона. Если же во втором варианте дополнительно нужно выделять год издания, то тогда вариант с объединением не подойдет и нужно определять два разных блока case:

```
match cmd:
    case (author, title, price):
        print(f"Книга: {author}, {title}, {price}")
    case (_, author, title, price, year):
        print(f"Книга: {author}, {title}, {price}, {year}")
    case _:
        print("непонятный формат данных")
```

Кроме того, используя текущие знания, можно применить прием, который запрещал бы в полученном операторе match/case обрабатывать кортежи. Для этого достаточно добавить следующий первый блок case:

```
match cmd:
    case tuple():
        print("формат кортежа недопустим")
    case (author, title, price):
        print(f"Книга: {author}, {title}, {price}")
    case (_, author, title, price, year):
        print(f"Книга: {author}, {title}, {price}, {year}")
    case _:
        print("непонятный формат данных")
```

Теперь, если переданная переменная cmd будет иметь тип tuple, то сработает первый блок case, а все другие будут проигнорированы.

Вот в целом такие вариации построения шаблонов проверок возможны при

обработке упорядоченных коллекций в конструкции match/case. Причем, обратите внимание, что обычная строка Python не относится к этому типу данных и делать ее распаковку внутри шаблонов недопустимо.

11.9. Конструкция match/case со словарями и множествами

На данный момент мы с вами научились строить шаблоны проверок в блоках case для отдельных переменных и упорядоченных коллекций типа list и tuple. Следующим шагом рассмотрим возможность обработки коллекций типа dict (словарь) и set (множество). Начнем со словарей.

11.9.1. Обработка словарей

Давайте представим, что имеется словарь с данными некоторого HTML-запроса:

```
request = {'url': 'https://proproprogs.ru/', 'method': 'GET', 'timeout': 1000}
```

Требуется выбрать из него данные ключей «url» и «method», если оба этих ключа присутствуют в словаре. Шаблон проверки можно записать так:

```
match request:
    case {'url': url, 'method': method}:
        print(f"Запрос: url: {url}, method: {method}")
    case _:
        print("неверный запрос")
```

После выполнения программы увидим строчку:

```
Запрос: url: https://proproprogs.ru/, method: GET
```

Обратите внимание, в программе выше используются фигурные скобки для извлечения информации из словаря. Как только они прописываются, шаблон ожидает данные в формате «ключ-значение». Внутри фигурных скобок прописаны два ключа "url" и "method", которые должны присутствовать в request. Если эти ключи действительно присутствуют, то создаются переменные url и method, которые ссылаются на соответствующие значения. Затем, с помощью print, в консоль выводится строка со значениями переменных url и method.

Таким образом, шаблон первого блока case будет срабатывать всякий раз, когда на его вход поступает словарь с ключами "url" и "method". При этом наличие или отсутствие других ключей не играет никакой роли. Например, в словаре request есть еще один ключ "timeout", но шаблон все-равно срабатывает. Это отличает проверку словарей от проверки кортежей и списков, для которых необходимо указывать точное число элементов. Для словарей достаточно наличие указанных ключей и не более того.

Если дополнительно требуется сделать проверку на тип данных в значениях словаря, то это можно реализовать следующим образом:

```
case {'url': str(url), 'method': int() as method}: ...
```

В целях напоминания, проверка типов здесь специально записана двумя разными

способами. Конечно, чаще используется второй подход с ключевым словом «as». Итак, в первом ключе "url" требуется, чтобы была строка, а во втором ключе "method" – целое число. Конечно, сейчас шаблон не сработает, т.к. method является строкой:

```
неверный запрос
```

Но если значение этого ключа изменить на целое число, например, так:

```
request = {'url': 'https://proproprogs.ru/', 'method': 1, 'timeout': 1000}
```

то первый блок case снова сработает:

```
Запрос: url: https://proproprogs.ru/, method: 1
```

Записывая в шаблоне после ключей имена переменных, эти переменные могут принимать произвольные значения заданного типа (или любого типа, если соответствующие проверки отсутствуют). Но также можно в шаблонах прописывать ключи со строго заданными значениями, например, так:

```
case {'url': url, 'method': method, 'timeout': 1000}: ...
```

Тогда оператор case сработает только при значении ключа 'timeout' равным 1000 и будет игнорировать все другие значения. Например, если:

```
request = {'url': 'https://proproprogs.ru/', 'method': 'GET', 'timeout': 2000}
```

то сработает последний блок case, т.к. ключ 'timeout' принимает значение 2000. Но эти два варианта можно скомбинировать в одном шаблоне следующим образом:

```
case {'url': url, 'method': method, 'timeout': 1000} | {'url': url,
'method': method, 'timeout': 2000}: ...
```

Или в данном случае можно записать и так:

```
case {'url': url, 'method': method, 'timeout': 1000 | 2000}: ...
```

Полученный шаблон будет отлавливать значения 1000 и 2000 для ключа 'timeout'. То есть, операция ‘|’ работает так же, как и с упорядоченными коллекциями.

При необходимости в шаблонах обработки словарей можно прописывать guard (защитника). Например, если в запросе request ожидается не более трех ключей, то такое ограничение имеет вид:

```
case {'url': url, 'method': method} if len(request) <= 3: ...
```

И так далее, здесь допустимо использовать все те же самые конструкции, которые были рассмотрены ранее.

Давайте теперь предположим, что требуется построить шаблон проверки словаря, в котором есть ключи «url» и «method» и дополнительно может быть еще не более двух ключей. Как это сделать? Для этого следует воспользоваться операцией упаковки «**» всех остальных данных словаря, кроме указанных ключей,

например, так:

```
match request:
    case {'url': url, 'method': method, **kwargs} if len(kwargs) <= 2:
        print(f"Запрос: url: {url}, method: {method}")
    case _:
        print("неверный запрос")
```

В результате коллекция kwargs будет хранить все остальные ключи с соответствующими значениями, кроме ключей «url» и «method». После этого идет проверка, чтобы словарь kwargs содержал не более двух дополнительных ключей.

Если на вход такого блока match/case подать словарь вида:

```
request = {'id': 2, 'url': 'https://proproprogs.ru/', 'method': 'GET',
'timeout': 2000}
```

то сработает первый шаблон и на экране появится строчка:

```
Запрос: url: https://proproprogs.ru/, method: GET
```

А для словаря, у которого более четырех ключей:

```
request = {'id': 2, 'url': 'https://proproprogs.ru/', 'method': 'GET',
'timeout': 2000, 'date': 100}
```

отработает последний блок case:

```
неверный запрос
```

Причем для срабатывания шаблона расположение ключей в словаре не играет никакой роли. Главное, чтобы они существовали, в данном примере – это ключи «url» и «method», и кроме них может быть не более двух дополнительных ключей.

По аналогии выполняется проверка на наличие строго двух ключей в словаре:

```
case {'url': url, 'method': method, **kwargs} if not kwargs: ...
```

Здесь в защитнике (в условии) проверяется, чтобы словарь kwargs был пустым.

Давайте несколько усложним структуру словаря и предположим, что на вход поступают такие данные:

```
json_data = {'id': 2, 'type': 'list', 'data': [1, 2, 3], 'access': True,
'date': '01.01.2023'}
```

Здесь множество разных ключей, причем один из них "data" может содержать список, но только в том случае, когда ключ "type" принимает значение "list". Шаблон обработки такого словаря можно записать следующим образом:

```
match json_data:
    case {'type': 'list', 'data': lst}:
        print(f"JSON-данные: type-list: {lst}")
    case _:
```

```
        print("неверный запрос")
```

После запуска программы, увидим:

```
JSON-данные: type-list: [1, 2, 3]
```

Здесь первый блок case срабатывает только в том случае, если ключ "type" имеет значение "list" и имеется еще один ключ "data". Ожидается, что переменная lst в этом случае будет ссылаться на список. Конечно, для надежности, можно сделать дополнительную проверку на этот тип данных, например:

```
case {'type': 'list', 'data': list() as lst}: ...
```

Тогда гарантированно, при срабатывании блока case, переменная lst будет принимать тип list. Видите, как легко, просто и элегантно делается обработка заданного формата данных с помощью операторов match/case. Гораздо проще, чем если бы мы использовали условные операторы if/elif/else. В этом и состоит преимущество конструкции match/case перед условными операторами.

Следующий, несколько более сложный пример шаблона обработки входных данных. Пусть, например, имеется словарь вида:

```
json_data = {'id': 2, 'access': True, 'info': ['01.01.2023', {'login':
'123', 'email': 'email@m.ru'}, True, 1000]}
```

в котором нужно проверить наличие ключей 'access' и 'info' и выделить значение ключа 'access' и email из ключа 'info'. Сделать это можно следующим образом:

```
match json_data:
    case {'access': access, 'info': [_, {'email': email}, _, _]}:
        print(f"JSON: access: {access}, email: {email}")
    case _:
        print("неверный запрос")
```

Реализация получается достаточно простой! Здесь буквально берется значение ключа 'access' и нужное значение ключа "email" из списка по ключу 'info'. Если каких-либо данных не будет, то шаблон не сработает. Причем список обязательно должен содержать четыре элемента и во втором элементе храниться словарь с ключом email. Если попробовать сделать то же самое через операторы if/elif/else, то получится гораздо более сложная программа. Тогда как конструкция match/case с легкостью решает поставленную задачу.

11.9.2. Обработка множеств

Помимо словарей есть еще одна похожая коллекция – это множества (set). Давайте посмотрим, как можно обрабатывать этот тип данных в шаблонах блоков case. Предположим, что требуется отбирать только те множества, которые состоят ровно из трех элементов:

```
primary_keys = {1, 2, 3}
```

Если сделать это по аналогии со словарем:

```
match primary_keys:
    case {a, b, c}:
        print(f"Primary Keys: {a}, {b}, {c}")
    case _:
        print("неверный запрос")
```

то получим синтаксическую ошибку. Фигурные скобки применимы только к словарям. Если же убрать фигурные скобки:

```
match primary_keys:
    case a, b, c:
        print(f"Primary Keys: {a}, {b}, {c}")
    case _:
        print("неверный запрос")
```

то ошибки не будет, но в переменной primary_keys будет ожидаться кортеж или список. Поэтому сработает последний блок case с выводом строки:

```
неверный запрос
```

Как же быть с множествами? Вариант, фактически, только один. Прописать явно тип set и присвоить некоторой переменной ссылку на переданное множество:

```
match primary_keys:
    case set() as keys:
        print(f"Primary Keys: {keys}")
    case _:
        print("неверный запрос")
```

После запуска увидим:

```
Primary Keys: {1, 2, 3}
```

При необходимости, дополнительно можно прописать проверку, чтобы множество содержало ровно три элемента:

```
match primary_keys:
    case set() as keys if len(keys) == 3:
        print(f"Primary Keys: {keys}")
    case _:
        print("неверный запрос")
```

По сути, только так можно делать обработку множеств в операторах match/case.

11.10. Некоторые примеры и ограничения конструкции match/case

В целом мы с вами научились прописывать в операторе match/case различные шаблоны для обработки отдельных переменных, упорядоченных коллекций (list, tuple), словарей (dict) и множеств (set). Для лучшего понимания их работы рассмотрим некоторые реальные приложения конструкции match/case, а также увидим ограничения использования этих операторов.

Давайте предположим, что для подключения к БД сформирован следующий словарь с данными:

```
request = {'server': '127.0.0.1', 'login': 'root', 'password': '1234', 'port': 24}
```

Ниже объявим функцию connect_db, в которой будет выполняться проверка данных запроса request и возвращаться результат подключения:

```
def connect_db(connect: dict) -> str:
    match connect:
        case {'server': host, 'login': login, 'password': psw, 'port': port}:
            return f"connection: {host}@{login}.{psw}:{port}"
        case {'server': host, 'login': login, 'password': psw}:
            port = 22
            return f"connection: {host}@{login}.{psw}:{port}"
        case _:
            return "error connection"
```

Вызовем эту функцию следующим образом:

```
result = connect_db(request)
print(result)
```

После выполнения программы, увидим строчку:

```
connection: 127.0.0.1@root.1234:24
```

Глядя на полученную программу, сразу бросается в глаза дублирование кода в первых двух блоках case, а именно, строчка с возвращением f-строки. Давайте это поправим. После блока match запишем команду return в самом конце функции:

```
def connect_db(connect: dict) -> str:
    match connect:
        case {'server': host, 'login': login, 'password': psw, 'port': port}:
            pass
        case {'server': host, 'login': login, 'password': psw}:
            port = 22
        case _:
            return "error connection"

    return f"connection: {host}@{login}.{psw}:{port}"
```

В результате получился первый блок case с оператором pass, который ничего не делает. Оператор pass нужен, так как пустые блоки case (без операторов) прописывать запрещено. Во втором блоке также стоит один оператор, который создает переменную port со значением 22. Если первые два блока case не сработают, то третий отработает в любом случае и функция connect_db вернет строку "error connection". Значит если выполнение программы дошло до последней строки функции:

```
return f"connection: {host}@{login}.{psw}:{port}"
```

то сработал или первый блок case, или второй. В любом случае будут сформированы переменные host, login, psw и port, так как все созданные

переменные внутри оператора match сохраняются и за его пределами. Поэтому формирование f-строки пройдет без ошибок и будет возвращен требуемый результат. Так мы избавляемся от дублирования кода при одинаковом наборе переменных.

Давайте рассмотрим еще один пример. Предположим, информация о книгах может быть представлена в разных форматах, например:

```
book_1 = ('Балакирев', 'Python', 2022)
book_2 = ['Балакирев', 'Python ООП', 2022, 3432.27]
book_3 = {'author': 'Балакирев', 'title': 'Нейросети', 'year': 2020}
book_4 = {'author': 'Балакирев', 'title': 'Keras + Tensorflow', 'price':
5430, 'year': 2020}
```

Наша задача написать функцию, которая бы обрабатывала все эти варианты представления книг и выдавала кортеж в формате:

(автор, название, год, цена)

Для этого объявим функцию book_to_tuple следующим образом:

```
def book_to_tuple(data: dict | tuple | list) -> tuple | None:
    match data:
        case author, title, year:
            price = None
        case author, title, year, price, *_:
            pass
        case {'author': author, 'title': title, 'year': year, 'price': price}:
            pass
        case {'author': author, 'title': title, 'year': year}:
            price = None
        case _:
            return None

    return author, title, year, price
```

А, затем, вызовем ее, например, так:

```
result = book_to_tuple(book_1)
print(result)
```

После выполнения программы, увидим:

```
('Балакирев', 'Python', 2022, None)
```

В нашем примере параметр data функции book_to_tuple может быть разных типов: словарь, кортеж или список. На выходе должен формироваться или кортеж с выделенными данными, или значение None, если переменная data не соответствует записанным шаблонам.

Первый шаблон в блоке case распаковывает кортеж или список, состоящий из трех элементов в формате: автор, название, год издания. Если шаблон срабатывает, то формируется четвертая переменная price со значением None. Следующий (второй) шаблон также ожидает список или кортеж, состоящий, как

минимум, из четырех элементов в порядке: автор, название, год издания, цена. Далее в коллекции могут идти какие-либо другие элементы, они будут игнорироваться. Следующие два блока case обрабатывают словари. Словарь может содержать или три ключа, тогда формируется четвертая переменная price со значением None, или четыре ключа с полным набором необходимых переменных. Если ни один из шаблонов не срабатывает, то последний блок case возвращает значение None, что означает несоответствие форматов. Если же отрабатывает один из предыдущих блоков case, то осуществляется переход к последней команде функции:

```
return author, title, year, price
```

Как видите, все достаточно просто. Но здесь опять же есть небольшое дублирование кода: дважды записана команда `price = None`. По логике ее можно вынести и записать перед оператором match:

```
def book_to_tuple(data: dict | tuple | list) -> tuple | None:
    price = None
    match data:
        case author, title, year:
            pass
        case author, title, year, price, *_:
            pass
        case {'author': author, 'title': title, 'year': year, 'price': price}:
            pass
        case {'author': author, 'title': title, 'year': year}:
            pass
        case _:
            return None

    return author, title, year, price
```

Получим абсолютно тот же самый результат. (Операторы pass не считаются дублированием кода).

Давайте немного усложним задачу и сделаем дополнительную проверку на целочисленный тип переменной year и ее принадлежность диапазону значений (min_year; max_year). Если сделать это «в лоб», то получим следующую реализацию:

```
def book_to_tuple(data: dict | tuple | list, min_year=1800, max_year=3000) ->
tuple | None:

    price = None
    match data:
        case author, title, int(year) if min_year < year < max_year:
            pass
        case author, title, int(year), price, *_ if min_year < year < max_year:
            pass
        case {'author': author, 'title': title, 'year': int(year), 'price': price}
if min_year < year < max_year:
            pass
        case {'author': author, 'title': title, 'year': int(year)} if min_year <
```

```
year < max_year:
            pass
        case _:
            return None

    return author, title, year, price
```

Здесь видим явное дублирование кода в шаблонах. Оставить так или поправить, конечно, решать разработчику, но мы, для примера, скомбинируем оператор match с оператором if следующим образом:

```
def book_to_tuple(data: dict | tuple | list, min_year=1800, max_year=3000) ->
tuple | None:
    price = None
    match data:
        case author, title, int(year):
            pass
        case author, title, int(year), price, *_:
            pass
        case {'author': author, 'title': title, 'year': int(year), 'price':
price}:
            pass
        case {'author': author, 'title': title, 'year': int(year)}:
            pass
        case _:
            return None

    if not (min_year < year < max_year):
        return None

    return author, title, year, price
```

То есть, буквально всю проверку вынесли в одну строчку оператора if. Этот пример показывает, что нас никто не ограничивает в различных комбинациях операторов match и if для написания более удобного и читаемого текста программы.

В заключение рассмотрим еще одну особенность конструкции match/case. Предположим, имеется некая целочисленная переменная:

```
cmd = 10
```

и в операторе match требуется выделить две ситуации со значениями 3 и 5. Очевидно, сделать это можно следующим образом:

```
match cmd:
    case 3:
        print("3")
    case 5:
        print("5")
```

Но опытный разработчик знает, что явно прописывать константы в программе не лучший ход. Правильнее было бы сначала задать константы с числами 3 и 5, а уже потом их использовать в операторах case. Давайте так и сделаем:

```
CMD_3 = 3
CMD_5 = 5

cmd = 3

match cmd:
    case CMD_3:
        print("3")
    case CMD_5:
        print("5")
```

Однако при запуске обнаруживается синтаксическая ошибка. Почему? Дело в том, что в Python нет констант как таковых. Здесь объявлены две переменные CMD_3 и CMD_5, которые указываются в операторах case. В действительности это аналогично тому, что первом case прописывается произвольная переменная, например, такая:

```
match cmd:
    case _:
        print("3")
    case CMD_5:
        print("5")
```

Принципиально ничего не изменилось, просто написали другое имя переменной. Получается, что первый case срабатывает всегда и поэтому второй case приводит к ошибке. Если оставить только один первый блок, то программа всегда будет выводить число 3:

```
match cmd:
    case _:
        print("3")
```

Но это не то, что нам нужно. Поэтому остается вопрос, как использовать константы в блоках case? Существует множество вариантов, мы рассмотрим наиболее простые. Если сделать все на уровне переменных CMD_3 и CMD_5 (которые играют роль констант), то можно пойти на следующую хитрость:

```
match cmd:
    case int(cmd) as x if x == CMD_3:
        print("3")
    case int(cmd) as x if x == CMD_5:
        print("5")
```

Однако выглядит это слишком громоздко. Тут проще было бы воспользоваться операторами if/elif/else. Однако есть другой способ. Язык Python разрешает нам использовать переменные как константы в match/case, если переменная указана через точку. Например, можно объявить произвольный класс с набором следующих атрибутов:

```
class Consts:
    CMD_3 = 3
```

```
    CMD_5 = 5
```

А затем использовать его для обозначения констант:

```
cmd = 3

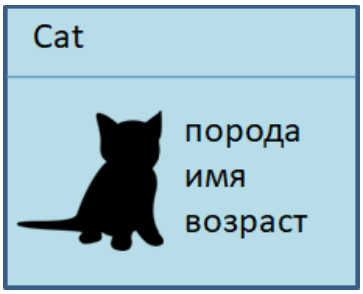
match cmd:
    case Consts.CMD_3:
        print("3")
    case Consts.CMD_5:
        print("5")
```

Тогда все отработает без ошибок.

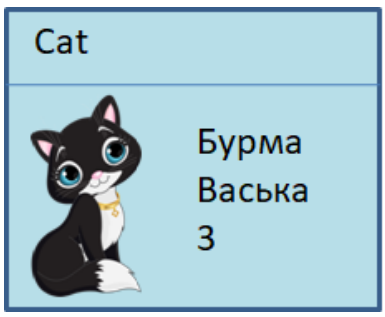
ЧАСТЬ II. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON

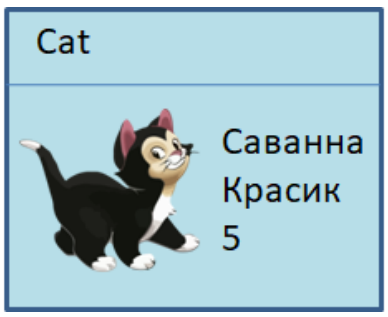
У многих начинающих программистов концепция объектно-ориентированного программирования (ООП) вызывает недоумение и вопрос: зачем она нужна? Я же умею оперировать данными, создавать циклы, прописывать условия, вызывать функции и тем самым писать программы любой сложности? Разве этого мало? Это было достаточно до 1960-х гг. когда, в общем то и зародилась концепция ООП. Даже можно сказать, что вплоть до начала 1990-х программисты могли свободно обходиться без нее, пока она не стала доминирующим направлением и внедрена в самый популярный (на тот момент) язык программирования C++. Так что же это такое и почему сейчас знать ООП должен каждый уважающий себя начинающий программист?

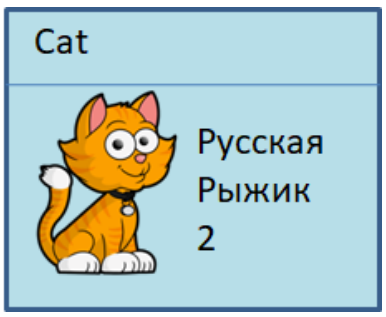
Постараемся ответить на все эти вопросы без привязки к конкретному языку программирования, то есть, в целом понять концепцию ООП – общую для всех языков программирования. Начнем с того, что почти всегда в программах выполняются различные операции с объектами данных. Например, создается программа по учету животных, в частности, котов и кошек. Значит, здесь мы имеем дело с объектом коты (кошки) (Cat). И было бы хорошо все данные, связанные с котами, представлять в программе как единое целое. Как раз это можно сделать с помощью класса:



Класс можно воспринимать, как шаблон, по которому формируются объекты с данными о котах. В этом шаблоне есть три свойства: порода, имя и возраст. Программист сам решает, сколько и какие свойства будет содержать класс. У нас, для примера, такие. А вот конкретные коты – это объекты данного класса. Например:







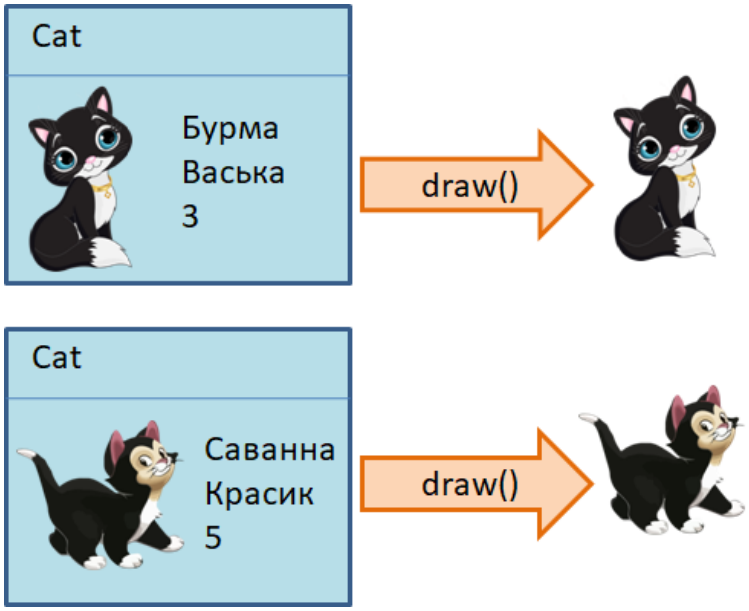
И таких объектов может быть сколько угодно. То есть, объекты создаются по образу и подобию шаблона (класса), в данном случае Cat. Далее в программе

можно работать с этими объектами-котами, как с единым целым.

Вообще, класс может содержать не только свойства, то есть, данные, но и методы – функции, определенные для работы с классом и его объектами. То есть, класс может описывать некий алгоритм, фрагмент программы, присущий именно этому объекту. Например, в классе можно определить метод (функцию), которая бы рисовала кота на экране устройства. Пусть, для примера, она называется draw:



Тогда у каждого объекта этого класса можно вызывать метод draw и на экране будет рисоваться соответствующий кот:



Видите, как удобно можно оперировать объектами на уровне классов. И это только начало.

Разумеется, таких методов в классах может быть сколько угодно. Они, как правило, имеют доступ ко всем свойствам текущего класса и объекта. В результате, в рамках одного класса можно сформировать полноценную программу, которая часто работает независимо с каждым объектом этого класса.

Инкапсуляция

Итак, мы теперь представляем себе, что такое класс и что такое объекты класса. Сделаем следующий шаг и отметим, что класс вообще должен восприниматься как единая, целостная конструкция, все внутренние манипуляции с его данными

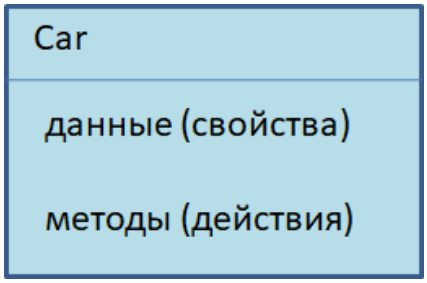
должны быть скрыты в этом классе и в идеале недоступны извне. То есть, программисты, использующие его, не могли бы обращаться ко всем его данным и методам, а только к некоторым – разрешенным.

Это, как с реальными вещами. Например, когда мы покупаем автомобиль, то нам важно его управление, характеристики, но совершенно не задумываемся, как работает двигатель, почему крутятся колеса, из чего сделаны стекла, как взаимодействуют между собой все его агрегаты и прочее и прочее. Пользователю автомобиля важно, чтобы он безупречно работал и довозил его с комфортом из пункта А в пункт Б.



То есть, машина в целом, это некая капсула, которая внутри себя скрывает огромное количество разных узлов, взаимодействующих между собой. И, в идеале, нам совершенно не важен их внутренний алгоритм функционирования. Важен конечный результат – завели и поехали!

Та же идея заложена и в классах. Они вроде капсулы, которая содержит в себе данные и методы. Именно они и определяют внутренний алгоритм функционирования каждого отдельного класса.

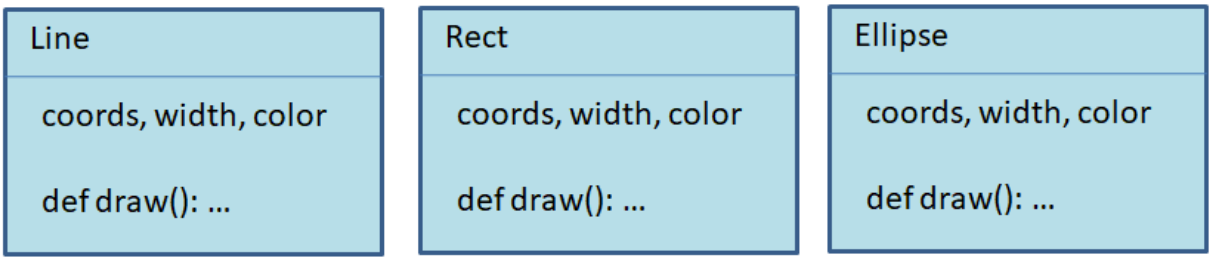


И чтобы пользователь класса не мог случайно или намеренно вмешаться в его внутреннюю работу, данные и методы можно скрывать. То есть, к скрытым данным и методам нельзя обращаться напрямую извне, только внутри класса. Такой механизм в ООП называется **инкапсуляцией**. Благодаря инкапсуляции класс становится единым целым, а работа с ним возможна только через разрешенные (публичные) свойства и методы.

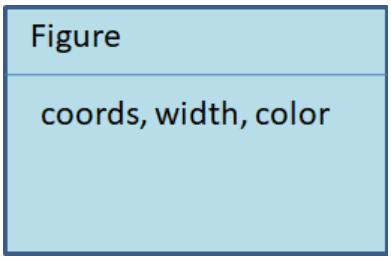
Например, в автомобиле такие разрешенные методы – это руль, коробка передач, педали газа, тормоза и прочее, чем мы можем управлять во время движения. А все остальное, лишнее и скрыто внутри. По большому счету, нам не важно, как именно достигается результат, главное, чтобы ехали! Это и есть идея инкапсуляции в ООП.

Наследование

Следующая важная идея ООП – это наследование классов. Наследование позволяет как бы вынести «за скобки» общее для разных классов. Часто приводят пример с разработкой графического редактора, в котором можно рисовать разные графические примитивы: линии, прямоугольники, эллипсы и другие фигуры. С позиции ООП каждый такой примитив удобно представлять отдельным классом:



Но, очевидно, в этих классах будут общие свойства, например, coords, width и color. Получаем дублирование кода, то есть, нарушение принципа DRY – не повторяйся. Эти общие свойства целесообразно выносить в общий для всех них базовый класс, например, Figure:



Но, чтобы данные и методы класса Figure появлялись в дочерних классах Line, Rect и Ellipse, необходимо унаследовать их от класса Figure. То есть, благодаря механизму наследования классов, можно использовать ранее созданные классы и расширять их функциональность. В нашем примере – это класс Figure, который только хранит данные. А унаследованные, дочерние классы расширяют его функциональность, выполняя рисование конкретных графических примитивов.

Полиморфизм

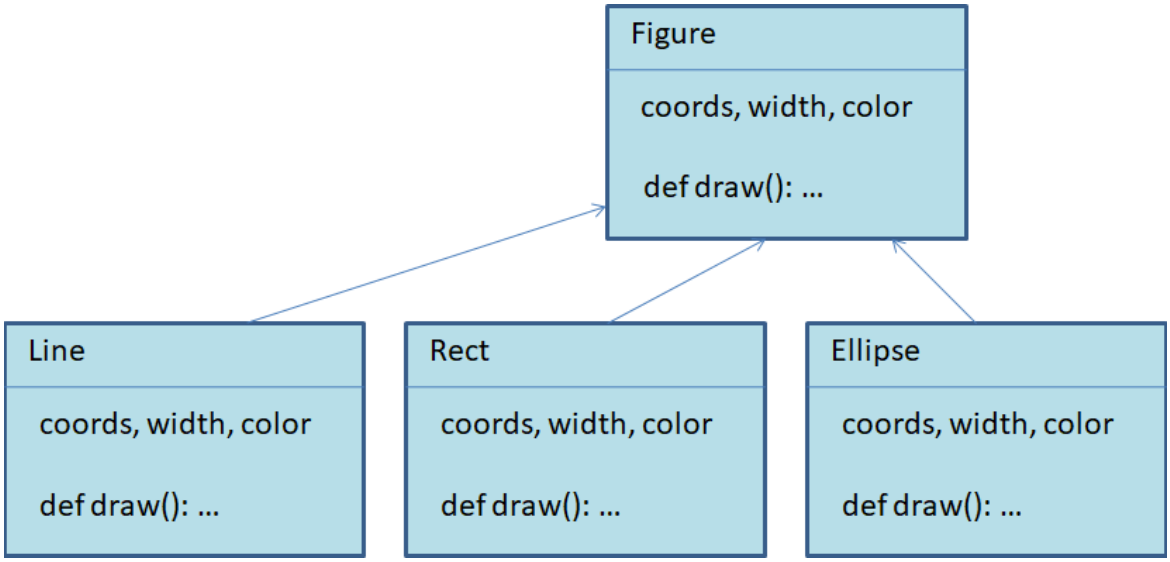
Наконец, последняя важная концепция ООП – полиморфизм. Это необычное слово, применительно к классам и объектам, означает возможность единым образом работать с разными типами данных. Вообще, в программировании различают два вида полиморфизма. Это, так называемый, «Ad hoc» полиморфизм и параметрический полиморфизм. Первый, «Ad hoc» существовал еще до появления ООП и реализовывался через перегрузку функций и приведение различных типов данных. Например, в языке C++ это выглядит следующим образом:

```
double abs(double x)
{
	return (x < 0) ? -x : x
}
```

```
int abs(int x)
{
	return (x < 0) ? -x : x
}
```

Не будем на нем подробно останавливаться, тем более, что в Python он не применяется. А вот второй, параметрический, рассмотрим подробнее.

Вернемся к примеру с графическим редактором, где было определено три класса с одним базовым классом Figure. В базовый класс также добавим метод draw. Вы сейчас поймете почему. Далее, предположим, что для каждого класса были созданы объекты. И наша задача их нарисовать в окне программы. Для этого есть метод draw, который существует в каждом дочернем классе: Line, Rect и Ellipse. Так вот, если бы не было полиморфизма, то мы в программе должны были бы сначала перебрать список из объектов Line, затем, список из объектов Rect и наконец, из объектов Ellipse. Очевидно, это не лучший подход. Представьте, что в будущем у нас будут появляться все новые и новые виды графических примитивов, например, треугольник, пятиугольник, трапеция и т.п. Тогда списков разных типов объектов будет все больше и больше, а программа все сложнее и сложнее. Да и, кроме того, появится дублирование кода.



Но, благодаря параметрическому полиморфизму появляется возможность оперировать разными типами объектов через их единый базовый класс, в нашем примере – это Figure. Достаточно создать список, каждый элемент которого будет иметь тип Figure, и через ссылку на базовый класс вызывать функцию draw. Причем автоматически будет вызван метод draw соответствующего дочернего класса. Так устроено наследование в ООП. В результате, мы имеем единый интерфейс – класс Figure для управления самыми разными типами графических примитивов. Причем в будущем в программу можно добавлять новые графические классы, просто унаследовав их от Figure и они автоматически будут встраиваться в общую логику работы программы. И это невероятно удобная вещь! Благодаря наследованию и полиморфизму мы можем на уровне ООП описывать общую, абстрактную архитектуру работы программы в целом, а потом,

создавая дочерние классы, наполнять эту программу конкретным содержимым. Это буквально переносит нас на совершенно другой, более высокий уровень программирования – целостного, абстрактного описания информационных потоков разрабатываемых приложений. Поэтому не случайно большинство современных паттернов проектирования основаны именно на концепции ООП. Без нее общее написание программных конструкций было бы очень непростым занятием для программистов.

Заключение

Подытоживая все вышесказанное, концепция ООП состоит из трех главных идей:

- инкапсуляции;
- наследования;
- полиморфизма.

Все это мы с вами будем подробно изучать в данной части пособия. Познакомимся со способами программной реализации каждого компонента. И в итоге, вы сможете использовать всю мощь ООП в своих реализациях! Поехали!

Практический курс по Python ООП: https://stepik.org/a/116336

ГЛАВА I. ПЕРВЫЕ ШАГИ В ООП

1.1. Классы и объекты. Атрибуты классов и объектов

Прежде всего, нам с вами нужно научиться определять в своих программах классы, создавать объекты (экземпляры) этих классов, а также добавлять и удалять их атрибуты (то есть, данные).

Рассмотрим простой пример определения класса с именем Point, который предназначен для описания координат точек на плоскости. Для начала запишем его без какого-либо содержимого следующим образом:

```
class Point:
    pass
```

То есть, сначала прописывается ключевое слово «class», после которого следует имя класса (Point) и после двоеточия описывается содержимое класса. Пока здесь стоит один оператор pass, который означает, что класс ничего не содержит. Также обратите внимание, что в соответствии со стандартом PEP8 имя класса принято записывать с заглавной буквы. И, конечно же, называть так, чтобы оно отражало суть этого класса. В дальнейшем мы будем придерживаться этого правила.

Итак, у нас получилось простейшее определение класса с именем Point. Но в таком виде он не особо полезен. Поэтому давайте в нем пропишем два атрибута:

- color – цвет точек;
- circle – радиус точек.

В результате класс Point примет следующий вид:

```
class Point:
    color = 'red'
    circle = 2
```

Обратите внимание, переменные внутри класса обычно называются **атрибутами** класса или его **свойствами**. В дальнейшем мы будем использовать эту терминологию.

Класс Point теперь содержит два атрибута: color и circle. Но как правильно воспринимать эту конструкцию? В действительности, класс образует свое пространство имен, в данном случае с именем Point, в котором находятся две переменные color и circle. Соответственно, к ним можно обращаться, используя синтаксис пространства имен, например, следующим образом:

```
Point.color = 'black'
```

Здесь атрибуту color класса Point присваивается строка "black". А команда вида:

```
res = Point.circle
```

выполняет чтение данных из атрибута circle.

Чтобы увидеть все атрибуты класса можно обратиться к специальной коллекции __dict__:

```
attrs = Point.__dict__
print(attrs)
```

После запуска программы, увидим:

```
{'__module__': '__main__', '__firstlineno__': 1, 'color': 'black',
'circle': 2, '__static_attributes__': (), '__dict__': <attribute '__dict__'
of 'Point' objects>, '__weakref__': <attribute '__weakref__' of 'Point'
objects>, '__doc__': None}
```

Здесь отображается множество служебных встроенных атрибутов и среди них есть два добавленных: color и circle.

Хорошо, как объявлять классы в самом простом варианте, мы с вами увидели. А как создавать объекты этого класса и чем они будут отличаться от самого класса Point? Давайте разберемся. Сначала создадим экземпляр (объект) класса. Для этого достаточно после имени класса Point прописать круглые скобки:

```
a = Point()
```

Будет создан объект, на который ссылается переменная a. Чтобы создать еще один объект этого класса, нужно повторить эту команду:

```
b = Point()
```

В результате, появился еще один объект. По аналогии можно создавать произвольное количество экземпляров класса.

Сразу отмечу, что с помощью функции type можно посмотреть тип данных для переменных a или b:

```
type(a)
```

Увидим строчку:

```
<class '__main__.Point'>
```

которая означает, что объект a – это экземпляр класса Point (имеет тип Point). Поэтому в программах проверку этого типа можно записать в следующем виде:

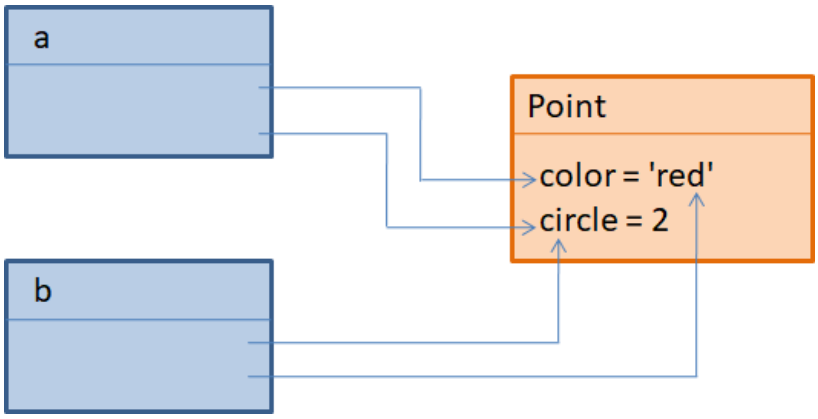
```
res = type(a) == Point   # res = True
```

или, используя знакомую нам функцию isinstance:

```
res = isinstance(a, Point)  # res = True
```

То есть, имя класса выступает в качестве типа данных.

Но давайте детальнее разберемся, что же из себя представляют полученные объекты a и b.



Во-первых, экземпляры a и b образуют свое пространство имен (пространство имен экземпляров класса) и, во-вторых, не содержат никаких собственных атрибутов. Свойства color и circle принадлежат непосредственно классу Point и находятся в нем, а объекты a и b лишь имеют ссылки на эти атрибуты класса. Поэтому они и называются **атрибутами класса**, подчеркивая факт принадлежности к классу. Атрибуты класса – общие для всех его экземпляров. И в этом легко убедиться. Если изменить свойство circle на 1:

```
Point.circle = 1
```

то в обоих объектах это свойство тоже изменится на 1:

```
print(a.circle)    # 1
print(b.circle)    # 1
```

Мало того, если посмотреть коллекцию __dict__ у этих объектов:

```
print(a.__dict__)  # {}
print(b.__dict__)  # {}
```

то они будут пустыми, так как экземпляры сами по себе не содержат никаких атрибутов. Однако, как мы видели, через них можно обращаться к атрибутам класса color и circle.

Но если выполнить присваивание вида:

```
a.color = 'green'
```

то команды:

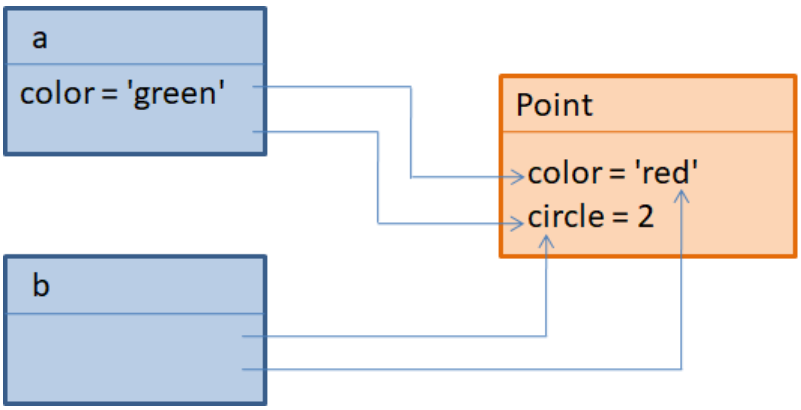
```
print(a.__dict__)  # {'color': 'green'}
print(b.__dict__)  # {}
```

для объекта a выведут словарь с атрибутом color, а объект b останется пустым. Почему так произошло? Дело в том, что при выполнении команды:

```
a.color = 'green'
```

происходит обращение к пространству имен объекта a. В результате оператор присваивания создает новую переменную color (если она отсутствует) в текущей области видимости и атрибут color создастся непосредственно в этом объекте a.

Таким образом, объект a приобретает новое **локальное** свойство (атрибут) color.



Этот момент нужно очень хорошо знать и понимать. Это фундаментальный принцип, на котором строится формирование **атрибутов классов** и **локальных атрибутов** их экземпляров.

1.1.1. Добавление атрибутов в класс. Функции getattr() и setattr()

Мы с вами уже видели, что атрибуты в классе можно прописывать сразу при его объявлении:

```
class Point:
    color = 'red'
    circle = 2
```

Но также добавлять по аналогии с тем, как это делалось с объектами класса, например, так:

```
Point.type_pt = 'disc'
```

В результате класс Point будет содержать три атрибута: color, circle и type_pt. Также новые атрибуты можно добавлять в класс с помощью специальной функции setattr, например, следующим образом:

```
setattr(Point, 'prop', 1)
```

Будет создан атрибут с именем prop и значением 1. Если же функция setattr применяется к существующему атрибуту:

```
setattr(Point, 'type_pt', 'square')
```

то просто меняется его значение на новое. В данном случае атрибут type_pt будет принимать значение в виде строки "square".

Для считывания значения из атрибута, как мы уже знаем, достаточно просто обратиться к нему по имени, например:

```
value = Point.circle  # value = 2
```

Однако при обращении к несуществующему атрибуту класса:

```
v = Point.a   # AttributeError (если ‘a’ не существует)
```

возникает ошибка AttributeError. Этого можно избежать, воспользовавшись специальной функцией getattr для получения значения указанного свойства объекта. Например, так:

```
value = getattr(Point, 'a', False)  # value = False
```

Здесь третий аргумент определяет возвращаемое значение в случае отсутствия указанного атрибута. Однако если функцию getattr вызвать с двумя аргументами:

```
res = getattr(Point, 'a')  # AttributeError (при отсутствии ‘a’)
```

то возникнет все та же ошибка AttributeError. В противном случае (при наличии атрибута класса), функция возвратит значение атрибута:

```
clr = getattr(Point, 'color')   # clr = 'red'
```

Как видим, функция getattr обладает большей гибкостью при обращении к атрибутам класса, чем при прямом доступе через имя класса. Хотя на практике ей пользуются только в том случае, если есть опасность обращения к несуществующим атрибутам. Обычно, все же, применяют синтаксис вида:

```
clr = Point.color
```

1.1.2. Удаление атрибутов из класса. Функции hasattr() и delattr()

Атрибуты также можно удалять из класса. Делается это разными способами. В самом простом случае используется оператор del, например, следующим образом:

```
class Point:
    color = 'red'
    circle = 2
    prop = 1


del Point.prop
```

В результате будет удален атрибут prop из класса Point, если этот атрибут в нем существует. Если попытаться удалить несуществующий атрибут, то возникнет ошибка. Например, повторное выполнение этой же команды:

```
del Point.prop  # AttributeError
```

приведет к ошибке AttributeError. Поэтому перед удалением рекомендуется проверять существование удаляемого свойства. Например, с помощью функции hasattr:

```
if hasattr(Point, 'prop'):
    del Point.prop
```

Эта функция возвращает True, если атрибут найден и False в противном случае. Также удалить атрибут можно с помощью функции delattr, которая работает по

аналогии с оператором del, например, так:

```
delattr(Point, 'type_pt')
```

Обратите внимание, что удаление атрибутов выполняется только в текущем пространстве имен (в данном случае в классе Point). Например, если попытаться удалить свойство color из объекта b:

```
b = Point()
del b.color   # AttributeError
```

то получим ошибку, так как объект b в данном примере не содержит своих локальных свойств и удалять нечего. Если же добавить локальный атрибут:

```
b.color = 255
```

то его, конечно, можно удалить, никакой ошибки не возникнет:

```
del b.color
```

Обратите внимание, что после удаления локального атрибута color в объекте b обращение к атрибуту вернет значение "red", т.к. оно будет браться из класса Point. И это логично, т.к. если свойство не обнаруживается в локальной области объекта, то поиск продолжается в следующей (внешней) области видимости (в классе). Этот момент нужно хорошо понимать и помнить при работе с локальными свойствами объектов.

1.1.3. Атрибуты экземпляров (объектов) классов

Теперь, когда мы знаем, как создаются атрибуты, вернемся к начальной задаче формирования объектов точек на плоскости. Для этого объявим класс с первой строкой в качестве его описания:

```
class Point:
    "Класс для представления координат точек на плоскости"
    color = 'red'
    circle = 2
```

В результате, он будет содержать атрибуты color и circle общие для всех объектов этого класса, и еще один специальный атрибут __doc__ с описанием класса, заданного в виде первой строки. Соответственно, это описание будет доступно через этот атрибут:

```
descr = Point.__doc__
```

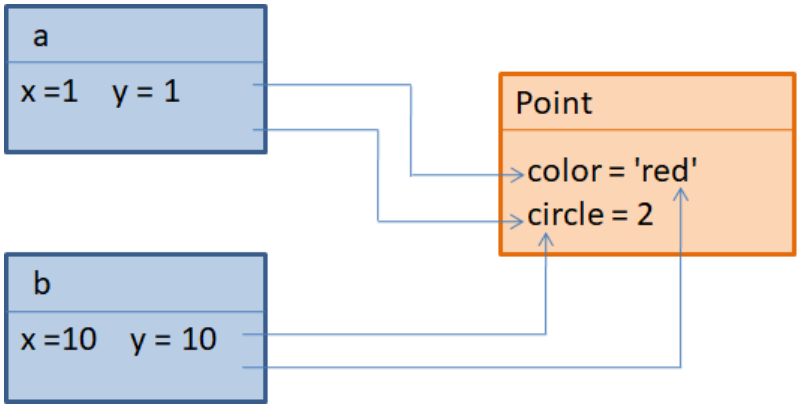
Так обычно делают при создании больших программ, когда в ключевые классы добавляют описания, чтобы в последующем было удобнее возвращаться к ранее написанному коду, корректировать его и использовать, не обращаясь к специальной документации.

Далее положим, что объекты класса Point должны иметь свои уникальные координаты точек (x, y), т.е. являться локальными атрибутами. Сделать это можно

следующим образом. Вначале формируются два объекта a и b, а затем добавляются локальные атрибуты x, y, например, с координатами (1; 2) и (10; 20):

```
a = Point()
b = Point()

a.x = 1
a.y = 2
b.x = 10
b.y = 20
```



В результате, каждый объект будет описывать свою точку с независимыми координатами на плоскости, но общим (для всех объектов) цветом (color) и размером (circle).

1.2. Методы классов. Параметр self

Любой класс помимо атрибутов (в виде переменных) может содержать еще и методы (аналог функций). С их помощью внутри класса можно реализовывать самые разные алгоритмы. То есть, методы – это действия, совершаемые над данными. Именно поэтому в названиях методов принято использовать глаголы (не строгое требование, рекомендация), например:

set_value, get_param, start, stop, и т.п.

А в качестве имен свойств (данных) обычно выбирают существительные, например:

color, size, x, y, и т.п.

Давайте, для примера объявим метод set_coords в классе Point, который будет выводить в консоль сообщение «вызов метода set_coords». Получим:

```
class Point:
    color = 'red'
    circle = 2

    def set_coords(self):
        print("вызов метода set_coords")
```

Здесь сразу бросается в глаза параметр self, который необходим в каждом

стандартном методе класса, и который автоматически прописывает интегрированная среда PyCharm. В чем его предназначение? Если попробовать убрать, то на первый взгляд никаких проблем не возникает. Мало того, этот метод успешно вызывается напрямую из класса Point:

```
class Point:
    color = 'red'
    circle = 2

    def set_coords():                       # метод без параметра self
        print("вызов метода set_coords")


Point.set_coords()
```

Обратите внимание, как записан этот вызов. Сначала указывается имя класса (Point), а затем, через точку прописывается имя метода (или любого другого атрибута этого класса). В конце обязательно нужно записать круглые скобки, так как это оператор вызова функций. После запуска программы, увидим строчку:

```
вызов метода set_coords
```

В результате, у нас с вами получился класс с именем Point, в котором две переменные и один метод. Создадим экземпляр этого класса уже знакомой нам командой:

```
pt = Point()
```

И, как мы с вами ранее говорили, через объект pt можно обращаться ко всем атрибутам класса Point, в том числе и к методу set_coords (имя метода – это тоже атрибут класса). Если записать команду:

```
method = pt.set_coords
```

то переменная method будет ссылаться на объект-функцию set_coords:

```
<bound method Point.set_coords of <__main__.Point object at
0x000001E7D0976F90>>
```

А раз это так, то можно попробовать вызвать ее командой:

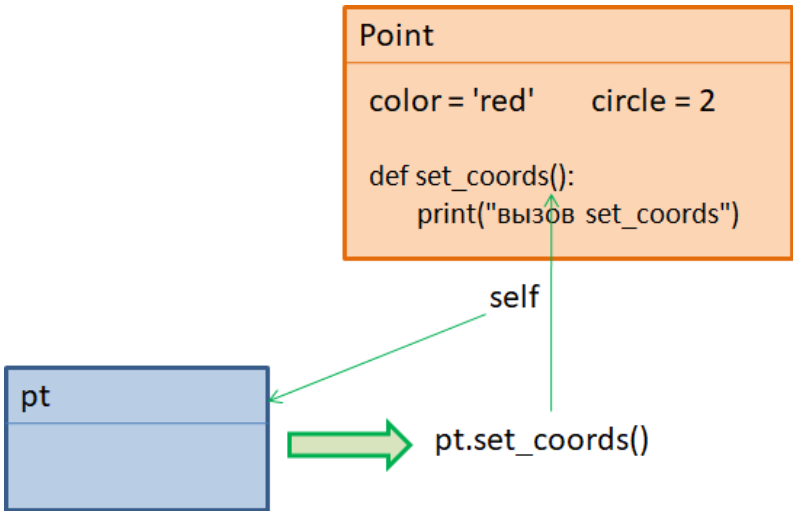
```
pt.set_coords()
```

Однако получим неожиданную, на первый взгляд, ошибку:

```
TypeError: Point.set_coords() takes 0 positional arguments but 1 was given
```

Она говорит о том, что в метод set_coords при его вызове через объект pt передается один аргумент, тогда как в классе он определен без параметров. Спрашивается, что это за аргумент, если при вызове set_coords, в круглых скобках ничего не записано? Дело в том, что когда методы класса вызываются через его объекты, то интерпретатор Python автоматически добавляет первым аргументом

ссылку на объект, из которого этот метод вызывается.



Поэтому, если метод класса предполагается вызывать через его объекты, то дополнительно в нем следует прописать первый параметр, который, обычно, называется self:

```
class Point:
    color = 'red'
    circle = 2

    def set_coords(self):
        print("вызов метода set_coords " + str(self))
```

Теперь команда:

```
pt.set_coords()
```

будет выполнена без ошибок. На экране увидим строчку:

```
вызов метода set_coords <__main__.Point object at 0x0000026312AC6F90>
```

В параметр self интерпретатор Python автоматически передал адрес объекта pt. То есть, при таком вызове, внутри метода set_coords параметр self ссылается на объект pt. Получается, что этот первый параметр в методах класса всегда ссылается на объект, через который был вызван тот или иной метод.

Конечно, после добавления параметра self в метод set_coords, его уже не получится вызвать напрямую через класс Point без указания первого аргумента. Поэтому следующая команда приведет к ошибке:

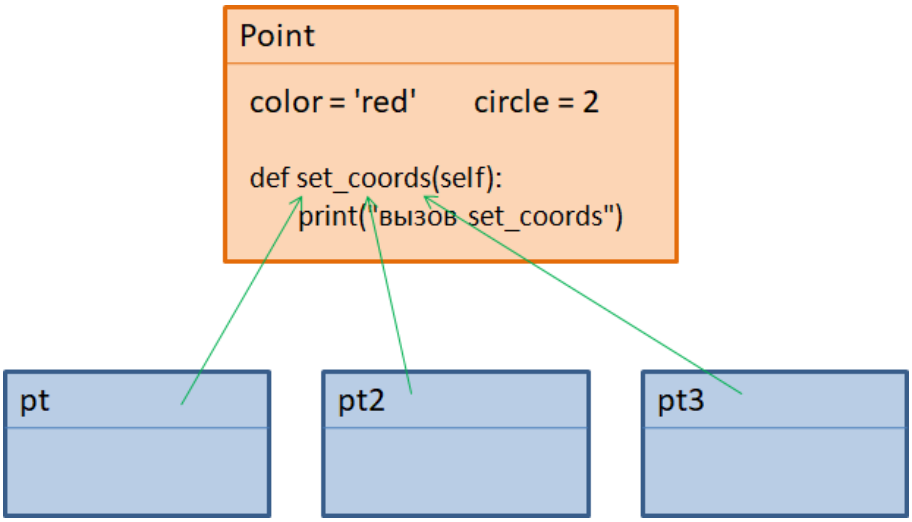
```
Point.set_coords()
```

Это из-за того, что когда метод вызывается напрямую через класс, то Python автоматически не подставляет никаких аргументов. А когда вызов идет через экземпляры класса, то первый аргумент – это всегда ссылка на экземпляр. Этот момент нужно четко понимать. Конечно, вызвать метод set_coords можно и через класс Point, если явно передать ссылку на объект pt:

```
Point.set_coords(pt)
```

Именно это делает Python, когда вызов осуществляется через объект pt.

Хорошо, но зачем понадобилось все так усложнять с методами класса? Дело в том, что метод класса – это тоже атрибут и когда создаются экземпляры класса, то метод становится общим для всех объектов и не копируется в них. Фактически, только благодаря параметру self метод «знает», из какого объекта он был вызван и может поддерживать с ним обратную связь.



Пусть, например, через метод set_coords задаются координаты точек для текущего объекта класса Point следующим образом:

```
class Point:
    color = 'red'
    circle = 2

    def set_coords(self, x, y):
        self.x = x
        self.y = y
```

Здесь в set_coords добавлены два параметра x, y – координаты точки на плоскости. А в самом методе команды `self.x = x; self.y = y` создают (если их не было) локальные атрибуты `x`, `y` непосредственно внутри объекта self, либо меняют значения уже существующих атрибутов `x`, `y`.

Этот момент вы должны уже хорошо знать. Ранее мы с вами говорили, что для любого объекта класса команда:

```
pt.digit = 5
```

создает новый атрибут digit, если его ранее не было, либо меняет значение, если он существует. Причем новый атрибут digit будет создаваться внутри объекта pt, а не внутри класса Point. Поэтому digit, в приведенном примере, принадлежит объекту pt, а не классу Point. Все то же самое справедливо и для команд `self.x = x; self.y = y`. Здесь self ссылается на объект, через который вызывается метод, и

в нем формирует соответствующие локальные атрибуты.

Если теперь вызвать метод set_coords через объект pt и, затем, вывести его локальные атрибуты (коллекция __dict__):

```
pt.set_coords(1, 2)
print(pt.__dict__)
```

то на экране увидим словарь:

```
{'x': 1, 'y': 2}
```

То есть, в объекте pt были созданы два локальных атрибута x, y со значениями 1 и 2. Если в программе создать еще один объект, например:

```
pt2 = Point()
```

и через него вызвать тот же самый метод:

```
pt2.set_coords(10, 20)
print(pt2.__dict__)
```

то увидим на экране два независимых словаря (для объектов pt и pt2):

```
{'x': 1, 'y': 2}
{'x': 10, 'y': 20}
```

Это означает, что локальные свойства x, y создаются независимо внутри каждого объекта, то есть в их пространствах имен. А параметр self каждый раз ссылается на тот объект, из которого был вызван метод set_coords.

Конечно, в классах можно объявлять произвольное количество методов. Например, определим еще один, который будет возвращать координаты точки в виде кортежа значений:

```
class Point:
    color = 'red'
    circle = 2

    def set_coords(self, x, y):
        self.x = x
        self.y = y

    def get_coords(self):
        return (self.x, self.y)
```

И вызовем его командой:

```
print(pt.get_coords())
```

Увидим кортеж:

```
(1, 2)
```

Еще раз обратите внимание, что имя метода – это атрибут класса и к нему можно

обратиться через уже знакомую нам функцию getattr следующим образом:

```
res = getattr(pt, 'get_coords')
print(res)
```

После запуска программы увидим строчку:

```
<bound method Point.get_coords of <__main__.Point object at
0x0000029F9F876F90>>
```

которая говорит нам, что атрибут get_coords является ссылкой на объект-функцию. И далее, можно выполнить вызов, используя переменную res:

```
value = res()   # value = (1, 2)
```

Конечно, так делают относительно редко. Обычно, при вызове методов, используют синтаксис через точку. Этот последний пример еще раз показывает гибкость языка Python и что имена методов – это те же самые атрибуты, только ведут не на данные, а на функции.

1.3. Инициализатор __init__() и финализатор __del__()

В каждом классе языка Python есть набор предопределенных магических (dunder) методов. Магические методы начинаются и заканчиваются двумя подчеркиваниями:

__<имя метода>__

Каждый dunder-метод имеет свое предназначение и речь о них еще впереди. Здесь же мы рассмотрим два таких метода, которые необходимы для полноценной работы при создании и удалении объектов класса:

- __init__(self) – инициализатор объекта класса;
- __del__(self) – финализатор класса.

Метод __init__ вызывается (по умолчанию) сразу после создания экземпляра класса, а второй __del__ срабатывает перед непосредственным удалением объекта. Начнем с первого магического метода __init__.

1.3.1. Инициализатор __init__()

Давайте вернемся к нашему примеру определения класса Point для описания точек на плоскости, который имеет два свойства color и circle и два метода set_coords и get_coords:

```
class Point:
    color = 'red'
    circle = 2

    def set_coords(self, x, y):
        self.x = x
        self.y = y
```

```
    def get_coords(self):
        return (self.x, self.y)
```

Использовать такой класс не очень удобно, так как после создания объекта:

```
pt = Point()          # создание объекта
```

координаты x, y в нем будут отсутствовать. Чтобы они появились дополнительно нужно еще вызвать метод set_coords, например, так:

```
pt.set_coords(1, 2)   # создание локальных атрибутов x, y
```

Было бы хорошо сделать эти действия сразу в момент создания экземпляра класса. Как раз для этого предназначен магический метод __init__. В самом простом виде он может быть записан следующим образом:

```
class Point:
    color = 'red'
    circle = 2

    def __init__(self):
        print("вызов __init__")
        self.x = 0
        self.y = 0
    ...
```

Напомню, что первый параметр self метода __init__ является ссылкой на экземпляр класса, через который метод вызывается.

Теперь, при создании каждого объекта класса Point автоматически будет вызываться этот магический метод __init__ (инициализатор класса). Поэтому после выполнения команды:

```
pt = Point()
```

увидим в консоли строчку:

```
вызов __init__
```

Это означает, что произошел вызов инициализатора и в объекте pt были созданы два локальных свойства x, y с нулевыми значениями. В этом легко убедиться, отобразив словарь __dict__, который содержит все локальные атрибуты объекта:

```
print(pt.__dict__)
```

После выполнения, увидим строчку:

```
{'x': 0, 'y': 0}
```

В деталях полученная программа работает следующим образом. Сначала выполняется создание объекта pt в памяти устройства. Непосредственно перед его созданием вызывается магический метод __new__ (о нем речь еще впереди). Затем, после успешного создания объекта, вызывается магический метод __init__

для начальной инициализации атрибутов созданного объекта. В нашем случае, в объекте pt создаются два локальных атрибута x и y со значениями 0.

Следующим шагом было бы хорошо сделать так, чтобы в момент создания экземпляра класса можно было бы сразу указывать значения координат точки. Для этого в инициализаторе можно прописывать дополнительные параметры, например, так:

```
class Point:
    color = 'red'
    circle = 2

    def __init__(self, a, b):
        self.x = a
        self.y = b

    ...
```

Тогда, при создании объекта класса Point необходимо в круглых скобках прописывать два аргумента для дополнительных параметров a и b (если этого не сделать, то возникнет ошибка):

```
pt = Point(1, 2)
print(pt.__dict__)
```

После запуска программы, увидим значения локальных атрибутов объекта pt:

```
{'x': 1, 'y': 2}
```

В приведенном примере в качестве дополнительных параметров специально указаны имена a и b, чтобы их не путать с локальными атрибутами x, y, которые создаются в самом объекте.

Конечно, на практике, имена параметров в инициализаторе нередко совпадают с именами создаваемых атрибутов и в нашем случае вполне можно записать:

```
class Point:
    color = 'red'
    circle = 2

    def __init__(self, x, y):
        self.x = x
        self.y = y

    ...
```

Это будет одно и то же. Также всегда следует помнить, что метод __init__ – это обычная функция, и параметры в ней можно прописывать с начальными значениями, например, так:

```
class Point:
    color = 'red'
```

```
    circle = 2

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    ...
```

Тогда, при создании объектов, можем вообще не указывать аргументы:

```
pt = Point()     # x=0, y=0
```

либо указать один или два позиционных аргумента:

```
pt = Point(10)      # x=10, y=0
pt = Point(10, 20)  # x=10, y=20
```

либо использовать именованные аргументы:

```
pt = Point(x=5, y=10)     # x=5, y=10
```

Все эти возможности вызова обычных функций сохраняются и при вызове методов классов.

Инициализатор класса следует рассматривать, как обычную функцию, только она автоматически вызывается непосредственно после создания объекта. Следовательно, в теле __init__ можно использовать любые конструкции (операторы) языка Python. Например, вызывать другие методы класса. В нашем случае было бы целесообразно использовать метод set_coords для формирования начальных значений атрибутов x, y. Это можно сделать следующим образом:

```
class Point:
    color = 'red'
    circle = 2

    def __init__(self, x=0, y=0):
        self.set_coords(x, y)      # вызов метода класса в инициализаторе

    def set_coords(self, x, y):
        self.x = x
        self.y = y

    def get_coords(self):
        return (self.x, self.y)
```

Обратите внимание, что вызов set_coords осуществляется через параметр self. Это вполне допустимо, т.к. self ссылается на объект класса Point, для которого был вызван метод __init__. В результате устраняется некоторое дублирование кода в классе Point. И этот прием (вызов других методов текущего класса) можно применять в любых методах.

1.3.2. Финализатор __del__()

Второй похожий магический метод __del__ автоматически вызывается интерпретатором языка Python непосредственно перед уничтожением объекта класса. Поэтому данный метод получил название **финализатор**.

Формально, магический метод __del__ прописывается следующим образом:

```
class Point:
    color = 'red'
    circle = 2

    def __init__(self, x=0, y=0):
        self.set_coords(x, y)

    def __del__(self):
        print("Удаление экземпляра: "+ str(self))

    def set_coords(self, x, y):
        self.x = x
        self.y = y

    def get_coords(self):
        return (self.x, self.y)
```

Метод __del__ должен иметь строго один параметр self, который является ссылкой на удаляемый объект. После запуска программы **возможно** появится строчка об удалении объекта:

```
Удаление экземпляра: <__main__.Point object at 0x000001B867066F90>
```

Почему «возможно», а не «точно»? Для этого нужно понять, как вообще происходит удаление объекта. В программе явно это нигде не прописывается. В действительности, интерпретатор языка Python имеет, так называемый, **сборщик мусора**. Это алгоритм, который отслеживает объекты, и как только они становятся ненужными, удаляет их. Но как он определяет нужный объект или ненужный? Все очень просто. Пока на какой-либо объект ведет хотя бы одна внешняя ссылка, то он считается используемым и сохраняется в памяти. Как только все внешние ссылки пропадают, то сборщик мусора его уничтожает (освобождает память, которую он занимал). Также все объекты автоматически удаляются при завершении программы. В результате программист не может знать, в какой именно момент будет удален тот или иной объект из памяти устройства, а, следовательно, когда будет вызван метод __del__. В ряде случаев он не вызывается вообще. Поэтому в методе __del__ допустимо, например, прописывать логирование жизненного цикла объекта, или освобождение внешних захваченных ресурсов (вроде сокетов) и т.п. На данный момент вам достаточно знать о существовании такого метода, а его использование отложить на потом.

1.4. Магический метод __new__()

На данный момент мы с вами знаем, что сразу после создания объекта класса автоматически вызывается магический метод __init__ (инициализатор объекта). Однако есть еще один магический метод __new__, который также автоматически вызывается непосредственно перед созданием объекта класса. Получается, что стандартный процесс создания экземпляров классов подразумевает последовательный вызов двух магических методов __new__ и __init__. Для чего это сделано и почему недостаточно одного метода __init__? Дело в том, что в практике программирования встречаются самые разнообразные задачи и иногда нужно выполнять определенные действия до создания объектов. Например, реализация известного паттерна Singleton в Python, как раз, использует метод __new__. О нем речь пойдет ниже, а для начала нужно познакомиться с работой самого метода __new__. Давайте пропишем его в классе Point следующим образом:

```
class Point:
    def __new__(cls, *args, **kwargs):
        print("вызов __new__ для " + str(cls))

    def __init__(self, x=0, y=0):
        print("вызов __init__ для " + str(self))
        self.x = x
        self.y = y
```

Первый обязательный параметр cls в методе __new__ – это ссылка на текущий класс Point (не на объект, его еще нет, а на сам класс). Далее прописаны коллекции *args и **kwargs, чтобы в этот метод можно было передавать произвольное количество различных аргументов. Это стандартное определение магического метода __new__ внутри класса. А в теле этого метода пока записан обычный вывод сообщения со значением параметра cls на экран.

Если попробовать создать экземпляр такого класса командой:

```
pt = Point(1, 2)
```

и вывести объект pt на экран:

```
print(pt)
```

то в консоли появятся следующие строчки:

```
вызов __new__ для <class '__main__.Point'>
None
```

Видим, что метод __new__ был вызван, а второй метод __init__ не вызывался. Кроме того, переменная pt равна None. Почему так произошло? Это связано с тем, что метод __new__ в классе Point, по сути, ничего не делает, только выводит сообщение на экран. А должен, во-первых, создавать объект и, во-вторых, возвращать адрес созданного объекта.

Если бы в классе Point не было магического метода __new__, то был бы автоматически вызван аналогичный метод базового класса, который формирует новый объект, вызывает инициализатор (метод __init__) и возвращает адрес созданного объекта. Но, так как мы объявили свой собственный метод __new__, то в нем нужно повторить весь этот функционал. Для этого достаточно вызвать аналогичный метод базового класса следующим образом:

```
class Point:
    def __new__(cls, *args, **kwargs):
        print("вызов __new__ для " + str(cls))
        return super().__new__(cls)     # вызов __new__ из базового класса

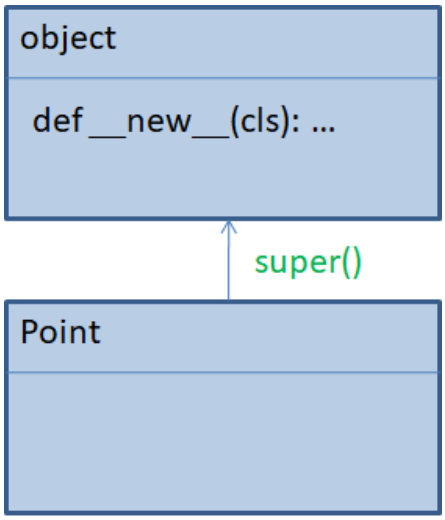
    def __init__(self, x=0, y=0):
        print("вызов __init__ для " + str(self))
        self.x = x
        self.y = y
```

Здесь функция super() возвращает ссылку на базовый класс и через нее выполняется вызов метода __new__ с передачей аргумента cls – ссылки на текущий класс Point.

После запуска измененной программы, в консоли увидим строчки:

```
вызов __new__ для <class '__main__.Point'>
вызов __init__ для <__main__.Point object at 0x0000015EA8406F90>
<__main__.Point object at 0x0000015EA8406F90>
```

То есть, теперь объект создается и вызывается инициализатор. Но подождите! Что это за базовый класс? Класс Point не наследуется ни от какого другого класса! И вообще, мы еще не изучали тему наследования! Да, поэтому, забегая вперед, скажу, что, начиная с версии Python 3, все классы автоматически и неявно наследуются от базового класса object:



Именно из этого базового класса вызывается метод __new__. И, как было сказано выше, если метод __new__ не прописывать в классе Point, то автоматически будет вызываться метод __new__ базового класса object. Этот метод (явно или неявно) всегда должен вызываться при создании нового объекта, иначе объект создан не

будет. При необходимости, его можно переопределять, дополняя стандартную логику работы. То же самое касается всех магических методов. По умолчанию они берутся из базового класса object, но могут быть выборочно переопределены для изменения их поведения в текущем классе. В этом сила базового класса object. В нем уже существует программный код общий для всех классов языка Python, который определяет стандартное поведение объектов и классов.

Возможно, у вас здесь возникает вопрос, зачем нужны списки параметров *args, **kwargs в магическом методе __new__? Они же нигде не используются? Да, в программе явно они нигде не участвуют. Однако необходимы по синтаксису метода __new__. Например, команда:

```
pt = Point(1, 2)
```

создает объект с двумя числовыми аргументами, которые автоматически попадают в коллекцию *args метода __new__. Если бы такого параметра не было:

```
class Point:
    def __new__(cls):
        print("вызов __new__ для " + str(cls))
        return super().__new__(cls)

    def __init__(self, x=0, y=0):
        print("вызов __init__ для " + str(self))
        self.x = x
        self.y = y
```

то предыдущая команда создания объекта привела бы к ошибке:

```
TypeError: Point.__new__() takes 1 positional argument but 3 were given
```

По этой причине прописываются параметры *args и **kwargs, чтобы метод __new__ работал при любом количестве и виде аргументов, то есть, был бы универсальным. Кроме того, эти параметры можно использовать при реализации определенной логики работы переопределенного метода __new__.

1.4.1. Пример использования метода __new__ в реализации учебного варианта паттерна Singleton

В качестве примера использования магического метода __new__ рассмотрим пример реализации довольно известного паттерна проектирования под названием Singleton. Этот паттерн будет представлен в учебном варианте с использованием только текущих знаний.

Давайте предположим, что ведется разработка класса для работы с базой данных (БД). В частности, через этот класс можно будет подключаться к системе управления БД (СУБД), читать, записывать информацию и закрывать соединение:

```
class DataBase:
    def __init__(self, user, psw, port):
        self.user = user
```

```
        self.psw = psw
        self.port = port

    def connect(self):
        print(f"соединение с БД: {self.user}, {self.psw}, {self.port}")

    def close(self):
        print("закрытие соединения с БД")

    def read(self):
        return "данные из БД"

    def write(self, data):
        print(f"запись в БД {data}")
```

Здесь все должно быть вам понятно. Далее полагаем, что в программе должен существовать только один экземпляр этого класса. То есть, одновременно два объекта класса DataBase существовать в одной и той же программе не должны. Для реализации такой логики, как раз, используется паттерн Singleton. Реализуем его для класса DataBase.

Вначале в класс DataBase добавим новый атрибут:

```
class DataBase:
    instance = None

    ...
```

который будет хранить ссылку на ранее созданный объект этого класса. Если объекта нет, то атрибут instance будет принимать значение None.

Чтобы гарантировать создание строго одного экземпляра, добавим в класс DataBase магический метод __new__:

```
class DataBase:
    instance = None

    def __new__(cls, *args, **kwargs):
        if cls.instance is None:
            cls.instance = super().__new__(cls)

        return cls.instance

    ...
```

Работает этот метод следующим образом. Вначале выполняется проверка на существование ранее созданного объекта класса, и если его нет, то есть, атрибут instance равен None, то объект создается путем вызова магического метода __new__ из базового класса. Адрес созданного объекта присваивается атрибуту instance. Если же, при вызове метода __new__ класса DataBase объект ранее уже был создан, т.е. атрибут instance не равен None, то новый объект не создается, а

возвращается адрес первого созданного объекта. В результате можно будет создать только один объект класса DataBase.

Дополнительно в классе DataBase также можно переопределить финализатор __del__, в котором атрибуту класса instance присваивается значение None:

```
class DataBase:
    instance = None

    def __new__(cls, *args, **kwargs):
        if cls.instance is None:
            cls.instance = super().__new__(cls)

        return cls.instance

    def __del__(self):
        DataBase.instance = None

    ...
```

Обратите внимание, как прописано изменение атрибута класса instance в методе __del__. Вначале указывается имя класса DataBase и через точку имя атрибута. Если бы здесь была прописана другая команда:

```
self.instance = None
```

то она бы создала в объекте self новый локальный атрибут instance со значением None, а аналогичный атрибут класса остался бы без изменений. Об этом нюансе мы с вами ранее уже неоднократно говорили.

Конечно, мы не знаем, в какой момент интерпретатор языка Python вызовет финализатор, поэтому нет гарантии, что instance будет равен None сразу после удаления всех ссылок на ранее созданный объект класса DataBase. Поэтому, эту реализацию следует воспринимать, лишь как учебный пример.

Итак, простейший вариант паттерна Singleton сделан. Если теперь попробовать создать два экземпляра класса DataBase:

```
db = DataBase('root', '1234', 80)
db2 = DataBase('root2', '5678', 40)

print(id(db), id(db2), sep='\n')
```

то их идентификаторы ожидаемо будут равны (ведут на один и тот же объект):

```
2562851434384
2562851434384
```

Однако если выполнить метод:

```
db.connect()
db2.connect()
```

то увидим значения:

```
соединение с БД: root2, 5678, 40
соединение с БД: root2, 5678, 40
```

то есть, были использованы аргументы, переданные при повторном создании объекта класса db2. Почему так произошло? Дело в том, что хотя второй объект не создавался, но при вызове команды:

```
db2 = DataBase('root2', '5678', 40)
```

также был вызван и магический метод __init__ с новым набором аргументов. Поэтому локальные свойства объекта изменили свое значение.

Исправить это достаточно просто. Мы знаем, что когда объект создается первый раз, то он не имеет никаких локальных атрибутов, то есть, его коллекция __dict__ пустая. Именно такое условие можно прописать в инициализаторе класса, чтобы создавать и присваивать значения локальным атрибутам только один раз:

```
class DataBase:
    instance = None

    def __new__(cls, *args, **kwargs):
        if cls.instance is None:
            cls.instance = super().__new__(cls)

        return cls.instance

    def __del__(self):
        DataBase.instance = None

    def __init__(self, user, psw, port):
        if not self.__dict__:
            self.user = user
            self.psw = psw
            self.port = port
```

Если теперь запустить программу, то увидим более корректный результат ее работы:

```
соединение с БД: root, 1234, 80
соединение с БД: root, 1234, 80
```

Так, в самом простом варианте, можно реализовать паттерн Singleton, используя магический метод __new__.

1.5. Методы класса (classmethod) и статические методы (staticmethod)

Методы классов в языке Python бывают нескольких видов:

- **Стандартные**. Вызываются через объекты класса; первый параметр self – ссылка на объект, через который был вызван метод.

- **Методы класса** (Class Methods). Вызываются напрямую из класса или через объекты; первый параметр cls – ссылка на класс. Определяются через декоратор classmethod.
- **Статические методы** (Static Methods). Вызываются напрямую из класса или через объекты. Не содержат специальных параметров, т.е. являются обычными функциями внутри класса. Определяются через декоратор staticmethod.

До сих пор в классах мы использовали исключительно стандартные (обычные) методы с первым параметром self. Например:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_coord(self):
        return self.x, self.y


v = Vector(10, 20)
coord = v.get_coord()
print(coord)
```

После запуска программы, в консоли увидим:

```
(10, 20)
```

Здесь все вам должно быть уже понятно. При вызове метода get_coord параметр self ссылается на объект v класса Vector и в операторе return формируется кортеж из значений локальных атрибутов x, y объекта v.

Если метод get_coord вызвать напрямую через класс Vector, то дополнительно ему явно нужно передать первый аргумент для параметра self:

```
coord2 = Vector.get_coord(v)
```

1.5.1. Методы класса (classmethod)

Однако если метод в классе обращается только к атрибутам данного класса и не работает с локальными атрибутами объекта self, то параметр self ему не нужен. Вместо этого первым параметром было бы правильно передавать ссылку cls на текущий класс. Именно так определяются и работают методы уровня класса, которые задаются с помощью стандартного декоратора classmethod.

Давайте на простом примере посмотрим, как объявляются и работают такие методы. Добавим в класс Vector два атрибута и еще один метод класса с именем validate:

```
class Vector:
    MIN_COORD = 0       # минимальное значение координаты
```

```
    MAX_COORD = 100     # максимальное значение координаты

    def __init__(self, x, y):
        self.x = x
        self.y = y

    @classmethod
    def validate(cls, arg):
        return cls.MIN_COORD <= arg <= cls.MAX_COORD

    def get_coord(self):
        return self.x, self.y
```

Обратите внимание на объявление метода validate. Во-первых, он декорирован с помощью classmethod и, во-вторых, первый параметр cls является ссылкой на текущий класс Vector, а не на объект этого класса. Сам же метод validate выполняет проверку принадлежности координаты arg диапазону [MIN_COORD; MAX_COORD].

Наличие параметра cls в методе validate означает, что данный метод может обращаться только к атрибутам текущего класса, но не к локальным свойствам его экземпляров. Мало того, этот метод можно теперь напрямую вызывать из класса, не передавая ссылку в параметр cls, например:

```
res = Vector.validate(5)   # res = True
```

То есть, метод validate можно вызывать, не создавая никаких объектов класса Vector, т.к. этот метод оперирует только данными класса. Во всем остальном метод validate работает абсолютно так же, как и любой другой метод, объявленный в классе.

В нашем примере метод validate может быть полезен для проверки корректности переданных значений координат, например, в инициализаторе:

```
class Vector:
    MIN_COORD = 0
    MAX_COORD = 100

    def __init__(self, x, y):
        self.x = self.y = 0
        if Vector.validate(x) and Vector.validate(y):
            self.x = x
            self.y = y

    @classmethod
    def validate(cls, arg):
        return cls.MIN_COORD <= arg <= cls.MAX_COORD

    def get_coord(self):
        return self.x, self.y
```

Обратите внимание, мы здесь обращаемся к методу класса через пространство

имен Vector. Но можно использовать и параметр self, т.к. все методы класса являются общими для объектов этого класса. Получим:

```
class Vector:
    MIN_COORD = 0
    MAX_COORD = 100

    def __init__(self, x, y):
        self.x = self.y = 0
        if self.validate(x) and self.validate(y):
            self.x = x
            self.y = y

    @classmethod
    def validate(cls, arg):
        return cls.MIN_COORD <= arg <= cls.MAX_COORD

    def get_coord(self):
        return self.x, self.y
```

В этом случае интерпретатор Python при вызове метода validate передаст в первый параметр cls ссылку на текущий класс.

1.5.2. Статические методы класса (staticmethod)

Третий тип методов – это статические методы, которые определяются с помощью декоратора staticmethod. Это методы, которые не имеют доступа ни к атрибутам класса, ни к локальным атрибутам его экземпляров. По сути, это независимая, самостоятельная функция, объявленная внутри класса. Обычно, статические методы добавляют в класс, когда нужно определить обычную функцию, работающую в рамках единой логики класса. Рассмотрим пример объявления и использования статического метода.

Допустим, в программе объявлен класс Vector (см. выше), который описывает произвольные радиус-векторы (исходят из начала координат):

```
rv1 = Vector(10, 20)
rv2 = Vector(15, -2)
```

И требуется по координатам `x`, `y` этих векторов вычислять квадратичную норму (длину вектора в квадрате) по формуле:

$$\|v\|^2 = v_x^2 + v_y^2$$

Для этого объявим в классе Vector статический метод с именем norm2 следующим образом:

```
class Vector:
    MIN_COORD = 0
    MAX_COORD = 100
```

```
    def __init__(self, x, y):
        self.x = self.y = 0
        if self.validate(x) and self.validate(y):
            self.x = x
            self.y = y

    @classmethod
    def validate(cls, arg):
        return cls.MIN_COORD <= arg <= cls.MAX_COORD

    def get_coord(self):
        return self.x, self.y

    @staticmethod
    def norm2(x, y):            # статический метод класса
        return x*x + y*y
```

Как видите, метод norm2 не содержит никаких скрытых параметров, которые бы автоматически заполнялись интерпретатором языка, а только координаты x, y для вычисления квадрата нормы радиус-вектора. Получается, что это некая вспомогательная, сервисная функция, вычисляющая квадратичную норму любого радиус-вектора.

Вызывать метод norm2 можно разными способами. Вне класса, например, командами:

```
res1 = Vector.norm2(10, 20)   # вызов напрямую из класса
res2 = rv2.norm2(10, 20)      # вызов через объект класса
```

А также внутри класса, например, в методе get_coord:

```
class Vector:
    ...

    def get_coord(self):
        return self.x, self.y, self.norm2(self.x, self.y)

    @staticmethod
    def norm2(x, y):
        return x*x + y*y
```

Подведем итог различных типов методов классов. Обычные (стандартные) методы, как правило, вызываются из экземпляров классов и работают, как с локальными атрибутами экземпляров, так и с атрибутами класса. Методы уровня классов (classmethod) имеют доступ только к атрибутам самого класса, в котором объявлены. Наконец, статические методы – это совершенно изолированные функции, которые работают только с переданными в них аргументами, и не имеют доступа ни к атрибутам класса, ни к локальным свойствам его экземпляров. Исходя из этих различий, программист делает выбор в пользу того или иного типа метода.

ГЛАВА II. РЕЖИМЫ ДОСТУПА, СВОЙСТВА, ДЕСКРИПТОРЫ

2.1. Режимы доступа public, private, protected. Сеттеры и геттеры

До сих пор, все объявляемые методы и атрибуты класса (либо объектов) являлись общедоступными, публичными (public). Однако согласно идее инкапсуляции, некоторые элементы класса следует скрывать от прямого доступа извне. Для этого в Python (как и во многих других языках программирования) возможны следующие режимы доступа к данным:

- Публичный режим (**public**). Без ограничений доступа к атрибутам.
- Защищенный режим (**protected).** Для обращения к атрибутам только внутри текущего класса и во всех его дочерних классах.
- Приватный режим (**private)** Для обращения к атрибутам только внутри текущего класса.

Давайте вернемся к примеру класса Point для представления точек на плоскости:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Создавая объекты этого класса:

```
pt = Point(1, 2)
```

имеем полный доступ к его локальным атрибутам `x`, `y` для чтения:

```
coord_x = pt.x
coord_y = pt.y
```

и для записи новых значений:

```
pt.x = 200
pt.y = "coord_y"
```

Причем присвоить можно любые величины, в том числе и строки.

Если в рамках объектов класса Point предполагается в качестве координат `x`, `y` использовать только числовые значения, то было бы правильно ограничить прямой доступ к этим локальным свойствам. Как это сделать? В языке Python определены следующие правила записи имен атрибутов:

- attr – публичный атрибут (режим **public**);
- _attr (с одним подчеркиванием) – защищенный атрибут (режим **protected**);
- __attr (с двумя подчеркиваниями) – приватный атрибут (режим **private**).

Например, в инициализаторе класса Point атрибуты `x`, `y` не содержат ни одного подчеркивания, следовательно, они считаются публичными (public). Если же перед их именами прописать один символ подчеркивания:

```
class Point:
    def __init__(self, x=0, y=0):
        self._x = x
        self._y = y
```

то внутри объекта self будут формироваться защищенные (protected) атрибуты с именами _x, _y. В этом легко убедиться, выполнив строчки:

```
pt = Point(5, 7)
print(pt.__dict__)
```

На экране увидим следующее содержимое словаря __dict__:

```
{'_x': 5, '_y': 7}
```

Но что означает защищенный (protected) режим доступа? Если попытаться напрямую обратиться к ним, как к обычным публичным (public) атрибутам:

```
coord_x = pt._x
coord_y = pt._y
```

то никаких ошибок не возникнет, все по-прежнему работает так, словно это обычные публичные свойства объекта pt. Только интегрированная среда PyCharm такой код подсвечивает желтым предупредительным цветом. Но если нет разницы, то какую роль тогда играет нижнее подчеркивание перед именами переменных? В действительности, в языке Python нижнее подчеркивание всего лишь предостерегает программиста от использования такого «защищенного» свойства за пределами класса, так как впоследствии это может стать причиной непредвиденных ошибок. Например, изменится версия класса и этот protected атрибут может перестать существовать, так как разработчик класса не предполагал доступа к нему извне. Из-за этого, к подобным атрибутам лучше не обращаться напрямую – одно нижнее подчеркивание указывает нам, что это внутренние, служебные переменные.

Давайте теперь посмотрим, как работает режим доступа **private**. Пропишем у локальных атрибутов x, y два подчеркивания:

```
class Point:
    def __init__(self, x=0, y=0):
        self.__x = x
        self.__y = y
```

И выведем локальные свойства объекта этого класса в консоль:

```
pt = Point(5, 7)
print(pt.__dict__)
```

На экране увидим:

```
{'_Point__x': 5, '_Point__y': 7}
```

То есть, внутри объекта pt автоматически были сформированы атрибуты с именами _Point__x и _Point__y, а не __x и __y. Поэтому, при обращении к ним

напрямую:

```
coord_x = pt.__x
coord_y = pt.__y
```

возникнет ошибка AttributeError, что таких свойств не существует. И это действительно так. Публичные имена атрибутов теперь _Point__x и _Point__y. Если их прописать:

```
coord_x = pt._Point__x
coord_y = pt._Point__y
```

то программа выполнится без ошибок. Но, разумеется, делать так строго не рекомендуется, так как два нижних подчеркивания сигнализирует программисту о служебных, приватных переменных, используемых исключительно внутри текущего класса.

В то же время, внутри любого стандартного метода класса доступ к локальным свойствам __x, __y открыт. Для примера, добавим в класс Point метод set_coords, который меняет значения атрибутов __x и __y:

```
class Point:
    def __init__(self, x=0, y=0):
        self.__x = x
        self.__y = y

    def set_coords(self, x, y):
        self.__x = x
        self.__y = y
```

Создадим объект pt класса Point и вызовем через него метод set_coords:

```
pt = Point()
pt.set_coords(1, 2)
```

Все выполнилось без ошибок и приватные атрибуты __x, __y объекта pt стали принимать значения 1 и 2. Чтобы в этом убедиться, добавим в класс Point еще один метод get_coords, а в инициализаторе будем вызывать метод set_coords:

```
class Point:
    def __init__(self, x=0, y=0):
        self.set_coords(x, y)

    def set_coords(self, x, y):
        self.__x = x
        self.__y = y

    def get_coords(self):
        return self.__x, self.__y
```

Вызовем метод get_coords следующим образом:

```
print(pt.get_coords())
```

На экране увидим кортеж:

```
(1, 2)
```

Теперь класс Point содержит два вспомогательных публичных метода set_coords и get_coords, через которые выполняется запись и чтение данных из защищенных (private) атрибутов класса. Такие методы в ООП называются **сеттерами** и **геттерами** или еще **интерфейсными методами**.

Зачем понадобилось в классах создавать приватные атрибуты, да еще и определять дополнительно методы для работы с ними извне? Как мы с вами уже говорили, это пример реализации принципа **инкапсуляции**. Класс в ООП следует воспринимать как некое единое целое. Чтобы случайно не нарушить целостность работы алгоритма внутри этого класса, следует взаимодействовать с ним только через публичные свойства и методы. В этом суть принципа инкапсуляции. Опять же, представьте автомобиль, в котором согласованно работают тысячи узлов. А управление предполагается только через разрешенные интерфейсы: руль, коробка передач, педали газа, тормоза и т.п. Если во время движения вмешиваться напрямую в его узлы, например, будем на ходу спускать воздух из шин, то, наверное, ничего хорошего не получится. То же самое, можно сказать и о программисте, который намеренно обходит запрет и обращается к скрытым атрибутам класса напрямую, а не через публичные сеттеры и геттеры. Именно для этого и существует разделение на публичные (public), защищенные (protected) и приватные (private) атрибуты внутри классов и объектов.

Назначение интерфейсных методов не только передавать значения между приватными или защищенными атрибутами, но и проверять их корректность. Пусть, например, объекты класса Point оперируют только числовыми координатами. Такую проверку целесообразно добавить в сеттер set_coords, например, следующим образом:

```
class Point:
    def __init__(self, x=0, y=0):
        self.__x = self.__y = None
        self.set_coords(x, y)

    def set_coords(self, x, y):
        if type(x) in (int, float) and type(y) in (int, float):
            self.__x = x
            self.__y = y

    def get_coords(self):
        return self.__x, self.__y
```

В методе set_coords перед изменением локальных атрибутов __x, __y выполняется проверка на числовой тип данных (int или float). А в инициализаторе класса начальные значения __x, __y устанавливаются равными None, чтобы эти

локальные атрибуты гарантированно существовали в объекте класса Point.

Если теперь выполнить команду:

```
pt = Point('1', 2)
print(pt.get_coords())
```

то увидим следующий словарь:

```
(None, None)
```

Обратите внимание, что приватными могут быть любые атрибуты класса, в том числе и имена методов. Давайте для примера добавим в класс Point приватный метод с именем __check_value для проверки корректности типов координат. Так как он не предполагает обращения ни к атрибутам класса, ни к атрибутам объектов, то сделаем его статическим и вызовем в сеттере set_coords, получим:

```
class Point:
    def __init__(self, x=0, y=0):
        self.__x = self.__y = None
        self.set_coords(x, y)

    @staticmethod
    def __check_value(x):
        return type(x) in (int, float)

    def set_coords(self, x, y):
        if self.__check_value(x) and self.__check_value(x):
            self.__x = x
            self.__y = y

    def get_coords(self):
        return self.__x, self.__y
```

Этот класс будет работать аналогичным образом. При этом метод __check_value будет доступен по своему имени только внутри класса, но не за его пределами. Например, следующая команда:

```
res = pt.__check_value(7)
```

приведет к ошибке AttributeError.

Конечно, здесь, как и с обычными приватными переменными, можно было бы записать другое внешнее имя этого метода:

```
res = pt._Point__check_value(7)
```

но делать так крайне не рекомендуется. Приватность на то и существует, чтобы обозначать атрибуты, используемые исключительно внутри текущего класса.

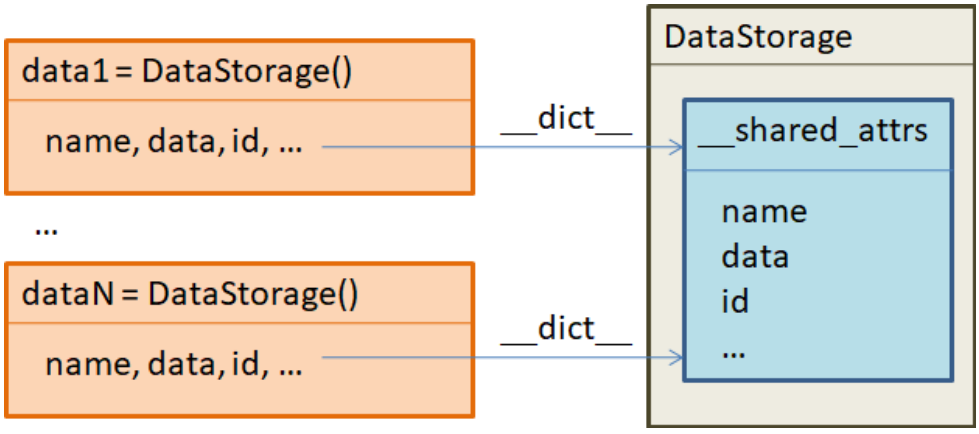
Вот общий принцип определения атрибутов с различными режимами доступа: public, protected и private. При этом всегда следует помнить, что эти режимы предназначены не для защиты от несанкционированного доступа, а всего лишь

сигнализируют программисту о назначении атрибутов: следует ли к ним обращаться за пределами класса, или же использовать исключительно для реализации внутренней логики.

2.2. Паттерн «Моносостояние» (Monostate)

В практике программирования существует большое число различных паттернов (схем, подходов) для построения хороших архитектур программ. Мы с вами рассмотрим один из таких, на примере которого увидим, как в Python легко, гибко и просто происходит управление локальными атрибутами экземпляров класса.

Идея паттерна «Моносостояние» очень проста. Пусть имеется некий класс, например, DataStorage, все объекты которого имели ли бы одни и те же локальные свойства:



Для этого достаточно сделать так, чтобы специальная коллекция __dict__ каждого нового созданного объекта ссылалась на один и тот же словарь. Этот общий словарь удобно определить, как атрибут класса DataStorage, например, следующим образом:

```
class DataStorage:
    __shared_attrs = {
        'name': 'public_data',
        'data': {},
        'id': 1,
    }

    def __init__(self):
        self.__dict__ = self.__shared_attrs
```

Словарь __shared_attrs, для примера, содержит некоторые начальные значения, которые автоматически становятся локальными атрибутами нового созданного объекта. Чтобы ссылка __dict__ сформированного объекта ссылалась на общий словарь __shared_attrs, в инициализаторе записана команда:

```
self.__dict__ = self.__shared_attrs
```

В результате, коллекция __dict__ в каждом новом экземпляре класса будет определяться словарем __shared_attrs и, таким образом, все объекты класса

DataStorage будут иметь единые локальные атрибуты.

Давайте посмотрим, как это будет работать. Создадим первый экземпляр класса DataStorage и выведем его локальные атрибуты на экран:

```
data1 = DataStorage()
print(data1.__dict__)
```

В консоли увидим:

```
{'name': 'public_data', 'data': {}, 'id': 1}
```

Соответственно, ко всем этим свойствам можно обращаться напрямую следующим образом:

```
name = data1.name     # name = public_data
d = data1.data        # d = {}
obj_id = data1.id     # obj_id = 1
```

Если создать еще один экземпляр класса DataStorage с выводом его локальных атрибутов на экран:

```
data2 = DataStorage()
print(data2.__dict__)
```

то увидим все те же самые локальные свойства:

```
{'name': 'public_data', 'data': {}, 'id': 1}
```

Причем изменение любого из них, например:

```
data2.id = 3
```

Приведет к изменению этого атрибута во всех объектах этого класса:

```
print(data1.__dict__)
print(data2.__dict__)
```

Увидим словари:

```
{'name': 'public_data', 'data': {}, 'id': 3}
{'name': 'public_data', 'data': {}, 'id': 3}
```

И это не удивительно, т.к. все объекты используют один и тот же словарь __shared_attrs класса DataStorage.

Также можно создавать новые локальные атрибуты внутри всех объектов, например:

```
data1.descr = 'данные из таблицы car'
```

Соответственно, при отображении словарей __dict__ объектов data1 и data2, увидим строчки:

```
{'name': 'public_data', 'data': {}, 'id': 3, 'descr': 'данные из таблицы car'}
```

```
{'name': 'public_data', 'data': {}, 'id': 3, 'descr': 'данные из таблицы car'}
```

Новый атрибут descr присутствует у обоих объектов.

Это пример реализации паттерна «Моносостояние» на языке Python, который приведен здесь исключительно в учебных целях, чтобы показать, как гибко можно управлять поведением объектов класса, просто меняя ссылку на коллекцию__dict__.

2.3. Объект-свойство property. Декоратор property

При описании различных классов программист часто сталкивается с необходимостью объявлять приватные атрибуты, а затем прописывать публичные сеттеры и геттеры для работы с ними. Например, разрабатывается класс Person для хранения и обработки данных о работниках предприятия. Предположим, он сохраняет данные об имени и возрасте сотрудника следующим образом:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, age):
        self.__age = age
```

Здесь определен сеттер set_age для записи возраста в приватную переменную __age, и геттер get_age для получения значения из переменной __age. Аналогичные методы можно прописать и для второй приватной переменной __name. Они здесь отсутствуют, чтобы не усложнять восприятие программы.

После этого объект класса Person можно создать, например, командой:

```
p = Person('Сергей', 20)
```

А через геттер и сеттер читать и менять возраст сотрудника:

```
p.set_age(35)
print(p.get_age())
```

Все это будет корректно работать. Однако при таком описании класса Person в нем вполне может возникнуть ситуация появления большого количества различных сеттеров и геттеров для обработки соответствующих приватных атрибутов. Например, если добавить еще два для переменной __name:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
```

```
    def get_age(self):
        return self.__age

    def set_age(self, age):
        self.__age = age

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name
```

то в голове придется держать названия имен всех сеттеров и геттеров. Здесь явно напрашивается упрощение описания такого класса. Один из способов заключается в использовании объекта property. Давайте посмотрим, как с его помощью можно улучшить взаимодействие с классом Person.

Итак, для работы с возрастом прописаны методы set_age и get_age. Было бы неплохо вместо этой пары иметь один атрибут, например, age, через который бы выполнялась запись значений:

```
p.age = 35  # запись числа 35 в переменную __age
```

и считывание:

```
value = p.age # чтение значения из переменной __age
```

Соответственно, в первой команде автоматически вызывался бы сеттер set_age, а во второй – геттер get_age. Как раз этот функционал позволяет сформировать объект property следующим образом:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, age):
        self.__age = age

    age = property(get_age, set_age)   # формирование свойства age
```

Как это работает? В действительности, property – это класс, а age – объект этого класса. При создании объекта age, первым аргументом передается ссылка на геттер get_age, а вторым аргументом – ссылка на сеттер set_age. Именно эти методы будут автоматически срабатывать при чтении и записи данных в командах:

```
value = p.age  # сработает геттер get_age
p.age = 43     # сработает сеттер set_age
```

Здесь может возникнуть резонный вопрос, почему команда:

```
p.age = 43
```

не создает новое локальное свойство age внутри объекта p, как это было ранее в программах, а происходит обращение к атрибуту age класса Person? Тут все дело в приоритетах. Если в классе прописан атрибут age как объект-свойство property, то в первую очередь выбирается он, вне зависимости от наличия или отсутствия аналогичного локального атрибута age в экземпляре класса p. В этом легко убедиться, если создать локальное свойство age в объекте p через словарь __dict__, например, следующим образом:

```
p.__dict__['age'] = 'age in object p'
```

А, затем, обратиться к атрибуту age:

```
print(p.age)
```

В консоли увидим число 43, а не строку 'age in object p'. Это означает, что обращение было выполнено к объекту-свойству age класса Person, а не к его локальному атрибуту.

Если же атрибуту age в классе Person присвоить, какое-либо числовое значение:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, age):
        self.__age = age

    age = 100 # теперь age число, а не property(get_age, set_age)
```

то будет отображена строка:

```
age in object p
```

В этом случае приоритеты меняются, и атрибут сначала берется из локальной области объекта p, а если его там нет, то из области видимости класса. Этот момент нужно хорошо запомнить и знать при работе с объектами-свойствами.

2.3.1. Добавление делитера (deleter) в объект-свойство property

Мы с вами знаем, что с любым публичным атрибутом класса или объекта можно выполнять три действия: читать из него значение, менять его значение и удалять атрибут. За первые два действия объекта-свойства age в классе Person отвечают методы get_age и set_age. А вот для удаления не прописано никакого метода. Давайте посмотрим, что будет, если для объекта p класса Person:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, age):
        self.__age = age

    age = property(get_age, set_age)


p = Person('Сергей', 43)
```

выполнить команду удаления атрибута:

```
del p.age
```

Видим следующее сообщение об ошибке:

```
AttributeError: property 'age' of 'Person' object has no delete
```

Она прямо указывает, что для объекта-свойства age не определен метод, который отвечает за удаление. Прописать его можно следующим образом:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, age):
        self.__age = age

    def del_age(self):
        self.__age = None

    age = property(get_age, set_age, del_age)
```

Теперь, при выполнении команды:

```
del p.age
```

никаких ошибок не будет, а значение локального приватного атрибута __age становится равным None.

Так в полной мере можно использовать возможности объекта-свойства property для обработки всех трех действий с различными атрибутами.

2.3.2. Методы getter(), setter() и deleter() объекта-свойства property

В общем случае, объекты класса property можно создавать командами:

```
p1 = property()
p2 = property(get_age)
p3 = property(get_age, set_age)
p4 = property(get_age, set_age, del_age)
```

Объект p1 не содержит никаких методов-обработчиков; объект p2 – только геттер; объект p3 – геттер и сеттер; объект p4 – все три метода, включая делитер.

Здесь возникает вопрос, какой смысл в создании объекта p1, который не имеет ни одного метода? Через него же ничего не получится сделать? Все верно. Однако объект p1 (равно, как и любой объект класса property) содержит следующий набор весьма полезных методов:

- getter() – метод для определения геттера;
- setter() – метод для определения сеттера;
- deleter() – метод для определения делитера.

Воспользоваться ими можно следующим образом:

```
class Person:
    ...

    age = property()
    age = age.getter(get_age)


p = Person('Сергей', 43)
print(p.age)
```

Здесь метод getter связывает метод get_age с получением данных через атрибут age и возвращает новый объект property, который снова присваивается переменной age. В результате выполнения этой программы, увидим значение 43:

```
43
```

Но, при попытке выполнить команду:

```
p.age = 18
```

получим ошибку, что нет метода для сеттера. Поправить это можно очевидным образом, воспользовавшись еще одним методом setter, например, так:

```
class Person:
    ...

    age = property()
    age = age.getter(get_age)
    age = age.setter(set_age)
```

Теперь никаких ошибок не возникает, и на экране высвечивается значение 18. По аналогии используется третий метод deleter.

2.3.3. Декоратор property

Часто в классе формировать объекты-свойства удобнее не через вызов класса property, а через декоратор property. Делается это довольно просто. Перед геттером, который будет играть роль объекта-свойства (обратите внимание, именно перед геттером, а не сеттером или делитером), прописывается декоратор следующим образом:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    @property
    def get_age(self):
        return self.__age
```

В результате, в классе Person появляется атрибут get_age, который является объектом класса property с одним геттером в виде декорированного метода. Поэтому при выполнении следующих строчек:

```
p = Person('Сергей', 43)
print(p.get_age)
```

На экране увидим значение 43.

Однако работает только геттер, т.к. сеттер в классе не определен. Добавить его можно следующим образом:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    @property
    def get_age(self):
        return self.__age

    @get_age.setter
    def get_age(self, age):
        self.__age = age
```

Обратите внимание, что теперь следует прописывать декоратор get_age.setter, а имя метода сеттера должно быть таким же get_age. В результате будет сформирован атрибут get_age, который может использоваться для чтения и записи значений в локальный приватный атрибут __age:

```
p.get_age = 15
print(p.get_age)
```

Конечно, здесь имя get_age совсем не подходящее, поэтому лучше его переименовать просто в age и еще добавить делитер. Получим:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        self.__age = age

    @age.deleter
    def age(self):
        self.__age = None
```

Вот так гибко можно работать с любыми атрибутами класса или его экземпляров через объект-свойство property.

2.4. Пример разработки класса Person

Для лучшего понимания принципа работы с атрибутами классов и его объектов, рассмотрим пример построения класса Person, который будет хранить следующие данные о сотрудниках некоторого предприятия:

- ФИО;
- возраст (целое число от 14 до 120);
- серию и номер паспорта в формате xxxx xxxxxx, где x – цифра (от 0 до 9);
- вес, в кг (вещественное число от 20 и выше).

Первым делом, нужно решить, какие типы данных использовать для представления каждого элемента информации. В рамках нашего примера они будут следующими:

- ФИО – список из трех строк (фамилия, имя, отчество);
- возраст – целое число;
- паспорт – строка с данными паспорта;
- вес – вещественное число.

Все эти данные можно сформировать в инициализаторе следующим образом:

```
class Person:
    def __init__(self, fio, age, ps, weight):
        self.__fio = fio.split()
        self.__age = age
        self.__passport = ps
```

```
        self.__weight = weight
```

Однако эти данные должны соответствовать заданным выше ограничениям. Следовательно, значение каждого атрибута должно удовлетворять заданным условиям. Для проверки указанных условий запишем соответствующие методы. Первый _verify_fio будет выполнять проверку корректности ФИО. Определим его уровня класса (classmethod), т.к. он использует только атрибуты класса, получим:

```
from string import ascii_letters

class Person:
    _S_RUS = 'абвгдеёжзийклмнопрстуфхцчшщьыъэюя- '
    _LETTERS = ascii_letters + _S_RUS + _S_RUS.upper()

    def __init__(self, fio, age, ps, weight):
        self.__fio = fio.split()
        self.__age = age
        self.__passport = ps
        self.__weight = weight

    @classmethod
    def _verify_fio(cls, fio):
        if type(fio) != str:
            raise TypeError("ФИО должно быть строкой")

        f = fio.split()
        if len(f) != 3:
            raise TypeError("Неверный формат записи ФИО")

        if len(fio.strip(cls._LETTERS)) != 0:
            raise ValueError("В ФИО можно использовать только буквенные
символы, пробел и дефис")
```

Обратите внимание, что вначале импортирована коллекция символов латинского алфавита из модуля strings, представленная в виде строки ascii_letters:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

А в самом классе Person определены вспомогательные атрибуты _S_RUS и _LETTERS. Последний содержит набор допустимых символов в ФИО.

Ниже прописан метод уровня класса _verify_fio, который первым делом проверяет, что переменная fio должна иметь тип str. Если это не так, то генерируется исключение (ошибка) командой:

```
raise TypeError("ФИО должно быть строкой")
```

О генерации исключений мы с вами еще будем говорить. Здесь достаточно знать, что, как только в программе срабатывает оператор raise, то (в данном случае) выполнение программы завершается с отображением типа ошибки (например, TypeError) и записанного в нем сообщения.

Давайте сразу пропишем метод _verify_fio в инициализаторе для проверки переданного значения fio:

```
class Person:
    _S_RUS = 'абвгдеёжзийклмнопрстуфхцчшщьыъэюя- '
    _LETTERS = ascii_letters + _S_RUS + _S_RUS.upper()

    def __init__(self, fio, age, ps, weight):
        self._verify_fio(fio)      # проверка корректности параметра fio

        self.__fio = fio.split()
        self.__age = age
        self.__passport = ps
        self.__weight = weight

    ...
```

Если проверка проходит успешно, то никаких исключений не возникнет и программа продолжает свою работу, иначе она завершается.

Для примера создадим объект с верной записью ФИО командой:

```
p = Person('Балакирев Сергей Михайлович', 30, '1234 567890', 80.0)
```

После запуска программа выполняется успешно без каких-либо ошибок. Но если записать неверный формат ФИО, например:

```
p = Person('Сергей Михайлович', 30, '1234 567890', 80.0)
```

то выполнение программы прерывается с отображением ошибки:

```
TypeError: Неверный формат записи ФИО
```

Добавим еще три аналогичных метода для проверки корректности возраста, веса и паспортных данных. Все они будут статическими, т.к. не обращаются ни к каким внешним данным:

```
from string import ascii_letters

class Person:
    _S_RUS = 'абвгдеёжзийклмнопрстуфхцчшщьыъэюя- '
    _LETTERS = ascii_letters + _S_RUS + _S_RUS.upper()

    def __init__(self, fio, age, ps, weight):
        self._verify_fio(fio)
        self._verify_age(age)
        self._verify_ps(ps)
        self._verify_weight(weight)

        self.__fio = fio.split()
        self.__age = age
        self.__passport = ps
        self.__weight = weight
```

```
    @classmethod
    def _verify_fio(cls, fio):
        if type(fio) != str:
            raise TypeError("ФИО должно быть строкой")

        f = fio.split()
        if len(f) != 3:
            raise TypeError("Неверный формат записи ФИО")

        if len(fio.strip(cls._LETTERS)) != 0:
            raise ValueError("В ФИО можно использовать только буквенные
символы, пробел и дефис")

    @staticmethod
    def _verify_age(age):
        if type(age) != int or age < 14 or age > 120:
            raise TypeError("Возраст должен быть целым числом: [14; 120]")

    @staticmethod
    def _verify_weight(w):
        if type(w) != float or w < 20:
            raise TypeError("Вес вещественное число от 20 и выше")

    @staticmethod
    def _verify_ps(ps):
        if type(ps) != str:
            raise TypeError("Паспорт должен быть строкой")

        s = ps.split()
        if len(s) != 2 or len(s[0]) != 4 or len(s[1]) != 6:
            raise ValueError("Неверный формат паспорта")

        for p in s:
            if not p.isdigit():
                raise ValueError("Данные паспорта должны быть числами")
```

Теперь, при создании каждого нового объекта класса Person будут выполняться проверки корректности переданных данных, и в случае каких-либо неточностей программа завершит свою работу с выводом ошибки на экран.

Последнее, что добавим в класс Person в рамках этого примера – это возможность взаимодействия с приватными локальными атрибутами через объекты-свойства property для чтения и записи данных. Здесь все достаточно просто и очевидно. Для доступа к атрибуту __fio определим аналогичное свойство fio через декоратор property следующим образом:

```
class Person:
    ...

    @property
```

```
    def fio(self):
        return self.__fio

    @fio.setter
    def fio(self, fio):
        self._verify_fio(fio)
        self.__fio = fio.split()
```

Теперь для записи и считывания значений в поле __fio можно использовать команды вида:

```
p = Person('Балакирев Сергей Михайлович', 30, '1234 567890', 80.0)
p.fio = 'Гвидо Ван Россум'
print(p.fio)    # ['Гвидо', 'Ван', 'Россум']
```

По аналогии добавляются объекты-свойства для всех остальных приватных атрибутов. В результате получим:

```
class Person:
    ...

    @property
    def fio(self):
        return self.__fio

    @fio.setter
    def fio(self, fio):
        self._verify_fio(fio)
        self.__fio = fio.split()

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        self._verify_age(age)
        self.__age = age

    @property
    def weight(self):
        return self.__passport

    @weight.setter
    def weight(self, w):
        self._verify_weight(w)
        self.__weight = w

    @property
    def passport(self):
        return self.__passport

    @passport.setter
```

```
    def passport(self, ps):
        self._verify_ps(ps)
        self.__passport = ps
```

Теперь класс Person содержит базовый функционал для хранения указанных данных о сотрудниках с проверкой их корректности. Воспользоваться возможностями этого класса можно, например, так:

```
p = Person('Балакирев Сергей Михайлович', 30, '1234 567890', 80.0)
p.fio = 'Гвидо Ван Россум'
p.age = 72
p.passport = '4567 123456'
p.weight = 78.3
print(p.__dict__)
```

После запуска программы увидим следующий набор локальных атрибутов объекта p:

```
{'_Person__fio': ['Гвидо', 'Ван', 'Россум'], '_Person__age': 72,
'_Person__passport': '4567 123456', '_Person__weight': 78.3}
```

В данном случае, везде были указаны корректные данные, поэтому программа выполнилась без ошибок.

Также в инициализаторе можно воспользоваться объектами-свойствами, чтобы упростить текст программы:

```
class Person:
    _S_RUS = 'абвгдеёжзийклмнопрстуфхцчшщьыъэюя- '
    _LETTERS = ascii_letters + _S_RUS + _S_RUS.upper()

    def __init__(self, fio, age, ps, weight):
        self.fio = fio
        self.age = age
        self.passport = ps
        self.weight = weight

    ...
```

Здесь сразу через объекты-свойства осуществляется проверка данных и создание приватных локальных атрибутов в текущем объекте self. В результате устраняются дублирующие вызовы методов проверки корректности данных.

2.5. Дескриптор данных (data descriptor)

Полученный в предыдущем параграфе класс Person, получился довольно объемным и неудобным для последующего восприятия и редактирования. Во многом это связано с набором однотипных объявлений объектов-свойств property для каждого приватного атрибута. А их было всего четыре! Представьте теперь аналогичный класс, который оперирует десятком приватных атрибутов через объекты property. Здесь, очевидно, нужен другой подход для обработки данных, который бы позволил сократить размер текста программы и при этом сохранить

прежнюю функциональность класса. Как раз для этого хорошо подходит инструмент под названием **дескриптор**.

Принцип работы дескрипторов проще всего понять на конкретном примере. Давайте предположим, что задается класс Point3D для представления точек в трехмерном пространстве, с целочисленными координатами x, y, z:

```
class Point3D:
    def __init__(self, x, y, z):
        self._x = x
        self._y = y
        self._z = z
```

Теперь, вместо определения множества объектов-свойств property для работы с локальными атрибутами _x, _y, _z, будем использовать дескрипторы. Что такое дескриптор? Это обычный класс, который, в общем случае, может содержать следующие магические методы:

- __set_name__(self, owner, name) – для формирования внутри объекта дескриптора имени атрибута, которое будет использовано для хранения данных;
- __get__(self, instance, owner) – геттер для получения информации из атрибута с ранее сформированным именем (в методе __set_name__);
- __set__(self, instance, value) – сеттер для записи данных в атрибут с ранее сформированным именем (в методе __set_name__);
- __delete__(self) – делитер, срабатывает при удалении атрибута.

В нашем примере в качестве дескриптора будет выступать класс с именем Integer (имя придумывается программистом), в котором определим следующие три магических метода:

```
class Integer:
    def __set_name__(self, owner, name):
        print(f'__set_name__: {owner = }, {name = }')

    def __get__(self, instance, owner):
        print(f'__get__: {instance = }, {owner = }')

    def __set__(self, instance, value):
        print(f'__set__: {instance = }, {value = }')
```

При срабатывании каждого метода выполняется вывод на экран информации об его параметрах. Чтобы увидеть их в действии, объект этого дескриптора нужно создать в классе Point3D на уровне атрибута класса, например, так:

```
class Point3D:
    x = Integer()

    def __init__(self, x, y, z):
        self._x = x
        self._y = y
```

```
        self._z = z
```

Затем, создадим объект этого класса и выполним обращение к объекту x дескриптора Integer:

```
pt = Point3D(1, 2, 3)
pt.x = 5   # запись значения через дескриптор
x = pt.x   # чтение данных через дескриптор
```

После выполнения программы, на экране увидим следующие строчки:

```
__set_name__: owner = <class '__main__.Point3D'>, name = 'x'
__set__: instance = <__main__.Point3D object at 0x000001918F4970E0>, value
= 5
__get__: instance = <__main__.Point3D object at 0x000001918F4970E0>, owner
= <class '__main__.Point3D'>
```

Что они означают? Смотрите, при создании объекта дескриптора Integer в классе Point3D:

```
class Point3D:
    x = Integer()

    ...
```

автоматически был вызван магический метод __set_name__, у которого параметр owner ссылается на класс Point3D, а параметр name содержит строку 'x'. Почему именно 'x'? Это имя создаваемого объекта-дескриптора внутри класса Point3D. Если бы объект формировался командой:

```
class Point3D:
    coord_x = Integer()

    ...
```

то параметр name содержал бы строку 'coord_x'. И так далее. Зачем это нужно мы увидим далее.

После формирования объекта pt класса Point3D происходит обращение к атрибуту x, то есть, к объекту-дескриптору Integer:

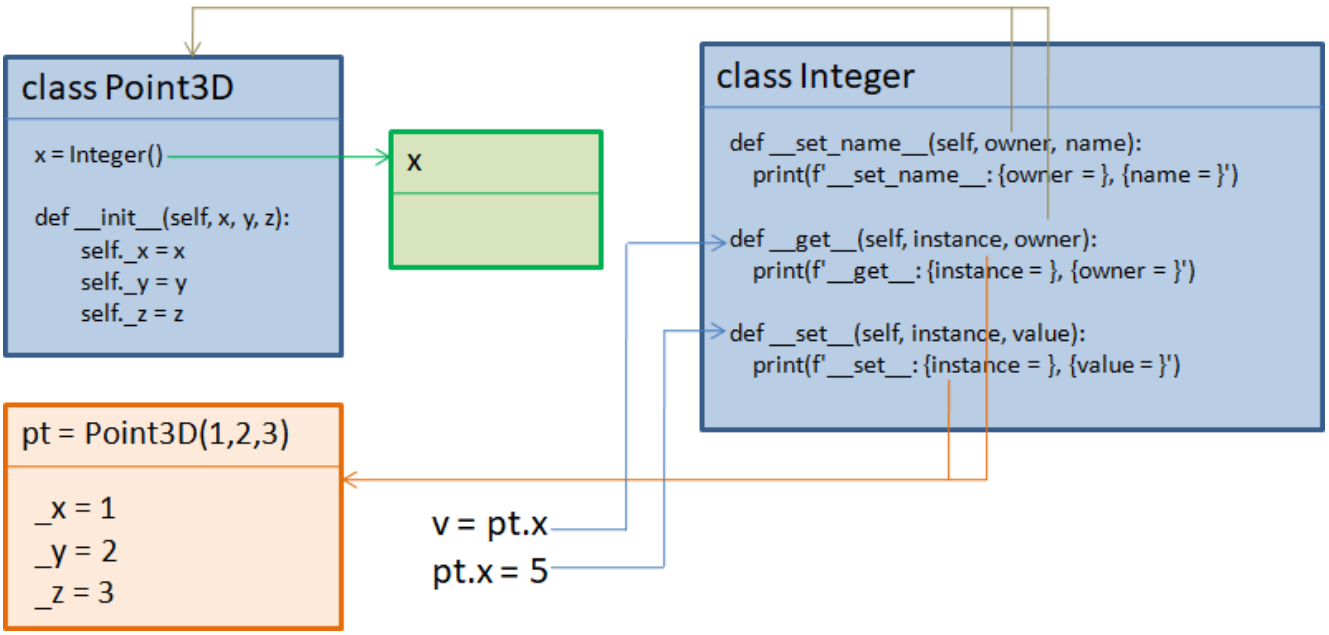
```
pt.x = 5   # запись значения через дескриптор
```

В результате в классе Integer автоматически срабатывает магический метод __set__, у которого параметр instance ссылается на объект pt, а параметр value равен 5, т.е. содержит присваиваемое значение.

Наконец, при попытке чтения данных через дескриптор x:

```
x = pt.x   # чтение данных через дескриптор
```

срабатывает магический метод __get__, у которого параметр instance ссылается на объект pt, а параметр owner – на класс Point3D.



Используя логику работы дескриптора Integer, нужно сделать так, чтобы он формировал локальный атрибут (с некоторым именем) в каждом объекте класса Point3D, сохранял в этот атрибут данные и читал их оттуда. Для этого вначале, при создании объекта x командой:

```
class Point3D:
    x = Integer()

    ...
```

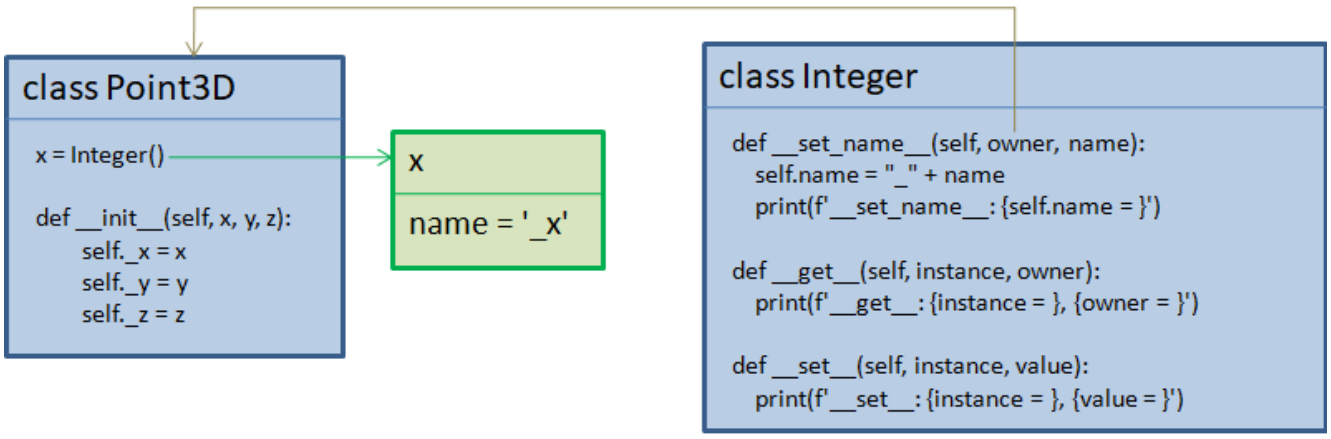
нужно сформировать имя локального атрибута. Так как здесь срабатывает магический метод __set_name__, то в нем пропишем следующую реализацию:

```
class Integer:
    def __set_name__(self, owner, name):
        self.name = "_" + name
        print(f'__set_name__: {self.name = }')
    ...
```

В результате, в объекте x будет формироваться локальный атрибут name со строкой `"_" + name`. Это имя в дальнейшем будет использовано для формирования локального атрибута уже внутри объектов класса Point3D. Чуть позже мы это увидим. Если же сейчас запустить полученную программу, то в консоли появятся строчки:

```
__set_name__: self.name = '_x'
__set__: instance = <__main__.Point3D object at 0x00000261013E70E0>, value
= 5
__get__: instance = <__main__.Point3D object at 0x00000261013E70E0>, owner
= <class '__main__.Point3D'>
```

То есть, внутри объекта-дескриптора x было сформировано имя '_x' для локального атрибута объектов класса Point3D.



Следующим шагом пропишем реализации в двух оставшихся магических методах дескриптора Integer следующим образом:

```
class Integer:
    def __set_name__(self, owner, name):
        self.name = "_" + name
        print(f'__set_name__: {self.name = }')

    def __get__(self, instance, owner):
        return getattr(instance, self.name, None)

    def __set__(self, instance, value):
        setattr(instance, self.name, value)
```

При выполнении команды:

```
v = pt.x
```

будет вызваться метод __get__, у которого параметр instance ссылается на объект pt. В объекте pt должен существовать локальный атрибут с именем '_x' (хранится в объекте-дескрипторе в поле name). С помощью стандартной функции getattr из объекта instance выполняется чтение локального атрибута с именем '_x' (используя в дескрипторе поле name). Если указанный атрибут отсутствует, то возвращается значение None.

Другая команда:

```
pt.x = 5
```

автоматически вызывает метод __set__, в котором параметр instance также ссылается на объект pt. Поэтому при выполнении функции:

```
setattr(instance, self.name, value)
```

происходит либо формирование локального атрибута с именем '_x' в объекте pt и значением value, либо (если атрибут уже существует) изменение существующего атрибута на значение value.

В результате мы с вами сформировали в базовом варианте дескриптор Integer для работы с координатами класса Point3D. Воспользоваться им можно следующим

образом:

```
class Point3D:
    x = Integer()
    y = Integer()
    z = Integer()

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

Здесь формируются три объекта-дескриптора – по одному на каждую координату. В инициализаторе они сразу используются для формирования соответствующих локальных атрибутов внутри создаваемого объекта класса Point3D.

После выполнения следующих команд:

```
pt = Point3D(1, 2, 3)
pt.x = 5
print(pt.x)
print(pt.__dict__)
```

На экране увидим строчки:

```
__set_name__: self.name = '_x'
__set_name__: self.name = '_y'
__set_name__: self.name = '_z'
5
{'_x': 5, '_y': 2, '_z': 3}
```

То есть, каждый дескриптор формирует свое имя локального атрибута, и все эти атрибуты успешно создаются внутри объекта pt.

Вот общий принцип работы дескрипторов. Достаточно один раз прописать в них логику обработки локальных свойств и затем многократно применять внутри класса (в нашем примере, в классе Point3D). На первый взгляд это может показаться сложным и запутанным. Но такое впечатление складывается из-за обилия новой информации. Если внимательно, детально во всем разобраться, то все становится предельно простым и понятным.

Конечно, дескриптор Integer должен дополнительно проверять присваиваемые значения на целый тип данных int. Сделать это достаточно просто, например, так:

```
class Integer:
    @staticmethod
    def _verify_coord(coord):
        if type(coord) != int:
            raise TypeError("Координата должна быть целым числом")

    def __set_name__(self, owner, name):
        self.name = "_" + name
        # print(f'__set_name__: {self.name = }')
```

```
    def __get__(self, instance, owner):
        return getattr(instance, self.name, None)

    def __set__(self, instance, value):
        self._verify_coord(value)
        setattr(instance, self.name, value)
```

В результате, если присваиваемое значение value окажется не целочисленным, то будет сгенерирована ошибка TypeError. Например, при выполнении команды:

pt = Point3D(1, 2.7, 3)

программа завершится с ошибкой:

```
TypeError: Координата должна быть целым числом
```

2.6. Дескриптор без данных (non-data descriptor)

Рассмотренный дескриптор Integer относится к, так называемым, дескрипторам данных, так как он позволяет читать и записывать новые значения в атрибуты. Если же дескриптор служит только для чтения данных, то он относится к non-data descriptor (дескрипторам без данных). Очевидно, такие дескрипторы не содержат магических методов __set__ и __delete__.

На первый взгляд, кажется, что нет особой причины разделять дескрипторы на эти два вида. Однако существует принципиальная разница в их работе, которая заключается в приоритетах доступа к атрибутам. О чем здесь речь? Давайте рассмотрим различия на конкретном примере.

Добавим в программу, созданную в предыдущем параграфе, еще один дескриптор типа non-data descriptor. Например, такой:

```
class ReadInteger:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, owner):
        return getattr(instance, self.name, None)
```

И воспользуемся им в классе Point3D:

```
class Point3D:
    x = Integer()  # формирование и работа с атрибутом _x
    y = Integer()
    z = Integer()

    xr = ReadInteger('_x')  # чтение локального атрибута _x

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
```

```
        self.z = z
```

Если выполнить команды:

```
pt = Point3D(1, 2, 3)
print(pt.xr)
```

то на экране увидим значение 1 атрибута _x:

```
1
```

Однако объект-дескриптор xr работает только чтения данных. А что будет, если выполнить команду:

```
pt.xr = 5
```

Ошибки никакой не возникнет, но в объекте pt будет создано новое локальное свойство с именем pt. Если вывести все локальные атрибуты:

```
print(pt.__dict__)
```

то в консоли отобразится словарь:

```
{'_x': 1, '_y': 2, '_z': 3, 'xr': 5}
```

Как видите, здесь появился локальный атрибут xr со значением 5. Мало того, при обращении к этому локальному атрибуту:

```
v = pt.xr  # v = 5
```

будем получать его значение 5, а не значение 1 атрибута _x. То есть, для дескрипторов типа non-data descriptor приоритет доступа к локальным свойствам объекта и к дескриптору одинаков. И, так как объект xr находится во внешней области видимости (в классе Point3D), то берется аналогичный атрибут xr со значением 5 из локальной области объекта pt.

Однако если в дескриптор ReadInteger добавить сеттер и превратить его в дескриптор данных:

```
class ReadInteger:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, owner):
        return getattr(instance, self.name, None)

    def __set__(self, instance, value):
        setattr(instance, self.name, value)
```

А создание локального атрибута xr в объекте pt выполнить путем добавления ключа 'xr' в коллекцию __dict__:

```
pt.__dict__['xr'] = 5
```

то при обращении к атрибуту xr:

```
print(pt.xr)
print(pt.__dict__)
```

на экране увидим строчки:

```
1
{'_x': 1, '_y': 2, '_z': 3, 'xr': 5}
```

Значение 1 было взято из локального атрибута _x, что говорит о более высоком приоритете дескриптора данных в сравнении с обычными атрибутами. То есть, сначала ищется объект-дескриптора с именем xr, если он обнаруживается, то используется в указанной команде, а если нет, то будет взят атрибут с аналогичным именем xr, если он существует. Об этих приоритетах доступа к объектам-дескрипторам типов data и non-data нужно хорошо знать и помнить.

2.7. Вложенные классы

Классы в языке Python представляют собой пространства имен, в которых можно прописывать все те же самые объекты, что и в основной программе, например: строки, списки, словари, методы и многое другое, включая другие классы. То есть, в одном классе можно записать другой класс. Получаем конструкцию из вложенных классов.

Давайте в качестве примера, в класс Student вложим класс FIO, который будет хранить и обрабатывать ФИО студента, следующим образом:

```
class Student:
    class FIO:
        def __init__(self, firstname, parentname, lastname):
            self._firstname = firstname
            self._parentname = parentname
            self._lastname = lastname

    def __init__(self, fio, age, group):
        self._fio = self.FIO(*fio.split())
        self._age = age
        self._group = group
```

В результате, FIO можно воспринимать, как атрибут класса Student, который ссылается на объект-класс. Соответственно, к этому атрибуту применимы все те же самые действия, что и ко всем остальным атрибутам класса. Например, команда:

```
print(Student.FIO)
```

выведет на экран ссылку на вложенный класс:

```
<class '__main__.Student.FIO'>
```

И через эту ссылку можно создать объект вложенного класса FIO, например,

следующим образом:

```
fio = Student.FIO("Гвидо", "ван", "Россум")
```

Именно так выполняется создание объекта FIO в магическом методе __init__. Только обращение к вложенному классу осуществляется через объект self в виде конструкции `self.FIO`. Конечно, как вариант, здесь можно было бы явно прописать имя класса `Student.FIO`, но прямого указания класса внутри самого класса лучше избегать, т.к. в случае изменения его имени придется вносить правки по всему коду текущего класса. В то же время командой `self.FIO` нужно пользоваться аккуратно, исключительно только для чтения атрибута FIO класса Student. Если вместо чтения выполнить операцию присваивания, например:

```
self.FIO = 123
```

то в текущем объекте self будет сформировано новое свойство с именем FIO и значением 123, а ссылка на вложенный класс FIO будет недоступна через текущий объект класса. Все это вы уже должны хорошо знать и четко понимать.

Давайте создадим объект класса Student командой:

```
s = Student("Гвидо ван Россум", 19, "ИТсд-12")
```

Соответственно, через объект s будут доступны все его защищенные (protected) локальные свойства:

```
print(s._fio)
print(s._age)
print(s._group)
```

На экране увидим строчки:

```
<__main__.Student.FIO object at 0x0000015967594190>
19
ИТсд-12
```

Конечно, в реальных проектах обращаться напрямую извне к protected-свойствам плохая практика. Здесь это сделано исключительно в учебных, отладочных целях, чтобы быстро посмотреть их значения. Как видим, свойство _fio ссылается на объект вложенного класса FIO. Следовательно, через атрибут _fio можно обратиться к свойствам объекта FIO:

```
print(s._fio._firstname)   # Гвидо
print(s._fio._parentname)  # ван
print(s._fio._lastname)    # Россум
```

Как видите, все работает достаточно просто и очевидно. И, так как класс FIO предполагается использовать только внутри класса Student, то перед его именем можно поставить одно или два подчеркивания, то есть, сделать этот атрибут или protected или private:

```
class Student:
```

```
    class __FIO:
        def __init__(self, firstname, parentname, lastname):
            self._firstname = firstname
            self._parentname = parentname
            self._lastname = lastname

    def __init__(self, fio, age, group):
        self._fio = self.__FIO(*fio.split())
        self._age = age
        self._group = group
```

Теперь по имени __FIO можно обращаться только внутри класса Student и в его методах. За пределами класса прямое обращение к __FIO приведет к ошибке. Например, при выполнении команды:

```
fio = Student.__FIO("Гвидо", "ван", "Россум")
```

получим сообщение:

```
AttributeError: type object 'Student' has no attribute '__FIO'
```

То есть, имена вложенных классов обладают всеми теми же свойствами, что и любые другие атрибуты класса.

Так как к атрибуту __FIO можно напрямую обращаться всюду внутри класса Student, то ничто нам не мешает сформировать атрибут def_fio со ссылкой на объект вложенного класса __FIO следующим образом:

```
class Student:
    class __FIO:
        def __init__(self, firstname, parentname, lastname):
            self._firstname = firstname
            self._parentname = parentname
            self._lastname = lastname

        def fullname(self):
            return ", ".join((self._firstname, self._parentname,
self._lastname))

    def_fio = __FIO("Иван", "Иванович", "Иванов")

    ...
```

Здесь в класс __FIO добавлен метод fullname, который возвращает строку формата ИОФ. Воспользоваться атрибутом def_fio можно так:

```
s = Student("Гвидо ван Россум", 19, "ИТсд-12")
defname = Student.def_fio.fullname()
print(defname)
```

На экране увидим строчку:

```
Иван, Иванович, Иванов
```

А вот из класса __FIO нет прямого доступа к атрибутам внешнего класса Student. Давайте в этом убедимся на следующем примере:

```
class Student:
    descr = "Это класс для хранения информации по студентам."

    class __FIO:
        descr = Student.descr   # NameError

        def __init__(self, firstname, parentname, lastname):
            self._firstname = firstname
            self._parentname = parentname
            self._lastname = lastname

    ...
```

Здесь внутри вложенного класса __FIO идет обращение к атрибуту descr внешнего класса Student. При выполнении этой программы, получим ошибку:

```
NameError: name 'Student' is not defined
```

которая говорит, что имя Student не определено. Действительно, класс Student на момент определения класса __FIO еще не сформирован окончательно, поэтому он не существует на момент выполнения команды `descr = Student.descr`. Вообще, на практике лучше избегать обращений к внешним атрибутам из вложенных классов. Их следует проектировать так, чтобы они образовывали независимую программную единицу. Тогда никаких проблем с их использованием не возникнет.

Конечно, вложений одного класса в другой можно делать сколько угодно, образуя матрешку из классов. Однако в большом числе приложений вложенные классы вообще не используются. В редких случаях целесообразно делать одноуровневые вложения. А необходимость вкладывать два и более класса друг в друга встречается крайне редко. Главное требование – это писать понятный и красивый программный код. И это часто можно сделать без вложений, либо, в редких случаях, с одним вложенным классом.

ГЛАВА III. МАГИЧЕСКИЕ МЕТОДЫ КЛАССОВ

Мы с вами уже видели, что в классах можно использовать специальные методы, которые записываются с двумя подчеркивания до и после имен. Например, методы:

- __init__() – инициализатор класса;
- __new__() – вызывается перед созданием объектов класса;
- __del__() – финализатор класса;
- __set__(), __get__(), __deleter__() – методы для дескрипторов.

Такие методы называются магическими или dunder-методы (от англ. **D**ouble **Under**score). Их особенность в том, что они автоматически вызываются при выполнении определенных операций с классом или его объектами. Например, инициализатор __init__ срабатывает сразу после создания объекта класса, а метод __del__ – перед непосредственным удалением объекта.

Обратите внимание, что все магические методы являются публичными, несмотря на то, что имеют два подчеркивания вначале своих имен. Наличие еще двух подчеркиваний в конце переводит их из разряда приватных в разряд публичных атрибутов класса.

В рамках этой главы мы с вами рассмотрим основные, наиболее широко используемые магические методы, с помощью которых вы сможете создавать самую разнообразную логику работы объектов классов.

3.1. Методы __setattr__, __getattribute__, __getattr__ и __delattr__

Мы уже знаем, что класс можно воспринимать как пространство имен, в котором записаны свойства (переменные) и методы. Например, следующий класс Point содержит две переменные MAX_COORD и MIN_COORD и два метода __init__ и set_coord (четыре атрибута класса):

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0

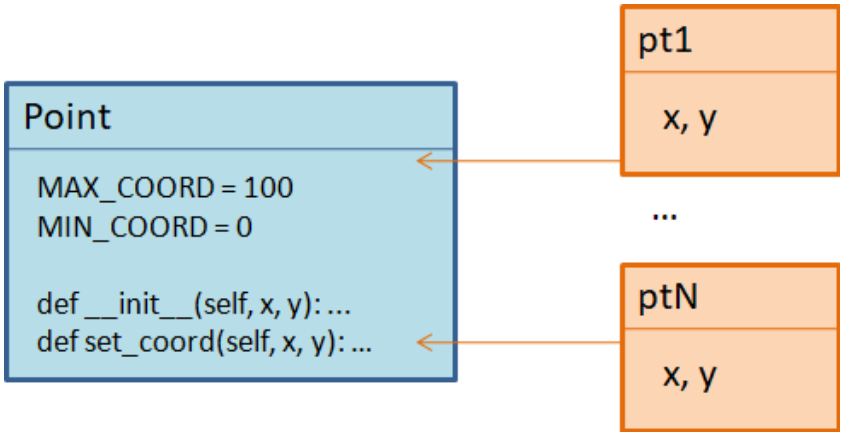
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def set_coord(self, x, y):
        self.x = x
        self.y = y
```

Все атрибуты класса при создании его экземпляров:

```
pt1 = Point(1, 2)
pt2 = Point(10, 20)
```

остаются в пространстве имен класса и не копируются в экземпляры. При этом из экземпляров pt1, pt2 можно совершенно спокойно к ним обращаться, так как пространство имен объектов содержит ссылку на внешнее пространство имен класса.



Если какой-либо атрибут не существует в экземпляре, то поиск переходит во внешнее пространство, то есть, в класс и поиск продолжается там. Поэтому через объекты можно обращаться к любому атрибуту класса, например, MAX_COORD:

```
print(pt1.MAX_COORD)
```

При обращении к атрибутам класса внутри методов этого класса необходимо перед их именами прописывать или сам класс:

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0

    ...

    def set_coord(self, x, y):
        if Point.MIN_COORD <= x <= Point.MAX_COORD:
            self.x = x
            self.y = y
```

или параметр self – ссылку на текущий объект:

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0

    ...

    def set_coord(self, x, y):
        if self.MIN_COORD <= x <= self.MAX_COORD:
            self.x = x
            self.y = y
```

Конечно, ссылку self для доступа к атрибутам класса можно указывать только в том случае, если данные читаются из атрибута. Если попытаться через такую конструкцию что-то записать, например, так:

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0

    def __init__(self, x, y):
        self.MIN_COORD = -5
        self.MAX_COORD = 5
        self.x = x
        self.y = y

    ...
```

то это приведет к формированию атрибутов MAX_COORD и MIN_COORD внутри объекта self, а аналогичные атрибуты класса останутся без изменений. Обо всем этом мы с вами уже говорили, и вы должны хорошо знать эти важные нюансы. Здесь же рассмотрим дополнительные возможности работы с атрибутами, которые дают нам следующие четыре магических метода:

- __setattr__(self, key, value) – автоматически вызывается при записи в атрибут с именем key значения value;
- __getattribute__(self, item) – автоматически вызывается при получении значения атрибута с именем item;
- __getattr__(self, item) – автоматически вызывается, когда другой магический метод __getattribute__ не смог найти указанный атрибут item;
- __delattr__(self, item) – автоматически вызывается при удалении атрибута item.

3.1.1. Магический метод __getattribute__

Давайте снова объявим уже привычный нам класс Point для представления точек на плоскости следующим образом:

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0

    def __init__(self, x, y):
        self._x = x
        self._y = y
```

Через этот класс можно обращаться к разным магическим методам, в том числе, и к __getattribute__. Например, следующая команда:

```
print(Point.__getattribute__)
```

выведет значение ссылки на экран:

```
<slot wrapper '__getattribute__' of 'object' objects>
```

Видим, что этот метод был взят из базового класса object, от которого неявно наследуются все классы в языке Python 3. О наследовании речь пойдет в следующей главе. На данный момент достаточно понимать, что все публичные атрибуты базового класса object напрямую доступны во всех его дочерних классах, в частности, в Point. Мало того, в методе __getattribute__ класса object прописан стандартный алгоритм получения значений атрибутов. Например, при выполнении команд:

```
pt = Point(1, 2)
v = pt._x
```

во второй строчке будет неявно вызван метод __getattribute__ для получения значения локального свойства _x объекта pt. Поэтому если переопределить этот магический метод в классе Point следующим образом:

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0

    def __init__(self, x, y):
        self._x = x
        self._y = y

    def __getattribute__(self, item):
        return None
```

то для всех атрибутов объекта pt будет возвращаться значение None.

Обратите внимание, что данный магический метод вызывается только при обращении к атрибутам через экземпляр класса pt. Например, следующие две команды дадут разный результат:

```
print(pt.MAX_COORD)     # None (метод __getattribute__ был вызван)
print(Point.MAX_COORD)  # 100 (метод __getattribute__ не вызывался)
```

В первом случае метод __getattribute__ сработает, а во втором нет. Это связано с тем, что команда `Point.MAX_COORD`, по сути, выполняет прямое обращение к данным с указанием области видимости класса Point.

Конечно, в таком виде метод __getattribute__ не несет полезной нагрузки. Было бы хорошо, чтобы он отрабатывал некоторую свою логику и далее повторял стандартный алгоритм получения значения атрибута. Сделать это можно следующим образом:

```
class Point:
    ...

    def __getattribute__(self, item):
        print("__getattribute__")
```

```
        return object.__getattribute__(self, item)
```

Здесь сначала вызывается функция print, а затем вызывается метод __getattribute__ напрямую из базового класса object, в котором реализован стандартный алгоритм обработки атрибута. Этот метод возвращает значение атрибута item, которое, в свою очередь, возвращается методом __getattribute__ класса Point. В результате происходит выполнение встроенного алгоритма получения значения атрибута, прописанного в базовом классе object. Теперь, при выполнении строчек:

```
pt = Point(1, 2)
print(pt.MAX_COORD)
print(Point.MAX_COORD)
```

в обоих случаях будет корректно получено значение атрибута MAX_COORD:

```
__getattribute__
100
100
```

Давайте, в качестве учебного примера с помощью магического метода __getattribute__ выполним ограничение доступа к локальным атрибутам _x, _y объектов класса Point. Сделаем это следующим образом:

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0

    def __init__(self, x, y):
        self._x = x
        self._y = y

    def __getattribute__(self, item):
        if item == '_x' or item == '_y':
            return None

        return object.__getattribute__(self, item)
```

Если параметр item в методе __getattribute__ содержит имена '_x' или '_y', то возвращается значение None, а для всех остальных атрибутов будет отрабатывать стандартный алгоритм получения значения.

Теперь, при выполнении команды:

```
print(pt._x, pt._y)
```

на экране увидим значения None:

```
None None
```

Если же обратиться к любому другому атрибуту, например:

```
print(pt.MAX_COORD)
```

то увидим значение этого атрибута, равное 100.

Вот так можно управлять извлечением значений из разных атрибутов через переопределение магического метода __getattribute__.

3.1.2. Магический метод __getattr__

Мы с вами знаем, что при обращении к несуществующему атрибуту объекта возникает ошибка AttributeError. Например, если в объекте pt класса Point прописать атрибут a:

```
value = pt.a   # AttributeError
```

то, как раз, появится эта ошибка. При этом видим, что был вызван магический метод __getattribute__:

```
Traceback (most recent call last):
  File "D:\Python\Projects\stl\main.py", line 17, in <module>
    value = pt.a
            ^^^^
  File "D:\Python\Projects\stl\main.py", line 13, in __getattribute__
    return object.__getattribute__(self, item)
           ~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^
AttributeError: 'Point' object has no attribute 'a'
```

В действительности, метод __getattribute__ в такой ситуации вызывает еще один магический метод с именем __getattr__. Давайте в этом убедимся, добавив его в класс Point следующим образом:

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0

    def __init__(self, x, y):
        self._x = x
        self._y = y

    def __getattribute__(self, item):
        print("__getattribute__: " + item)
        return object.__getattribute__(self, item)

    def __getattr__(self, item):
        print("__getattr__: " + item)
        return -1
```

Обратите внимание, что в метод __getattribute__ была добавлена функция print, чтобы мы видели ее вызов, и удалена проверка. Теперь, после запуска программы, никаких ошибок не возникнет, а на экране появятся следующие строчки:

```
__getattribute__: a
__getattr__: a
```

Таким образом, добавление метода __getattr__ в класс Point позволяет

обрабатывать ситуации с недоступными атрибутами. И то, что вернет этот метод, то значение будет возвращаться при обращении к таким атрибутам.

Если выполнить строчки:

```
pt = Point(1, 2)
value = pt.a
print(value)   # -1
```

то увидим на экране значение -1, которое возвратил магический метод __getattr__:

```
__getattribute__: a
__getattr__: a
-1
```

Если же указать любой из существующих атрибутов, например:

```
pt = Point(1, 2)
value = pt._x
print(value)   # 1
```

то метод __getattr__ не вызывается и на экране увидим:

```
__getattribute__: _x
1
```

3.1.3. Магический метод __setattr__

Следующий магический метод __setattr__ автоматически вызывается в момент присваивания атрибуту нового значения. Добавим его в класс Point следующим образом:

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0

    def __init__(self, x, y):
        self._x = x
        self._y = y

    ...

    def __setattr__(self, key, value):
        print(f"__setattr__: {key} = {value}")
        object.__setattr__(self, key, value)
```

Тогда, при выполнении команд:

```
pt = Point(1, 2)
pt.value = 5
```

На экране увидим:

```
__setattr__: _x = 1
__setattr__: _y = 2
```

```
__setattr__: value = 5
```

Обратите внимание, что магический метод __setattr__ вызывается всякий раз, когда локальному атрибуту объекта присваивается какое-либо значение. Первые два вызова связаны с присвоением значений в инициализаторе класса Point, а третий при выполнении команды `pt.value = 5`. Также в переопределенном методе __setattr__ необходимо вызвать аналогичный метод из базового класса object, который реализует основную логику формирования локальных атрибутов с заданным значением. Если команду `object.__setattr__(self, key, value)` поставить в комментарии, то формирования локальных атрибутов в объекте pt происходить не будет:

```
pt = Point(1, 2)
print(pt.__dict__)
```

На экране увидим строчки:

```
__setattr__: _x = 1
__setattr__: _y = 2
__getattribute__: __dict__
{}
```

Поэтому вызов функции __setattr__ из базового класса object необходим для корректного создания и изменения атрибутов в объектах класса.

Обратите также внимание, что магический метод __setattr__ будет вызываться только при обращении к атрибуту через объект класса. Например, при выполнении следующих команд:

```
pt = Point(1, 2)
Point.MIN_COORD = -5   # метод __setattr__ не вызывается
pt.MIN_COORD = -5      # метод __setattr__ вызывается
```

в первом случае обращения к атрибуту MIN_COORD через класс Point метод не срабатывает, а при использовании объекта pt, срабатывает. Правда, последняя команда лишь создаст новый локальный атрибут MIN_COORD в самом объекте pt и никак не изменит аналогичный атрибут класса Point. Об этом всегда следует помнить.

3.1.4. Магический метод __delattr__

Наконец, последний магический метод __delattr__ вызывается в момент удаления какого-либо атрибута из экземпляра класса. Пропишем его в классе Point следующим образом:

```
class Point:
    ...

    def __delattr__(self, item):
        print("__delattr__: "+item)
```

И выполним удаление атрибута _x из объекта pt класса Point:

```
pt = Point(1, 2)
del pt._x
```

После выполнения программы увидим строчки:

```
__setattr__: _x = 1
__setattr__: _y = 2
__delattr__: _x
```

То есть, метод __delattr__ действительно был вызван при удалении атрибута _x. Правда, если посмотреть коллекцию из всех атрибутов объекта pt:

```
print(pt.__dict__)
```

то увидим, что этот атрибут в ней по-прежнему присутствует:

```
{'_x': 1, '_y': 2}
```

Конечно, для полноценного удаления в переопределенном методе __delattr__ нужно вызывать аналогичный метод из базового класса object:

```
class Point:
    ...

    def __delattr__(self, item):
        print("__delattr__: "+item)
        object.__delattr__(self, item)
```

Теперь, после выполнения программы, атрибут _x действительно исчезает из объекта pt.

Вот принцип работы всех четырех магических методов, через которые можно гибко настраивать работу с атрибутами объектов классов.

3.2. Магический метод __call__. Классы-декораторы

Следующий магический метод __call__ автоматически вызывается, когда у объекта класса прописывается операция вызова (круглые скобки, как у функции). Давайте рассмотрим действие этого метода на конкретном примере.

Пусть в программе объявлен класс Counter (счетчик) следующим образом:

```
class Counter:
    def __init__(self):
        self.__counter = 0
```

Его объекты можно создавать командой:

```
counter = Counter()
```

И, затем, с этим экземпляром можно выполнять различные действия: создавать, удалять его атрибуты, вызывать через него методы и т.п. Но нельзя вызывать

операцией `()` (круглые скобки), т.к. counter – объект класса, а не функция. Если попытаться это сделать:

```
counter()
```

то получим ошибку, говорящую, что объекты класса Counter не являются вызываемыми:

```
TypeError: 'Counter' object is not callable
```

Однако это стандартное поведение можно изменить. Если прописать в классе Counter магический метод __call__, например, следующим образом:

```
class Counter:
    def __init__(self):
        self.__counter = 0

    def __call__(self, *args, **kwargs):
        print("__call__")
```

Тогда при выполнении команды:

```
counter()
```

ошибка пропадет и на экране появится строчка:

```
__call__
```

По сути, добавление магического метода __call__ в класс Counter изменил стандартное поведение объекта и сделал его вызываемым, подобно функциям. Но, обратите внимание, добавленный метод __call__ вызывается исключительно для объектов, а не для класса Counter. То есть, команда создания его экземпляров:

```
counter = Counter()
```

по-прежнему работает стандартным образом, т.к. не использует добавленный метод __call__. Поэтому на экране видим только одну строчку "__call__".

Давайте пропишем в методе __call__ следующую логику его работы:

```
class Counter:
    def __init__(self):
        self.__counter = 0

    def __call__(self, *args, **kwargs):
        self.__counter += 1
        return self.__counter
```

Здесь при каждом вызове магического метода будет увеличиваться значение счетчика __counter и возвращаться его значение. В результате выполнения команд:

```
counter = Counter()
counter()            # __counter = 1
```

```
cnt = counter()      # __counter = 2
```

внутренняя переменная __counter увеличится на два и это значение сохраняется в переменной cnt.

Если создать еще один объект-счетчик и также вызвать его несколько раз:

```
counter = Counter()   # первый счетчик
dist = Counter()      # второй счетчик

counter()             # counter.__counter = 1
cnt = counter()       # counter.__counter = 2

length = dist()       # dist.__counter = 1
```

то эти счетчики будут работать независимо друг от друга, т.к. имеют свой независимый набор локальных атрибутов, в частности, свойство __counter.

Если внимательно посмотреть на определение метода __call__, то видим параметры *args, **kwargs, которые позволяют передавать произвольное количество позиционных и именованных аргументов. Также допустимо прописывать свои параметры у этого метода. Давайте для примера, добавим параметр step с начальным значением 1, который будет определять шаг изменения счетчика. Получим:

```
class Counter:
    def __init__(self):
        self.__counter = 0

    def __call__(self, step=1.0, *args, **kwargs):
        self.__counter += step
        return self.__counter
```

Теперь подсчет можно менять в зависимости от параметра step, например, следующим образом:

```
counter = Counter()

counter(2)        # __counter = 2
counter(-0.5)     # __counter = 1.5
cnt = counter(3)  # __counter = 4.5
```

Вот общий принцип работы магического метода __call__.

Рассмотрим еще один пример использования этого метода. Объявим класс с именем StripChars, который будет удалять указанные символы в начале и конце переданных строк:

```
class StripChars:
    def __init__(self, chars):
        self.__chars = chars

    def __call__(self, s_treat, *args, **kwargs):
```

```
        if not isinstance(s_treat, str):
            raise TypeError("Параметр s_treat должен быть строкой.")

        return s_treat.strip(self.__chars)
```

Для этого в инициализаторе формируется локальный атрибут __chars, содержащий строку из удаляемых символов. А при вызове метода __call__ выполняется удаление символов __chars из переданной строки s_treat.

Воспользоваться этим классом можно следующим образом. Вначале создаются объекты для обработки строк, например:

```
strip_chars = StripChars("?:!.; ")
strip2 = StripChars(" ,.")
```

А, затем, используются для удаления из строк указанных символов:

```
res1 = strip_chars(" Hello World! ")
res2 = strip2(" Hello World! ")

print(repr(res1))
print(repr(res2))
```

После выполнения программы на экране появятся две строчки:

```
'Hello World'
'Hello World!'
```

Как видите, эти два объекта (strip_chars и strip2) работают совершенно независимо друг от друга. В результате получили довольно элегантное решение поставленной задачи.

3.2.1. Класс-декоратор без параметров

Используя магический метод __call__, можно создавать классы-декораторы. Мы с вами уже рассматривали декораторы на уровне функций. Тот же самый функционал можно реализовать и с помощью классов. Давайте, для примера, запишем декоратор CallLimiter, который будет ограничивать максимальное количество вызова следующей декорируемой функции:

```
def get_perimetr(width, height):
    "Функция для вычисления периметра прямоугольника."
    return 2 * (width + height)
```

Процесс декорирования этой функции с помощью класса CallLimiter в явном виде можно записать следующей командой:

```
get_perimetr = CallLimiter(get_perimetr)
```

После этого полученный объект должен вызываться ограниченное количество раз, например, максимум три раза:

```
result = get_perimetr(1, 2)  # result = 6
```

```
result = get_perimetr(3, 4)  # result = 14
result = get_perimetr(5, 6)  # result = 22
result = get_perimetr(7, 8)  # result = 22 (четвертый раз не вызывается)
```

Эти строчки показывают, как следует построить класс-декоратор CallLimiter. В его инициализатор передается ссылка на декорируемую функцию (в нашем примере get_perimetr), а сам объект класса должен быть вызываемым, то есть, содержать метод __call__, в котором и реализуется логика выполнения декорируемой функции. Получаем следующее определение класса-декоратора CallLimiter:

```
class CallLimiter:
    def __init__(self, func):
        self.__func = func              # ссылка на декорируемую функцию
        self.__limit_call = 3           # лимит вызовов
        self.__last_value = None        # последнее значение функции

    def __call__(self, *args, **kwargs):
        if self.__limit_call:           # если лимиты вызовов не исчерпаны
            self.__limit_call -= 1      # счетчик лимитов уменьшается
            self.__last_value = self.__func(*args, **kwargs)
            return self.__last_value    # возвращается значение функции

        return self.__last_value        # если лимит достигнут
```

Как видите, все достаточно просто. В инициализаторе сохраняется ссылка на функцию в приватном локальном атрибуте __func и создается еще несколько вспомогательных атрибутов. Последующий вызов декорированной функции приводит к вызову метода __call__, в котором реализуется заданная логика – ограничение на максимальное количество вызовов исходной функции __func.

Полученный класс-декоратор можно применять к функции get_perimetr и через оператор @ следующим образом:

```
@CallLimiter
def get_perimetr(width, height):
    "Функция для вычисления периметра прямоугольника."
    return 2 * (width + height)
```

Результат будет аналогичным.

Конечно, как мы с вами уже знаем, после декорирования функция get_perimetr теряет значения двух своих важных атрибутов:

```
get_perimetr.__name__   # имя функции
get_perimetr.__doc__    # описание функции
```

Для их сохранения удобно использовать функцию update_wrapper из модуля functools следующим образом:

```
from functools import update_wrapper
```

```
class CallLimiter:
    def __init__(self, func):
        self.__func = func              # ссылка на декорируемую функцию
        self.__limit_call = 3           # лимит вызовов
        self.__last_value = None        # последнее значение функции
        update_wrapper(self, func)      # сохранение __name__ и __doc__

    def __call__(self, *args, **kwargs):
        if self.__limit_call:           # если лимиты вызовов не исчерпаны
            self.__limit_call -= 1      # счетчик лимитов уменьшается
            self.__last_value = self.__func(*args, **kwargs)
            return self.__last_value    # возвращается значение функции

        return self.__last_value        # если лимит достигнут
```

Обратите внимание, что атрибуты __name__ и __doc__ создаются для объекта self, т.е. для объекта класса CallLimiter, т.к. после декорирования функция get_perimetr будет являться экземпляром класса CallLimiter, соответственно, атрибуты __name__ и __doc__ будут браться из этого объекта.

Теперь, при выполнении команд:

```
print(get_perimetr.__name__)
print(get_perimetr.__doc__)
```

на экране увидим следующие строчки:

```
get_perimetr
Функция для вычисления периметра прямоугольника.
```

3.2.2. Класс-декоратор с параметрами

Полученный класс-декоратор CallLimiter не использует дополнительных параметров в своей работе. Ему передается только ссылка на декорируемую функцию. В то же время было бы неплохо дополнительно задавать максимальный лимит вызова функций, например:

```
@CallLimiter(limit=2)
def get_perimetr(width, height):
    "Функция для вычисления периметра прямоугольника."
    return 2 * (width + height)
```

Это эквивалентно такому явному вызову декоратора:

```
get_perimetr = CallLimiter(limit=2)(get_perimetr)
```

с последующим выполнением декорированной функции get_perimetr:

```
result = get_perimetr(1, 2)  # result = 6
result = get_perimetr(3, 4)  # result = 14
result = get_perimetr(5, 6)  # result = 14 (третий раз не вызывается)
```

Из этих приведенных команд видно, что в инициализатор класса CallLimiter передаются значения параметров (в нашем примере limit=2), а магический метод

__call__ должен использоваться для непосредственного декорирования функции. Получаем следующую реализацию класса-декоратора с параметрами:

```
from functools import wraps


class CallLimiter:
    def __init__(self, limit=3):
        self.__func = None         # ссылка на декорируемую функцию
        self.__limit_call = limit  # лимит выполнения функции
        self.__last_value = None   # последнее значение функции

    def __call__(self, func):      # функция-декоратор
        self.__func = func    # сохранение ссылки на декорируемую функцию

        @wraps(func)          # сохранение атрибутов __name__ и __doc__
        def wrapper(*args, **kwargs):
            if self.__limit_call:
                self.__limit_call -= 1
                self.__last_value = self.__func(*args, **kwargs)
                return self.__last_value

            return self.__last_value

        return wrapper
```

Обратите внимание, магический метод __call__ реализует логику обычной функции-декоратора без параметров. Как работает такая конструкция, мы с вами уже подробно разбирали при рассмотрении функций-декораторов. Теперь, этим классом-декоратором можно воспользоваться, например, следующим образом:

```
@CallLimiter(limit=2)
def get_perimetr(width, height):
    "Функция для вычисления периметра прямоугольника."
    return 2 * (width + height)
```

В результате, функция get_perimetr будет вызвана максимум два раза:

```
result = get_perimetr(1, 2)  # result = 6
result = get_perimetr(3, 4)  # result = 14
result = get_perimetr(5, 6)  # result = 14 (третий раз не вызывается)
```

Также этот декоратор можно вызывать без передачи каких-либо аргументов:

```
@CallLimiter()
def get_perimetr(width, height):
    "Функция для вычисления периметра прямоугольника."
    return 2 * (width + height)
```

Тогда значение параметра limit будет равно значению по умолчанию 3. А вот прописывать декоратор без круглых скобок недопустимо:

```
@CallLimiter
```

```
def get_perimetr(width, height):
    "Функция для вычисления периметра прямоугольника."
    return 2 * (width + height)
```

Он отработает неверно, так как это будет эквивалентно следующей команде:

```
get_perimetr = CallLimiter(get_perimetr)
```

То есть, здесь в инициализатор передается ссылка на декорируемую функцию, а не какие-либо параметры самого декоратора.

Исправить эту ситуацию достаточно просто путем добавления еще одного магического метода __new__ в класс-декоратор CallLimiter со следующей реализацией:

```
class CallLimiter:
    def __new__(cls, func_or_limit=None, *args, **kwargs):
        # если функция, то декоратор вызыван без скобок ()
        if callable(func_or_limit):
            instance = super().__new__(cls)  # создаем объект класса
            instance.__init__()              # вызываем инициализатор
            return instance(func_or_limit)   # вызываем __call__

        return super().__new__(cls)         # это для вызова с параметрами

    def __init__(self, limit=3):
        self.__func = None
        self.__limit_call = limit
        self.__last_value = None

    def __call__(self, func):
        self.__func = func

        @wraps(func)
        def wrapper(*args, **kwargs):
            if self.__limit_call:
                self.__limit_call -= 1
                self.__last_value = self.__func(*args, **kwargs)
                return self.__last_value

            return self.__last_value

        return wrapper
```

В методе __new__ вначале проверяется, если первый переданный параметр является вызываемым объектом, то делается вывод о декорировании функции без указания круглых скобок (без параметров). В этом случае мы самостоятельно создаем объект класса и вручную вызываем метод __init__ с параметрами по умолчанию. Затем, декорируем функцию func_or_limit путем вызова магического метода __call__ командой `instance(func_or_limit)`. Теперь любая запись декоратора будет корректно применяться к функциям.

3.3. Магические методы __str__() и __repr__()

Каждый объект языка Python можно вывести на экран, используя функцию print. Например, если объявить какой-либо простой класс и затем сформировать объект этого класса:

```
class Car:
    def __init__(self, model, price):
        self._model = model
        self._price = price


car = Car('BMW X7', 4_067_493)
```

то при выводе экземпляра car на экран командой:

```
print(car)
```

увидим следующую строчку:

```
<__main__.Car object at 0x0000015A252C6F90>
```

И здесь возникает вопрос. Почему функция print вывела именно такую информацию? Где прописан этот формат вывода? В действительности, когда какой-либо объект выводится на экран, то автоматически срабатывает или магический метод __str__, или магический метод __repr__, которые формируют и возвращают информационную строку о состоянии текущего объекта. Так как в классе Car ни один из этих методов не объявлен, то они берутся из базового класса object для формирования стандартной информационной строки.

Почему речь идет сразу о двух магических методах для получения строки состояния объекта? В действительности, они похожи и взаимосвязаны между собой, но имеют свои назначения:

- __str__() – служит для формирования строки с общедоступной информацией об объекте и вызывается, например, в функциях str(), print(), в F-строках (по умолчанию);
- __repr__() – предназначен для формирования строки со служебной информацией об объекте и часто применяется при отладке программного кода.

Соответственно, при выполнении команды:

```
print(car)
```

вызывается метод __str__ базового класса object с формированием строки:

```
<__main__.Car object at 0x0000015A252C6F90>
```

Чтобы изменить вывод этой информации можно переопределить магический метод __str__ в классе Car. Но мы сделаем немного по-другому и переопределим

второй магический метод __repr__ следующим образом:

```
class Car:
    def __init__(self, model, price):
        self._model = model
        self._price = price

    def __repr__(self):
        return f"{self.__class__}: {self._model=}, {self._price=}"
```

Обратите внимание, методы __repr__ и __str__ должны возвращать строку. Любой другой тип данных приведет к ошибке.

После выполнения команды:

```
print(car)
```

на экране увидим строчку:

```
<class '__main__.Car'>: self._model='BMW X7', self._price=4067493
```

Но почему так произошло? В нашем случае должен был быть вызван метод __str__, а вызвался __repr__. В действительности, базовый класс object при вызове метода __str__, вызывает, в свою очередь, метод __repr__ и возвращает результат его работы. Поэтому получается, что магический метод __repr__ является, как бы, более общим для формирования информации об объекте.

Конечно, если в классе Car добавить еще и метод __str__, например, так:

```
class Car:
    def __init__(self, model, price):
        self._model = model
        self._price = price

    def __str__(self):
        return f"{self._model, self._price}"

    def __repr__(self):
        return f"{self.__class__}: {self._model=}, {self._price=}"
```

то он будет использован в функции print и на экране отобразится кортеж с локальными атрибутами объекта car:

```
('BMW X7', 4067493)
```

Если же в функции print прописать вызов функции repr:

```
print(repr(car))
```

то сработает магический метод __repr__ класса Car и на экране отобразится строчка:

```
<class '__main__.Car'>: self._model='BMW X7', self._price=4067493
```

Вот принцип работы и взаимодействия этих двух магических методов __str__ и __repr__, которые доступны для всех объектов языка Python. Здесь важно помнить, что в базовом классе object на нижнем уровне вызова располагается метод __repr__, а метод __str__ этого же класса лишь использует результат его работы. Поэтому, переопределяя метод __repr__, например, в классе Car, автоматически изменится результат работы и метода __str__, что мы с вами уже видели. Но не наоборот. Переопределение метода __str__ никак не сказывается на работе метода __repr__ базового класса object.

3.4. Магические методы __len__() и __abs__()

Следующие два магических метода __len__ и __abs__ имеют следующие назначения, применительно к объектам класса:

- __len__() – вызывается при использовании функции len();
- __abs__() – вызывается при использовании функции abs().

Давайте, для примера, объявим класс Point со следующей реализацией:

```
class Point:
    def __init__(self, *args):
        self.__coords = args
```

И нам бы хотелось определять количество координат в объекте этого класса с помощью функции len следующим образом:

```
p = Point(1, 2)
dims = len(p)
```

Если сейчас запустить эту программу, то получим ошибку:

```
TypeError: object of type 'Point' has no len()
```

которая говорит, что функция len не применима к экземплярам класса Point. Однако это поведение можно изменить, переопределив магический метод __len__, например, так:

```
class Point:
    def __init__(self, *args):
        self.__coords = args

    def __len__(self):
        return len(self.__coords)
```

Теперь никаких ошибок не возникает:

```
p = Point(10, 20)
dims = len(p)   # dims = 2
```

и переменная dims равна двум, т.к. объект p содержит ровно две координаты. То есть, в момент применения функции len к объекту p был автоматически вызван магический метод __len__, который вычисляет и возвращает количество

координат. Это же значение, затем, возвращается функцией len.

Следующий похожий магический метод __abs__ работает аналогичным образом, только срабатывает в момент вызова функции abs для экземпляра класса. Например, следующая команда:

```
res = abs(p)
```

приведет к ошибке:

```
TypeError: bad operand type for abs(): 'Point'
```

из-за отсутствия магического метода __abs__ в классе Point. Добавим его со следующей реализацией:

```
class Point:
    def __init__(self, *args):
        self.__coords = args

    def __len__(self):
        return len(self.__coords)

    def __abs__(self):
        return list(map(abs, self.__coords))
```

В приведенной реализации метод __abs__ вычисляет абсолютные координаты кортежа __coords и помещает их в список, который, затем, возвращается этим методом. Если теперь воспользоваться полученным классом:

```
p = Point(-5, 12)
dims = len(p)     # dims = 2
res = abs(p)      # res = [5, 12]
print(dims, res)
```

то на экране увидим строчку:

```
2 [5, 12]
```

Вот принцип работы магических методов __len__ и __abs__. Они используются, когда в программе предполагается применять функции len и abs к объектам того или иного класса.

3.5. Магические методы __add__(), __radd__(), __iadd__()

Следующие три магических метода, которые мы рассмотрим, связаны с арифметической операцией сложения:

- __add__() – вызывается при выполнении бинарной операции «+», когда слева записан объект класса;
- __radd__() – вызывается при выполнении бинарной операции «+», когда справа записан объект класса и предыдущий метод __add__ не сработал;
- __iadd__() – вызывается для выполнения команды «+=», т.е. при изменении

состояния текущего объекта.

Как всегда, понять принцип работы этих методов проще на конкретных примерах. Для этого реализуем класс Clock работы со временем. Его экземпляры будут хранить секунды текущего дня с начальной точкой отсчета 00:00 часов ночи (соответственно, 0 секунд), и конечной 24:00 часами (соответственно, 86400 секунд). Выходить за эти пределы недопустимо. Начальная заготовка класса будет следующей:

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds:int=0):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY
```

Здесь в инициализатор передаются секунды времени текущего дня и, так как они должны лежать в диапазоне от 0 до 86400, то применяется операция взятия остатка от деления «%» с атрибутом `__DAY = 86400`. Также в инициализаторе выполняется проверка на целочисленный тип переданного значения в параметр seconds.

Добавим в класс Clock метод get_time, который будет возвращать время в виде кортежа формата:

```
(часы, минуты, секунды)
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds:int=0):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def get_time(self):
        s = self.seconds % 60             # секунды
        m = (self.seconds // 60) % 60     # минуты
        h = (self.seconds // 3600) % 24   # часы
        return h, m, s
```

Воспользоваться этим классом можно следующим образом:

```
c1 = Clock(1000)
print(c1.get_time())
```

После выполнения программы, увидим кортеж со значениями:

```
(0, 16, 40)
```

3.5.1. Магический метод __add__()

Предположим, что теперь в объекте c1 необходимо увеличить количество секунд

на 100. Сделать это можно через локальный атрибут seconds, например, так:

```
c1.seconds = c1.seconds + 100
```

Но было бы лучше иметь возможность менять секунды командой:

```
c1 = c1 + 100
```

Если сейчас попытаться выполнить эту строчку, то возникнет ошибка:

```
TypeError: unsupported operand type(s) for +: 'Clock' and 'int'
```

которая означает, что объект класса Clock не поддерживает операцию «+» с целым числом. Изменить это поведение можно путем добавления в класс магического метода __add__. Запишем его в следующем виде:

```
class Clock:
    ...

    def __add__(self, other:int):
        if not isinstance(other, int):
            raise ArithmeticError("Правый операнд должен иметь тип int")

        return Clock(self.seconds + other)
```

В этом методе сначала выполняется проверка на целочисленный тип данных, а, затем, в операторе return формируется и возвращается новый объект класса с суммарным временем.

Если теперь выполнить строчки программы:

```
c1 = Clock(1000)
c1 = c1 + 100
print(c1.seconds)
```

то в консоли увидим увеличенное количество секунд:

```
1100
```

Как в деталях это работает? Когда выполняется команда:

```
c1 = c1 + 100
```

то это эквивалентно вызову:

```
c1 = c1.__add__(100)
```

То есть, здесь срабатывает магический метод __add__ и в его параметр other передается значение 100. Далее, формируется новый объект Clock с суммарным числом секунд и возвращается методом __add__. В результате объект c1 после операции суммирования – это новый объект класса Clock с измененным количеством секунд, а прежний объект автоматически удаляется (если на него нет внешних ссылок). И здесь возникает вопрос. Зачем в методе __add__ создавать новый объект? Почему бы не менять атрибут seconds внутри текущего объекта?

Чтобы это понять, давайте расширим функционал для операции сложения так, чтобы можно было складывать два разных объекта класса Clock, например:

```
c1 = Clock(1000)
c2 = Clock(2000)
c3 = c1 + c2
print(c3.seconds)  # 3000
```

Конечно, если сейчас запустить программу, то увидим ошибку ArithmeticError, так как параметр other не соответствует типу int. Поэтому изменим реализацию метода __add__ следующим образом:

```
class Clock:
    ...

    def __add__(self, other):
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Должен быть тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return Clock(self.seconds + sc)
```

В результате программа будет работать, и выполнять сложение, как с числами, так и с объектами класса Clock. Так вот, если бы в операторе return не формировался бы новый объект Clock, то команда:

```
c3 = c1 + c2
```

изменила бы состояние объекта c1, а переменная c3 вела бы на объект c1. Как понимаете, это значительно усложнило бы понимание и применение такой программной конструкции. Поэтому создание нового объекта – это необходимый шаг при выполнении подобных операций.

Также можно выполнять сложение сразу трех объектов, например, следующим образом:

```
c1 = Clock(1000)
c2 = Clock(2000)
c3 = Clock(3000)

c4 = c1 + c2 + c3
print(c4.seconds)   # 6000
```

Здесь каждая операция «+» возвращает новый объект класса Clock, который используется в последующих операциях.

3.5.2. Магический метод __radd__()

Еще одним важным нюансом работы операции сложения для объектов классов Clock, является порядок их записи. Если сначала прописать объект, а затем число:

```
c1 = c1 + 100
```

то никаких ошибок не будет. А, если записать, наоборот:

```
c1 = 100 + c1
```

то возникнет ошибка:

```
TypeError: unsupported operand type(s) for +: 'int' and 'Clock'
```

Это произошло из-за того, что здесь, фактически выполняется команда:

```
100.__add__(c1)
```

Но у числа 100 (объекта int) метод __add__ работает только с числами, а не с объектами класса Clock. Отсюда и возникает подобная ошибка.

Как выйти из этой ситуации? Очень просто. Язык Python предоставляет нам специальный набор эквивалентных магических методов с начальной буквой r в их именах, например:

__radd__()

Этот метод будет вызываться при расположении объекта класса Clock справа от операции сложения, и когда неприменим метод __add__.

Давайте добавим метод __radd__ в класс Clock следующим образом:

```
class Clock:
    ...

    def __add__(self, other):
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Должен быть тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return Clock(self.seconds + sc)

    def __radd__(self, other):
        return self + other
```

Смотрите, в __radd__ записана команда сложения текущего объекта self с параметром other, который может быть или числом или объектом класса Clock. В свою очередь здесь будет вызван метод __add__, но уже с правильным порядком типов данных, поэтому сложение пройдет без ошибок. Повторяя команду:

```
c1 = 100 + c1
print(c1.seconds)  # 1100
```

получаем результат 1100 секунд. Теперь такой тип сложения тоже поддерживается объектами класса Clock.

3.5.3. Магический метод __iadd__()

Наконец, у всех магических методов, связанных с арифметическими операторами,

есть еще одна модификация с первой буквой i вначале имени:

__iadd__()

Этот метод вызывается при выполнении команд вида:

```
c1 += 100
```

Если сейчас запустить программу, то никаких ошибок не возникнет, т.к. отработает метод __add__. Но в методе __add__ создается новый объект класса Clock, тогда как в операции «+=» этого делать не нужно. Поэтому имеет смысл добавить магический метод __iadd__ в класс Clock следующим образом:

```
class Clock:
    ...

    def __iadd__(self, other):
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Должен быть тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        self.seconds += sc
        self.seconds %= self.__DAY

        return self
```

Реализация этого метода аналогична методу __add__, только без создания нового объекта. Все изменения касаются исключительно текущего объекта self.

Вот принцип работы и назначение трех магических методов: __add__, __radd__ и __iadd__.

3.6. Магические методы остальных арифметических операций

По аналогии с методом __add__ в классах можно прописывать другие магические методы, реализующие различных арифметические операции:

Операция	Метод операции	Операция	Метод операции
x + y	__add__ и __radd__	x += y	__iadd__(self, other)
x - y	__sub__ и __rsub__	x -= y	__isub__(self, other)
x * y	__mul__ и __rmul__	x *= y	__imul__(self, other)
x / y	__truediv__ и __rtruediv__	x /= y	__itruediv__(self, other)
x // y	__floordiv__ и __rfloordiv__	x //= y	__ifloordiv__(self, other)
x % y	__mod__ и __rmod__	x %= y	__imod__(self, other)

Здесь все достаточно просто. Для примера добавим в класс Clock (взятого из прошлого параграфа) еще несколько арифметических магических методов. Например:

```
class Clock:
    __DAY = 86400   # число секунд в одном дне
```

```
    def __init__(self, seconds:int=0):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def _operator(self, other, _func=None, replace=False):
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Должен быть тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds

        if replace:
            self.seconds = _func(self.seconds, sc) % self.__DAY
            return self
        else:
            return Clock(_func(self.seconds, sc))

    def __add__(self, other):
        return self._operator(other, lambda x, y: x + y)

    def __radd__(self, other):
        return self + other

    def __iadd__(self, other):
        return self._operator(other, lambda x, y: x + y, replace=True)

    def __mul__(self, other):
        return self._operator(other, lambda x, y: x * y)

    def __rmul__(self, other):
        return self * other

    def __imul__(self, other):
        return self._operator(other, lambda x, y: x * y, replace=True)

    def __floordiv__(self, other):
        return self._operator(other, lambda x, y: x // y)

    def __rfloordiv__(self, other):
        return self // other

    def __ifloordiv__(self, other):
        return self._operator(other, lambda x, y: x // y, replace=True)
```

Чтобы исключить дублирование кода в магических методах, общая логика их работы вынесена в отдельный метод _operator, у которого параметр _func – это ссылка на функцию для выполнения конкретной арифметической операции; параметр replace при значении True выполняет изменение текущего объекта, а при значении False создает новый с соответствующим количеством секунд. Далее, метод _operator вызывается в каждом магическом методе с передачей ему

аргумента other и лямбда-функции для вычисления определенной арифметической операции. В магических методах с буквой «i» дополнительно прописывается аргумент `replace=True`. Получился простой и понятный код класса Clock, воспользоваться которым можно следующим образом:

```
c1 = Clock(100)
c2 = Clock(20)

c3 = c1 * c2    # c3.seconds = 2000
c3 //= 2        # c3.seconds = 1000
```

По аналогии добавляются и используются все остальные арифметические магические методы.

3.7. Магические методы операций сравнения

Следующий набор магических методов, который мы рассмотрим, связан с различными операциями сравнения. Они следующие:

- __eq__() – сравнение на равенство «==»;
- __ne__() – сравнение на неравенство «!=»;
- __lt__() – сравнение на меньше «<»;
- __le__() – сравнение на меньше или равно «<=»;
- __gt__() – сравнение на больше «>»;
- __ge__() – сравнение на больше или равно «>=».

Как и ранее, для демонстрации их работы воспользуемся классом Clock, из которого удалим все лишнее:

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds:int=0):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY
```

Изначально для объектов классов реализован только один тип сравнения на равенство и, противоположный ему, на неравенство. Например, если создать два объекта класса Clock:

```
c1 = Clock(1000)
c2 = Clock(1000)
```

то с ними можно выполнять команды:

```
res_eq = c1 == c2     # False
res_not = c1 != c2    # True
```

Однако стандартная операция сравнения объектов на равенство сравнивает идентификаторы (id) этих объектов, а не количество секунд, записанное в

локальном атрибуте seconds. Чтобы изменить этот алгоритм сравнения, в классе Clock нужно переопределить магический метод __eq__. Сделаем это следующим образом:

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds:int=0):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def __eq__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Справа должен быть тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return self.seconds == sc
```

Реализация метода __eq__ вам должна быть уже хорошо знакома. Здесь все по аналогии с арифметическими операциями, только возвращается булево значение True или False в зависимости от равенства секунд.

Теперь, после выполнения сточек:

```
c1 = Clock(1000)
c2 = Clock(1000)

res_eq = c1 == c2   # True
res_not = c1 != c2  # False
```

увидим значение True при равенстве секунд и False – при неравенстве.

Обратите внимание, что операция «!=» также стала работать по новому алгоритму, хотя явно для нее прописывается магический метод __ne__. Все дело в том, что основополагающей считается операция сравнения на равенство, т.е. магический метод __eq__. А метод __ne__ в базовом классе object просто вызывает его с инверсией результата. Поэтому для реализации обеих операций сравнения на равенство и неравенство достаточно в классе переопределить один метод __eq__.

Следующий метод __lt__, который мы добавим в класс Clock, будет выполнять сравнение «на меньше». Получим:

```
class Clock:
    ...

    def __eq__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Справа должен быть тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
```

```
        return self.seconds == sc

    def __lt__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Справа должен быть тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return self.seconds < sc
```

Теперь с объектами класса можно выполнять следующие операции:

```
c1 = Clock(1000)
c2 = Clock(1200)

res_lt1 = c1 < c2   # True
res_lt2 = c1 < 500  # False
res_gt = c1 > c2    # False
```

Но сравнение «на больше» с числом приведет к ошибке:

```
res_gt = c1 > 200   # TypeError: '>' not supported ...
```

Это связано с тем, что эти операции сравнения не взаимозаменяемые. Когда использовалась команда:

```
res_gt = c1 > c2    # False
```

то интерпретатор языка Python смог использовать операцию сравнения «<» относительно объекта c2. И, так как он так же являлся объектом класса Clock, то никаких проблем не было. В случае с целым числом ситуация иная. Чтобы ее исправить в класс Clock нужно добавить второй аналогичный метод __gt__ следующим образом:

```
class Clock:
    ...

    def __eq__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Справа должен быть тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return self.seconds == sc

    def __lt__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Справа должен быть тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return self.seconds < sc

    def __gt__(self, other):
        return not (self < other)
```

Теперь команда сравнение «на больше» с числом также будет работать:

```
res_gt = c1 > 200   # True
```

В полученном классе Clock у нас снова появилось дублирование кода. Давайте вынесем общую логику в отдельный метод, а в магических будем вызывать его. Получим:

```
class Clock:
    __DAY = 86400   # число секунд в одном дне

    def __init__(self, seconds:int=0):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def _operator_cmp(self, other, _func):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Справа должен быть тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return _func(self.seconds, sc)

    def __eq__(self, other):
        return self._operator_cmp(other, lambda x, y: x == y)

    def __lt__(self, other):
        return self._operator_cmp(other, lambda x, y: x < y)

    def __gt__(self, other):
        return self._operator_cmp(other, lambda x, y: x > y)
```

Так текст программы выглядит намного понятнее и стал значительно короче. Теперь достаточно просто добавлять или убирать разные операции сравнения. Все они работают по аналогии с рассмотренной операцией сравнения на «меньше» и «больше». Особняком стоит только операция сравнения на равенство, которая способна объединять в себе и сравнение на неравенство. Остальные операции сравнение лучше прописывать в полном объеме, т.к. они по умолчанию не реализованы в базовом классе object.

3.8. Магический метод __hash__() и его взаимосвязь с методом __eq__()

Язык Python имеет встроенную функцию hash, которая позволяет вычислять хэш различных объектов, например:

```
h1 = hash(123)          # h1 = 123
h2 = hash("Python")     # h2 = -4171804609644596644
h3 = hash((1, 2, 3))    # h3 = 529344067295497451
```

Хэш представляет собой целочисленное значение, вычисленное по определенному алгоритму, и является как бы «отпечатком» объекта, позволяя быстро сравнивать их между собой. Благодаря хэшу объект можно быстро найти

в, так называемых, хэш-таблицах, на базе которых построены такие коллекции, как dict (словарь) и set (множество).

Функция hash применима только к хэшируемым объектам. В языке Python неизменяемые типы данных относятся к таким объектам. А вот при попытке вычислить хэш для изменяемых типов, например, списка:

```
h_lst = hash([1, 2, 3])  # TypeError: unhashable type: 'list'
```

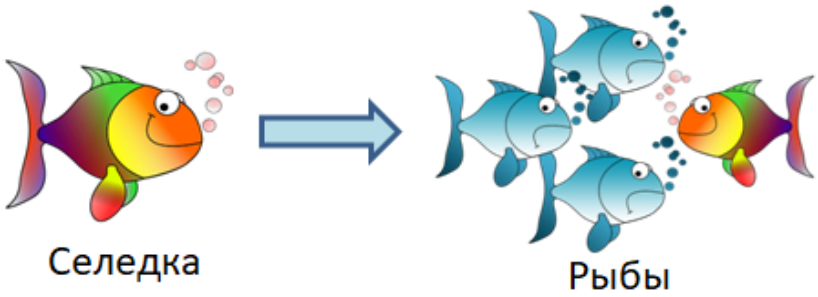
получим ошибку TypeError. Именно поэтому в ключах словарей и значениях множеств нельзя использовать изменяемые типы данных, а, точнее, нехэшируемые.

Алгоритм вычисления хэша построен так, чтобы для равных объектов на выходе получались равные хэши. Например:

```
h1 = hash(123)          # h1 = 123
h2 = hash("Python")     # h2 = -4171804609644596644
h3 = hash("Python")     # h3 = -4171804609644596644
h4 = hash(123)          # h4 = 123
```

Обратите внимание, что числа 123 – это два разных объекта с одним и тем же содержимым. И то же самое для строк "Python" – это два разных объекта в памяти устройства. Тем не менее, соответствующие значения хэшей у них равны.

Однако обратное утверждение будет неверным: равные хэши не гарантируют равенство объектов. Это, как в известном выражении: «селедка – это рыба, но не каждая рыба селедка». С хэшами все то же самое.



Однако если хеши не равны, то и объекты точно не равны. Получаем следующие свойства для хешей:

1. Если объекты a == b (равны), то равны и их хэши.
2. Если равны хеши: hash(a) == hash(b), то объекты могут быть равны, но могут быть и не равны.
3. Если хеши не равны: hash(a) != hash(b), то объекты точно не равны.

Итак, для любого неизменяемого объекта (точнее, хэшируемого) можно вычислять хэш с помощью встроенной функции hash. Затем, словари и множества используют эти хэши для доступа к данным. Соответственно, только хэшируемые объекты можно использовать в качестве ключей словаря или значений множеств.

Например:

```
d = {}
s = set()

d[5] = 5
d["python"] = "python"
d[(1, 2, 3)] = [1, 2, 3]

s.add(-5.4)
s.add("python")
```

Для чего все это понадобилось? Дело в том, что первоначально нужная запись в словаре ищется по хэшу, так как существует быстрый алгоритм его поиска. А, затем, если по найденному хэшу хранится несколько ключей (т.к. разные объекты могут иметь одинаковые хэши), то дополнительно осуществляется поиск по исходному значению ключа. Благо, объектов с равными хэшами появляется крайне мало в процессе работы программы. Такой подход значительно ускоряет поиск значений в словарях или множествах.

Давайте теперь посмотрим на порядок работы функции hash с объектами пользовательских классов. Пусть, например, имеется уже знакомый нам класс Point для представления координат точек на плоскости:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

К экземплярам этого класса:

```
p1 = Point(1, 2)
p2 = Point(1, 2)
```

можно применять функцию hash:

```
print(hash(p1), hash(p2), sep='\n')
```

На экране увидим строчки:

```
134901356281
134901519585
```

Получаем разные хэши для разных объектов p1 и p2, хотя они содержат одинаковые координаты. Дело в том, что функция hash по умолчанию вызывает магический метод __hash__, который прописан в базовом классе object, и реализует некоторую стандартную логику вычисления хэшей, не привязанной к значениям атрибутов объектов. Если необходимо, чтобы объекты класса Point с равными координатами возвращали одинаковые хэши, то нужно переопределить магический метод __hash__, например, следующим образом:

```
class Point:
```

```
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __hash__(self):
        return hash((self.x, self.y))
```

Теперь хэш объекта будет вычисляться на основе кортежа координат. В результате команды:

```
p1 = Point(1, 2)
p2 = Point(1, 2)

print(hash(p1))   # -3550055125485641917
print(hash(p2))   # -3550055125485641917
```

покажут одинаковые значения хэшей. При этом сами объекты считаются разными, т.к. операция сравнения на равенство вернет значение False:

```
print(p1 == p2)   # False
```

Это не противоречит условиям хэшей, т.к. разные объекты могут иметь одинаковые хэши. А вот обратное утверждение более строгое, и гласит:

«Если объекты a == b (равны), то должны быть равны и их хэши.»

Давайте, попробуем его нарушить. Пропишем в классе Point магический метод __eq__ и удалим метод __hash__:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

Теперь, при попытке вычислить хэш:

```
h1 = hash(p1)   # TypeError: unhashable type: 'Point'
```

возникает ошибка TypeError. Это говорит о том, что при переопределении операции сравнения на равенство, первоначальный алгоритм вычисления хэша перестает работать. Давайте вернем магический метод __hash__, получим:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __hash__(self):
```

```
        return hash((self.x, self.y))
```

Теперь условие равенства объектов и их хэшей соблюдается.

Конечно, формально, его можно нарушить, например, добавить в кортеж при вычислении хэша случайное значение:

```
from random import randint


class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __hash__(self):
        return hash((self.x, self.y, randint(0, 100)))
```

В этом случае для равных объектов будет возвращаться разный хэш. Мало того, функция hash для одного и того же объекта каждый раз будет возвращать разные значения, что нарушает общую логику работы с хэшами. Конечно, в реальной практике программирования, так делать нельзя. Это лишь учебный пример, показывающий высокое доверие к программисту со стороны языка Python. При разработке классов нам необходимо самостоятельно соблюдать для их объектов следующие три свойства хешей:

1. Если объекты a == b (равны), то равны и их хэши.
2. Если равны хеши: hash(a) == hash(b), то объекты могут быть равны, но могут быть и не равны.
3. Если хеши не равны: hash(a) != hash(b), то объекты точно не равны.

Тогда программы будут работать предсказуемо и максимально быстро. Поэтому из кортежа класса Point следует убрать генерацию случайного значения.

Объекты полученного класса Point можно использовать в качестве ключей словаря, так как они являются хэшируемыми. Например:

```
d = {}

d[p1] = 1
d[p2] = 2
```

И, учитывая, что в нашем примере объекты p1 и p2 равны между собой и имеют равные хэши, то в словаре будут восприниматься, как один и тот же ключ:

```
print(d)  # {<__main__.Point object at 0x000002E4CF1E6F90>: 2}
```

Вот примеры того, как работает функция hash языка Python и как

взаимодействуют между собой магические методы __eq__ и __hash__.

3.9. Магический метод __bool__()

Следующий магический метод __bool__ автоматически вызывается, когда к объектам класса применяется функция bool. С этой функцией мы с вами уже знакомы. Например, со стандартными типами данных она работает следующим образом:

```
bool(123)        # True
bool(-1)         # True
bool(0)          # False
bool("python")   # True
bool("")         # False
bool([])         # False
```

Здесь возвращается True для «непустых» объектов и False – для «пустых». К «пустым», в нашем примере, относится число 0, пустая строка и пустой список.

Давайте посмотрим, что она будет возвращать для объектов пользовательских классов. Объявим в программе простой класс Point:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

создадим его объект и применим к нему функцию bool:

```
p = Point(3, 4)
print(bool(p))   # True
```

После выполнения этой программы, в консоли увидим значение True. В действительности, эта функция всегда возвращает True для любых объектов пользовательских классов. Чтобы изменить это стандартное поведение, в классе Point следует прописать один из двух магических методов:

- __len__() – вызывается функцией bool(), если не определен магический метод __bool__();
- __bool__() – вызывается в приоритетном порядке функцией bool().

Давайте, для примера, добавим метод __len__ в класс Point следующим образом:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        print("__len__")
        return self.x * self.x + self.y * self.y
```

В методе __len__ выполняется вычисление квадрата длины радиус-вектора с координатами (x; y) по формуле:

$$L^2 = x^2 + y^2$$

После запуска программы, увидим строчки:

```
__len__
True
```

То есть, действительно, был вызван метод __len__ и, так как он вернул не нулевое значение, то функция bool интерпретировала его как значение True.

Если же, при создании объекта указать нулевые координаты:

```
p = Point(0, 0)
print(bool(p))   # False
```

то получим значение False, т.к. метод __len__ вернул значение 0, которое интерпретируется, как False.

Конечно, для явной реализации алгоритма функции bool, применительно к объектам класса Point, следует использовать магический метод __bool__, например, следующим образом:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        print("__len__")
        return self.x * self.x + self.y * self.y

    def __bool__(self):
        print("__bool__")
        return self.x == self.y
```

В результате, функция bool для объекта этого класса будет возвращать True, когда его координаты равны. Например:

```
p = Point(10, 10)
print(bool(p))   # True
pt = Point(10, 20)
print(bool(pt))   # False
```

На экране появятся строчки:

```
__bool__
True
__bool__
False
```

означающие, что был вызван магический метод __bool__, а не метод __len__.

Конечно, это лишь учебная реализация магического метода __bool__, демонстрирующая принцип его работы. Единственное строгое условие – это возвращаемый тип данных, который должен быть булевым True или False. Указывать другие возвращаемые типы в методе __bool__ недопустимо.

Также, обратите внимание, что функция bool может неявно применяться к объектам классов. Например, если объект p класса Point прописать в следующем условии:

```
if p:
    print("Условие истинно.")
else:
    print("Условие ложно.")
```

то оператор if автоматически применит bool к объекту p и в зависимости от полученного значения высветится либо первая, либо вторая строчка:

```
__bool__
Условие истинно.
```

Вот принцип работы магических методов __len__ и __bool__.

3.10. Магические методы __getitem__(), __setitem__() и __delitem__()

Следующий набор магических методов позволяет реализовывать логику обращения к значениям объекта по ключу:

- __getitem__(self, item) – получение значения по ключу item;
- __setitem__(self, key, value) – запись значения value по ключу key;
- __delitem__(self, key) – удаление элемента по ключу key.

3.10.1. Магический метод __getitem__()

Рассмотрим их работу на конкретном примере. Предположим, что описывается класс Student для представления студентов:

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = list(marks)
```

Объект этого класса можно сформировать следующим образом:

```
s1 = Student('Сергей', [5, 5, 3, 2, 5])
```

В полученном объекте s1 локальное свойство marks является списком с оценками студента и к нему можно обратиться командой:

```
m_2 = s1.marks[1]   # m_2 = 5
```

В результате будет выбрано второе значение из списка marks. Но что если нам

нужно сделать то же самое непосредственно через объект s1:

```
m_2 = s1[1]
```

В этом случае получим ошибку:

```
TypeError: 'Student' object is not subscriptable
```

которая говорит, что объект s1 не поддерживает такую операцию.

Как вы, наверное, уже догадались, поправить это можно с помощью магического метода __getitem__. Если записать его в классе Student следующим образом:

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = list(marks)

    def __getitem__(self, item):
        return self.marks[item]
```

то ошибок не будет и переменная m_2 примет значение 5:

```
m_2 = s1[1]  # m_2 = 5
```

Работает магический метод __getitem__ следующим образом. При использовании операции [] (обращения к элементу по ключу) параметр item принимает значение ключа (в нашем примере item = 1). Затем, идет обращение к списку marks, из которого выбирается значение элемента с индексом 1 и возвращается методом __getitem__. Однако если здесь прописать неверное значение индекса, например:

```
print(s1[20])   # IndexError: list index out of range
```

то получим ошибку IndexError, которую сгенерировал список marks. Поэтому дополнительно в методе __getitem__ следует добавить проверку корректности индексов, например, так:

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = list(marks)

    def __getitem__(self, item):
        length = len(self.marks)
        if type(item) == int and -length <= item < length:
            return self.marks[item]

        return None
```

Теперь, для некорректных индексов будет возвращаться значение None. Например:

```
print(s1['abc'])  # None
print(s1[7])      # None
```

```
print(s1[-10])    # None
```

А для верных будем получать значения из списка marks:

```
print(s1[-5])  # 5
print(s1[4])   # 5
print(s1[-2])  # 2
```

3.10.2. Магический метод __setitem__()

Если значения из объекта класса Student можно читать по индексу, то можно ли присваивать новые, используя тот же синтаксис? Давайте попробуем, выполнив команду:

```
s1[2] = 4
```

Получаем ошибку:

```
TypeError: 'Student' object does not support item assignment
```

Для поддержки такой операции в классе Student необходимо прописать еще один магический метод с именем __setitem__. Добавим его со следующей реализацией:

```
class Student:
    ...

    def __setitem__(self, key, value):
        length = len(self.marks)
        if type(key) == int and -length <= key < length:
            self.marks[key] = value
```

В методе __setitem__ выполняется проверка на целочисленный тип ключа key и его допустимое значение. Если индекс указан корректно, то в списке marks меняется значение соответствующего элемента.

Теперь команда:

```
s1[2] = 4
```

отрабатывает без ошибок и элементу с индексом 2 присваивается значение 4. Если же указать положительный индекс, превышающий размер списка, например:

```
s1[10] = 3
```

то никакого изменения списка marks не произойдет. Давайте, в учебных целях, модифицируем метод __setitem__ так, чтобы такие команды добавляли новое значение в список, а всем промежуточным элементам присваивалось значение None:

```
class Student:
    ...

    def __setitem__(self, key, value):
        length = len(self.marks)
```

```
        if type(key) == int and -length <= key < length:
            self.marks[key] = value
            return

        if type(key) == int and key > 0:
            off = key + 1 - length
            self.marks.extend([None]*off)
            self.marks[key] = value
```

Здесь добавлено еще одно условие, которое сработает, если не сработало первое. Обратите внимание на оператор return в первом условии. Как только он выполнится, метод __setitem__ завершит свою работу, и следующие команды в нем выполняться не будут. Во втором условии проверяется, чтобы ключ key был целочисленным и больше нуля. Если это условие выполняется, значит, key превышает размер списка marks. В этом случае происходит его расширение с помощью метода extends на количество элементов, равное off и значениями None. Затем в последний элемент по индексу key заносится значение value.

Если теперь выполнить команду:

```
s1[10] = 3
print(s1.marks)
```

то на экране увидим следующий список:

```
[5, 5, 3, 2, 5, None, None, None, None, None, 3]
```

То есть, он был расширен до 11 элементов и в последний элемент записано число 3. Это пример того, как можно реализовывать различные алгоритмы работы операции присваивания по ключу.

3.10.3. Магический метод __delitem__()

Третий магический метод __delitem__ из этого набора автоматически вызывается при удалении ключа. Например, командой:

```
del s1[2]
```

Если ее сейчас выполнить, то увидим сообщение об ошибке:

```
AttributeError: __delitem__
```

Она нам явно говорит, что для реализации такой команды в классе Student должен присутствовать магический метод __delitem__. Добавим его следующим образом:

```
class Student:
    ...

    def __delitem__(self, key):
        length = len(self.marks)
        if type(key) == int and -length <= key < length:
            del self.marks[key]
```

Теперь, при выполнении команды:

```
del s1[2]
```

из списка marks будет удален элемент с индексом 3, и итоговый список будет иметь вид:

```
[5, 5, 2, 5, None, None, None, None, None, 3]
```

Вот принцип работы магических методов __getitem__, __setitem__ и __delitem__.

3.11. Магические методы __iter__() и __next__()

Магические методы __iter__ и __next__, объявленные в классе, позволяют превращать его экземпляры в объекты-итераторы. Об итераторах мы с вами ранее уже говорили, и отмечали, что они являются универсальным механизмом для перебора значений элементов произвольных коллекций. То есть, каждый итерируемый объект (строка, список, кортеж, словарь, множество) внутри себя реализует алгоритм последовательного перебора значений элементов через объект-итератор. Этот итератор можно получить с помощью функции iter, например, для следующего списка:

```
lst = [5, 3, 2]
it_lst = iter(lst)   # объект-итератор
```

Затем с помощью другой функции next можно выполнять последовательное извлечение значений через этот итератор, пока не возникнет исключение StopIteration:

```
print(next(it_lst))   # 5
print(next(it_lst))   # 3
print(next(it_lst))   # 2
print(next(it_lst))   # StopIteration
```

Также нужно помнить, что итератор пробегает по элементам коллекции только один раз. Его нельзя заново инициализировать для повторного перебора значений. Например, если после перебора двух элементов:

```
print(next(it_lst))   # 5
print(next(it_lst))   # 3
```

вызвать функцию iter для объекта-итератора it_lst:

```
it_lst = iter(it_lst)
```

то это никак не повлияет на его состояние и перебор списка продолжится:

```
print(next(it_lst))   # 2
print(next(it_lst))   # StopIteration
```

Аналогичный функционал предоставляют генераторы и функции-генераторы. Например, любой генератор:

```
gen = (x ** 2 for x in range(3))
```

также является и объектом-итератором, который перебирается ровно один раз функцией next:

```
print(next(gen))     # 0
print(next(gen))     # 1
print(next(gen))     # 4
print(next(gen))     # StopIteration
```

Для реализации более сложной логики подходит функция-генератор. Пусть, например, она возвращает вещественные числа последовательности в заданном диапазоне [start; stop] с шагом step:

```
def frange(start=0.0, stop=0.0, step=1.0):
    while start <= stop:
        item = start
        start += step
        yield item
```

Вызов этой функции возвращает объект-итератор:

```
frange_it = frange(1.0, 2.0, 0.5)
```

который, при каждом вызове функции next, будет возвращать очередное значение последовательности, пока не возникнет исключение StopIteration:

```
print(next(frange_it))   # 1.0
print(next(frange_it))   # 1.5
print(next(frange_it))   # 2.0
print(next(frange_it))   # StopIteration
```

3.11.1. Магический метод __next__()

Все это вы должны уже хорошо знать и понимать. Так вот, помимо использования итераторов стандартных коллекций, генераторов и функций-генераторов в качестве итераторов, язык Python позволяет описывать классы, объекты которых будут вести себя, как итераторы, то есть, фактически, являться итераторами. Очевидно, классы в качестве итераторов имеет смысл использовать при реализации сложной логики, которую неудобно описывать на уровне функций-генераторов. Однако в учебных целях мы выберем простую задачу и повторим алгоритм функции-генератора frange.

Для генерации последовательности вещественных чисел в диапазоне [start; stop] с шагом step объявим следующий класс:

```
class FRangeIterator:
    def __init__(self, start=0.0, stop=0.0, step=1.0):
        self._start = start
        self._stop = stop
        self._step = step

    def __next__(self):
```

```
        if self._start <= self._stop:
            item = self._start
            self._start += self._step
            return item
        else:
            raise StopIteration
```

В нем прописан инициализатор, в который передаются значения start, stop, step и сохраняются в защищенных (protected) локальных свойствах текущего объекта. Магический метод __next__ реализует алгоритм последовательного вычисления очередного значения item последовательности и возвращает его. Когда достигается правая граница диапазона, генерируется исключение StopIteration. Благодаря наличию метода __next__, к любому объекту этого класса, например:

```
it = FRangeIterator(1.0, 2.0, 0.5)
```

можно применить функцию next:

```
print(next(it))   # 1.0
print(next(it))   # 1.5
print(next(it))   # 2.0
print(next(it))   # StopIteration
```

Эта функция каждый раз вызывает магический метод __next__ класса FRangeIterator и возвращает то значение, которое вернул метод __next__. То есть, объект it ведет себя, как итератор.

3.11.2. Магический метод __iter__()

Но будет ли объект it полноценным итератором? В языке Python к каждому итератору должны быть применимы функции iter и next. Почему это важно? Например, цикл for вначале автоматически вызывает функцию iter для указанного объекта, а затем, на каждой итерации применяет функцию next для получения текущего значения. Если попробовать объект it перебрать с помощью цикла for:

```
for x in it:
    print(x)
```

то получим ошибку:

```
TypeError: 'FRangeIterator' object is not iterable
```

Та же самая ошибка возникнет и при выполнении команды:

```
it = iter(it)
```

которая означает, что сейчас к объекту it не применима функция iter.

Как вы уже догадались, чтобы исправить эту ситуацию в класс FRangeIterator необходимо добавить магический метод __iter__, который автоматически вызывается при использовании функции iter. Но, что должен делать этот метод? Для класса-итератора достаточно, чтобы он возвращал текущий объект, т.е. нам

всего лишь нужно реализовать поддержку использования функции iter с объектами класса FRangeIterator. В результате получаем следующее определение класса:

```
class FRangeIterator:
    def __init__(self, start=0.0, stop=0.0, step=1.0):
        self._start = start
        self._stop = stop
        self._step = step

    def __iter__(self):
        return self

    def __next__(self):
        if self._start <= self._stop:
            item = self._start
            self._start += self._step
            return item
        else:
            raise StopIteration
```

После этого объект it можно использовать, как полноценный итератор, например:

```
it = FRangeIterator(1.0, 2.0, 0.5)

for x in it:
    print(x)
```

На экране увидим следующие значения:

```
1.0
1.5
2.0
```

Причем перебрать его можно только один раз. Если дважды выполнить цикл for с объектом it:

```
for x in it:
    print(x)

for x in it:
    print(x)
```

то на экране отобразятся те же три строчки.

3.11.3. Пример реализации итератора для перебора атрибутов объекта класса

Однако не каждый перебираемый объект является итератором. Например, если взять какой-либо кортеж:

```
t = (7, 4, 3)
```

и применить к нему функцию next:

```
next(t)  # TypeError: 'tuple' object is not an iterator
```

то получим ошибку, говорящую, что объект t не является итератором. Тем не менее, мы его легко можем перебрать в цикле for. Все дело в том, что кортежи поддерживают функцию iter, которая возвращает итератор для перебора элементов этой коллекции:

```
it = iter(t)   # получение итератора для кортежа
```

и уже к нему можно применять функцию next:

```
print(next(it))  # 7
print(next(it))  # 4
```

В чем преимущество такого подхода? Так как объект t сам по себе не итератор, то функция iter, примененная к нему, каждый раз может возвращать новый объект-итератор, а не самого себя. В частности, это означает, что повторный вызов:

```
it2 = iter(t)
```

создаст еще один итератор, которым можно с самого начала вновь перебирать элементы кортежа:

```
print(next(it2))  # 7
print(next(it2))  # 4
```

Именно поэтому один и тот же кортеж можно много раз перебирать в циклах for, т.к. каждый раз он возвращает новый итератор.

Давайте повторим этот функционал и реализуем итератор, который будет перебирать значения атрибутов объектов определенного класса, например, такого:

```
class Thing:
    def __init__(self, name, weight, price, volume):
        self.name = name
        self.weight = weight
        self.price = price
        self.volume = volume
```

Согласно поставленной задаче, к объектам этого класса:

```
th = Thing('Книга', 1.2, 4300, 500)
```

должна быть применима функция iter, которая возвращает объект-итератор:

```
th_iter = iter(th)
```

Однако сейчас получим ошибку:

```
TypeError: 'Thing' object is not iterable
```

Очевидно, нам нужно добавить магический метод __iter__ в класс Thing. При этом объекты этого класса не должны быть итераторами. В методе __iter__ нужно каждый раз создавать отдельный объект-итератор, описываемый другим классом.

Пусть, для примера, это будет класс ThingAttrIter, который мы вложим в класс Thing. Соответственно, в магическом методе __iter__ будет создаваться объект вложенного класса и возвращаться. Получаем следующую реализацию:

```
class Thing:
    class ThingAttrIter:
        def __init__(self, obj):
            self._obj = obj
            self._attrs = tuple(obj.__dict__.keys())
            self._attr_no = 0

        def __iter__(self):
            return self

        def __next__(self):
            self._attr_no += 1
            if self._attr_no <= len(self._attrs):
                return getattr(self._obj, self._attrs[self._attr_no-1])
            else:
                raise StopIteration

    def __init__(self, name, weight, price, volume):
        self.name = name
        self.weight = weight
        self.price = price
        self.volume = volume

    def __iter__(self):
        return self.ThingAttrIter(self)
```

Класс ThingAttrIter описывает работу обычного итератора для перебора локальных атрибутов переданного ему объекта. В этом классе определены два магических метода __iter__ и __next__. Первый просто возвращает объект self класса ThingAttrIter, а второй описывает логику перебора и извлечения значения текущего атрибута из переданного объекта.

Внешний класс Thing также имеет магический метод __iter__, при каждом вызове которого создается новый объект-итератор с помощью класса ThingAttrIter с передачей ссылки на текущий объект класса Thing. В результате, команды:

```
th_iter1 = iter(th)
th_iter2 = iter(th)
```

каждый раз будут возвращать новые итераторы для перебора значений локальных атрибутов объекта th, например:

```
print(next(th_iter1))  # Книга
print(next(th_iter1))  # 1.2

print(next(th_iter2))  # Книга
print(next(th_iter2))  # 1.2
```

А в цикле for можно непосредственно указывать объект th для многократного перебора его атрибутов:

```
for x in th:
    print(x, end=" ")

print()

for x in th:
    print(x, end=" ")
```

На экране увидим строчки:

```
Книга 1.2 4300 500
Книга 1.2 4300 500
```

Таким образом, мы разделили объекты класса Thing и итератор для перебора локальных атрибутов внутри этих объектов.

ГЛАВА IV. НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

На данный момент мы с вами научились объявлять классы, создавать их объекты, ограничивать видимость атрибутов (режимы public, protected, private) и разобрали наиболее употребительные магические методы. Следующий важный шаг в изучении ООП – это наследование классов. Наследование составляет один из трех принципов, положенный в основу парадигмы ООП, наряду с инкапсуляцией (реализуется через режимы доступа к атрибутам) и полиморфизмом. Что же такое наследование? Для чего используется? И как реализуется? Все эти вопросы мы будем рассматривать в рамках этой главы. Кроме того, познакомимся с идеей и полиморфизма на уровне классов и ее воплощением на языке Python.

4.1. Наследование в объектно-ориентированном программировании

Давайте предположим, что в программе объявлены два независимых класса:

```
class Geom:
    name = 'Geom'


class Line:
    def draw(self):
        print("Рисование линии")
```

Каждый объект класса работает только со своими атрибутами. Например, если создать экземпляр класса Geom:

```
g = Geom()
```

то можно обратиться к его свойству name:

```
fig_name = g.name
```

но нельзя вызвать метод draw класса Line, так как это совершенно другой, независимый класс:

```
g.draw()   # AttributeError: 'Geom' object has no attribute 'draw'
```

И то же самое для объекта Line:

```
l = Line()         # ok
l.draw()           # ok
line_name = l.name # AttributeError: 'Line' object has no attribute 'name'
```

Можно вызвать его метод draw, но нельзя обратиться к свойству name другого класса Geom. То есть, эти классы (пространства имен) совершенно независимы между собой. Но, при необходимости, мы можем настроить связь между ними и, например, сделать так, чтобы публичные (public) атрибуты класса Geom были доступны в классе Line. Записывается такая связка, следующим образом:

```
class Line(Geom):
    def draw(self):
```

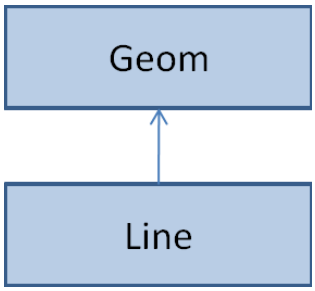
```
        print("Рисование линии")
```

В круглых скобках указывается класс Geom, который расширяется классом Line и все публичные (public) и защищенные (protected) атрибуты класса Geom становятся доступным в классе Line. В частности, теперь можно обратиться к свойству name через объект класса Line:

```
line_name = l.name   # line_name = "Geom"
```

Такая конструкция, когда один класс определяется на основе другого, называется **наследованием**. Причем класс Geom является **родительским** или **базовым**, а класс Line – **подклассом родительского** или **дочерним классом**.

Графически иерархию наследования классов принято изображать в виде прямоугольников, соединенных стрелками, причем стрелка направлена от дочернего к базовому классу:



Какую практическую роль играет механизм наследования? Рассмотрим это на следующем примере. Давайте, добавим в класс Line метод set_coords для записи координат в объект-линию:

```
class Line(Geom):
    def set_coords(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

    def draw(self):
        print("Рисование линии")
```

А, затем, объявим еще один подобный класс для прямоугольников:

```
class Rect(Geom):
    def set_coords(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

    def draw(self):
        print("Рисование прямоугольника")
```

Как видите, здесь появилось дублирование кода. И оно будет быстро нарастать с

увеличением классов для различных геометрических фигур. Чтобы этого не было, можно общее для всех дочерних классов вынести в базовый класс Geom и, в частности, записать в нем метод set_coords. В итоге получим следующие определения классов:

```
class Geom:
    name = 'Geom'

    def set_coords(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

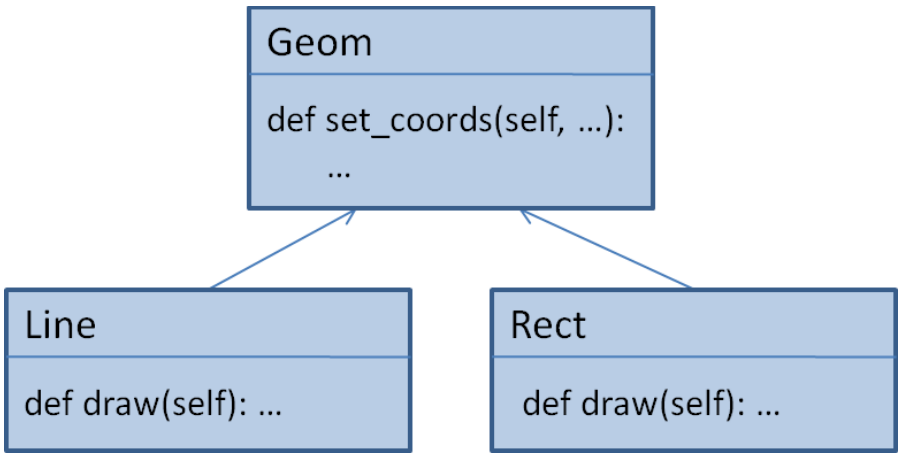

class Line(Geom):
    def draw(self):
        print("Рисование линии")


class Rect(Geom):
    def draw(self):
        print("Рисование прямоугольника")
```

В результате устранили дублирование кода, а метод set_coords теперь можно вызывать напрямую из базового класса Geom через объекты его дочерних классов, например:

```
l = Line()
r = Rect()
l.set_coords(1, 1, 2, 2)
r.set_coords(10, 10, 20, 20)
```

Иерархия наследования принимает вид:



Давайте теперь внимательнее посмотрим, как отрабатывает метод set_coords базового класса Geom. Во-первых, когда происходит его вызов, например, через объект класса Line:

```
l.set_coords(1, 1, 2, 2)
```

то метод с таким именем сначала ищется в классе Line и если не находится, то поиск продолжается в базовых классах. В данном случае – это класс Geom. В нем этот метод обнаруживается и запускается.

Во-вторых, при вызове метода set_coords из базового класса его параметр self будет ссылаться на объект, через который этот метод был вызван. То есть, на объект класса Line. И это очень важный момент! Параметр self в базовых классах может ссылаться не только на объекты этого же класса, но и на объекты производных (дочерних) от него классов, как в нашем случае – на объект класса Line. Почему это важно? Смотрите, формально сейчас в методе set_coords можно вызвать метод дочернего класса draw:

```
class Geom:
    name = 'Geom'

    def set_coords(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.draw()     # когда self объект Line или Rect
```

После запуска программы увидим, что сначала была нарисована линия, а затем, прямоугольник:

```
Рисование линии
Рисование прямоугольника
```

Это произошло по той причине, что сначала вызывался метод set_coords через объект класса Line, а потом – через объект класса Rect.

Однако такой вызов метода draw в базовом классе Geom таит в себе потенциальную ошибку. Например, если создать объект базового класса:

```
g = Geom()
```

а, затем, выполнить метод set_coords:

```
g.set_coords(0, 0, 0, 0)
```

то при выполнении программы получим ошибку, что метод draw не определен:

```
AttributeError: 'Geom' object has no attribute 'draw'
```

Действительно, теперь параметр self метода set_coords ссылается на объект базового класса Geom, в котором метод draw не существует. Поэтому при реализации методов, в классах следует обращаться только к разрешенным атрибутам либо внутри текущего класса, либо базовых классов. Но не дочерних.

Чтобы подобная ошибка в дальнейшем не возникала, следует убрать вызов метода

draw в методе set_coords класса Geom.

Похожим образом происходит обращение к атрибутам базовых классов. Сейчас в классе Geom прописано свойство name и, как мы видели, к нему можно обращаться из объектов дочерних классов:

```
l = Line()
r = Rect()

print(l.name)   # Geom
print(r.name)   # Geom
```

Правило поиска здесь такое же: если указанное свойство не находится в текущем объекте класса или в текущем классе, то поиск продолжается в его базовых классах. Поэтому если свойство name прописать непосредственно в дочернем классе, например, Line:

```
class Line(Geom):
    name = 'Line'

    def draw(self):
        print("Рисование линии")
```

то для объектов-линий оно будет взято непосредственно из класса Line, а для прямоугольников, по-прежнему, из базового класса Geom:

```
print(l.name)   # Line
print(r.name)   # Geom
```

Обратите внимание, что определение в дочернем классе какого-либо существующего атрибута базового класса называется **переопределением**. Как пример, в классе Line переопределяется атрибут name, который присутствует в базовом классе Geom.

То же самое происходит и при переопределении методов. После добавления в класс Geom метода draw:

```
class Geom:
    name = 'Geom'

    def set_coords(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

    def draw(self):
        print("Рисование примитива")
```

в соответствующих его дочерних классах этот метод будет переопределяться.

Вызов переопределенных методов выполняется очевидным образом. Когда метод

draw вызывается из дочерних объектов:

```
l.draw()    # Рисование линии
r.draw()    # Рисование прямоугольника
```

то берутся реализации из соответствующих дочерних классов. А при использовании объекта базового класса Geom, метод draw, конечно же, берется из этого класса:

```
g = Geom()
g.draw()    # Рисование примитива
```

Разумеется, если в каком-либо дочернем классе убрать метод draw, например, в прямоугольнике (Rect), то для него будет вызван метод базового класса Geom.

4.1.1. Расширение и переопределение

Когда дочерний класс наследуется от базового, добавляя новый функционал, то это называется **расширением** (extended) базового класса. Следующий пример показывает расширение класса Geom его дочерним классом Line за счет добавления нового метода draw:

```
class Geom:
    name = 'Geom'


class Line(Geom):
    def draw(self):
        print("Рисование линии")
```

Как правило, дочерние классы создаются именно для расширения функциональности базовых классов. Однако если в классе Geom тоже прописать метод draw:

```
class Geom:
    name = 'Geom'

    def draw(self):
        print("Рисование примитива")
```

то теперь класс Line лишь **переопределяет** (overriding) поведение базового класса, не добавляя ничего принципиально нового. Поэтому, когда говорят о расширении, то подразумевают добавление новых атрибутов в дочерних классах, а при переопределении (обычно методов) – изменение поведения уже существующего функционала.

4.2. Функция issubclass(). Наследование от стандартных типов

Ранее мы с вами уже не раз отмечали, что в Python 3 любой пользовательский класс по умолчанию наследуется от базового класса object. В этом легко убедиться, если, например, объявить пустой класс:

```
class Geom:
    pass
```

и вывести его свойство __base__:

```
print(Geom.__base__)
```

Увидим строчку:

```
<class 'object'>
```

Поэтому простое объявление любого класса эквивалентно следующей записи:

```
class Geom(object):
    pass
```

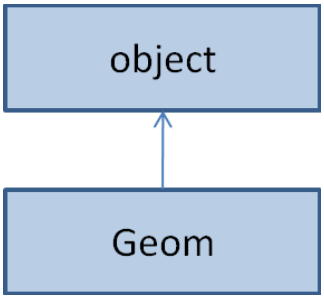
Соответственно, все публичные (public) атрибуты (включая методы) класса object доступны и в классе Geom. Но зачем это было сделано? Очевидно, чтобы обеспечить стандартный базовый функционал работы с классами. В частности, когда происходит создание экземпляра класса:

```
g = Geom()
```

автоматически вызывается магический метод __new__ класса object. А при выводе объекта g на экран:

```
print(g)
```

вызывается магический метод __str__, который также определен в базовом классе object. Таким образом, благодаря наличию базового класса object, мы сразу получаем весь начальный функционал работы с классами и его объектами. В результате, по умолчанию имеем следующую иерархию наследования:

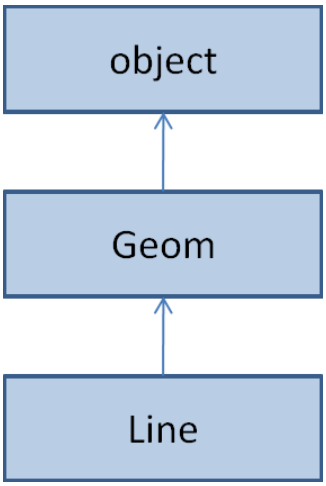


Однако если при определении класса явно прописать базовый класс, отличный от object, например, еще один класс Line наследуется от класса Geom:

```
class Geom(object):
    pass


class Line(Geom):
    pass
```

то иерархия наследования станет следующей:



Разумеется, здесь объекты класса Line также имеют полный доступ ко всем открытым атрибутам класса object:

```
l = Line()
print(l.__hash__())   # 129549141853
```

При возникновении сложных иерархий наследования, нередко возникает необходимость определения, является ли тот или иной класс подклассом другого класса. Например, определить, что класс Line – это дочерний класс для Geom. Для этого в Python существует стандартная функция issubclass, которой можно воспользоваться следующим образом:

```
res = issubclass(Line, Geom)   # res = True
```

Эта функция возвращает True, если класс Line является подклассом класса Geom, и False в противном случае. Причем функция issubclass определяет дочерний класс, расположенный в любом месте цепочки наследования. Например, тот же результат будет и при указании базового класса object:

```
res = issubclass(Line, object)   # res = True
```

так как класс Line в качестве базового имеет и его, только через класс Geom. А вот если поменять порядок записи классов Geom и Line:

```
res = issubclass(Geom, Line)     # res = False
```

то получим False, так как класс Geom не является дочерним для класса Line.

Обратите внимание, что функция issubclass работает только с классами, но не с его объектами. Например, если записать:

```
line = Line()
res = issubclass(line, Geom)
```

то получим ошибку:

```
TypeError: issubclass() arg 1 must be a class
```

говорящую, что аргумент должен быть классом, а не его экземпляром. Как мы с вами уже знаем, для определения принадлежности объекта к тому или иному

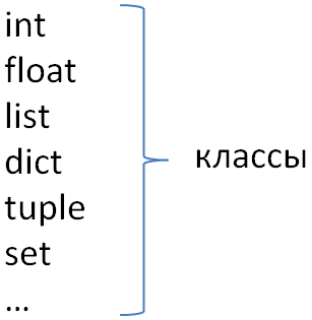
классу (это эквивалент проверки типа данных объекта), следует использовать функцию isinstance. Например:

```
line = Line()
geom = Geom()

res1 = isinstance(line, Geom)  # res1 = True
res2 = isinstance(line, Line)  # res2 = True
res3 = isinstance(geom, Line)  # res3 = False
```

4.2.1. Наследование от стандартных типов данных

Мы с вами ранее в программах использовали целые и вещественные числа (int, float), списки и кортежи (list, tuple), словари и множества (dict, set). Так вот, все эти обозначения типов в языке Python являются классами:



В этом легко убедиться, если применить к ним функцию issubclass, например:

```
res1 = issubclass(int, object)    # res1 = True
res2 = issubclass(list, object)   # res2 = True
```

И, так как, эта функция работает исключительно с классами, а не с объектами, то это явно показывает, что все стандартные типы являются классами. Но раз это так, то можно ли наследоваться от них и расширять их функционал в дочерних классах? Да, это вполне возможно. Например, объявим класс Vector, который будет наследоваться от стандартного класса list и иметь следующую реализацию:

```
class Vector(list):
    def __repr__(self):
        return " ".join(map(str, self))
```

В этом классе переопределен магический метод, который будет вызываться при выводе объекта этого класса в консоль, например, так:

```
v = Vector([1, 2, 3])
v.append(4)
print(v)
```

На экране увидим:

```
1 2 3 4
```

Теперь класс Vector обладает всем функционалом списков языка Python, только выводит их на экран несколько иначе. Конечно, это всего лишь учебный пример,

и для изменения вывода не имеет смысла объявлять отдельный класс. Можно было бы прописать отдельную функцию, если в этом возникает необходимость. Здесь лишь показывается, как гибко можно использовать стандартные типы языка Python, которые сами по себе являются классами.

4.3. Функция super() и делегирование

При разработке дочерних классов в ряде случаев возникает необходимость явного вызова методов из базового класса. Конечно, во многих ситуациях этот процесс выполняется автоматически. Например, если в базовом классе Geom прописать инициализатор, а затем создать объект дочернего от него класса Line:

```
class Geom:
    name = 'Geom'

    def __init__(self):
        print("инициализатор Geom")


class Line(Geom):
    def draw(self):
        print("Рисование линии")


line = Line()
```

то после запуска программы увидим ожидаемое поведение: автоматически будет вызван инициализатор базового класса Geom. В действительности, здесь происходит следующая последовательность вызовов:

1. Сначала вызывается магический метод __call__ из метакласса type (о метаклассах речь еще впереди).
2. Затем метод __call__ вызывает магический метод __new__ класса Line. Но, так как в классе Line (и классе Geom) не переопределен метод __new__, то он вызывается из базового класса object.
3. После создания объекта, метод __call__ вызывает магический метод __init__ класса Line. Но, так как он в этом классе не прописан, то вызов осуществляется из базового класса Geom.

В упрощенном виде реализацию метода __call__ можно записать так:

```
def __call__(cls, *args, **kwargs):           # шаг 1
    obj = cls.__new__(cls, *args, **kwargs)   # шаг 2
    if isinstance(obj, cls):
        cls.__init__(obj, *args, **kwargs)    # шаг 3
    return obj
```

Обратите внимание, что поиск всех этих методов начинается с дочернего класса Line. Если какой-либо из них не находится, то поиск продолжается в последующих родительских классах в порядке иерархии наследования. Причем параметр cls в методах __call__ и __new__ будет ссылаться на класс Line, а

параметр self в методе __init__ на созданный объект класса Line. Об этом мы с вами уже говорили, и это нужно помнить. Параметр self в методах класса всегда ссылается на объект, из которого метод был вызван.

Учитывая порядок вызова методов в иерархии классов, легко понять, что если прописать инициализатор еще и в классе Line (в классе Geom он остается):

```
class Line(Geom):
    def __init__(self):
        print("инициализатор Line")

    def draw(self):
        print("Рисование линии")
```

то при создании объекта этого класса:

```
line = Line()
```

именно он и будет вызван:

```
инициализатор Line
```

Инициализатор класса Geom в этом случае не вызывается, т.е. инициализатор класса Line как бы «скрывает» инициализаторы всех остальных базовых классов, если они в них есть.

Давайте с помощью метода __init__ класса Line будем формировать локальные свойства с координатами линии следующим образом:

```
class Line(Geom):
    def __init__(self, x1, y1, x2, y2, color=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

    def draw(self):
        print("Рисование линии")
```

Теперь объекты этого класса следует создавать с передачей четырех или пяти аргументов, например, так:

```
line = Line(0, 0, 10, 20)
```

Однако такую реализацию метода __init__ лучше прописывать в базовом классе Geom, так как при добавлении новых классов геометрических фигур получим дублирование кода. Поэтому вынесем общую реализацию инициализатора из класса Line в базовый класс Geom:

```
class Geom:
    name = 'Geom'

    def __init__(self, x1, y1, x2, y2):
```

```
        print(f"инициализатор Geom для {self.__class__}")
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
```

А в дочернем классе Line оставим только формирование локального атрибута color:

```
class Line(Geom):
    def __init__(self, x1, y1, x2, y2, color=0):
        print("инициализатор Line")
        self.color = color

    def draw(self):
        print("Рисование линии")
```

Если теперь создать экземпляр класса Line и вывести его локальные атрибуты:

```
line = Line(0, 0, 10, 20)
print(line.__dict__)
```

то в консоли увидим строчки:

```
инициализатор Line
{'color': 0}
```

Был сформирован только один атрибут color. Отсутствуют еще четыре с координатами линии. И это логично, так как был вызван только инициализатор класса Line, а метод __init__ класса Geom проигнорирован. Однако для полноценного формирования всех локальных атрибутов объекта необходимо также вызывать инициализатор базового класса Geom. Как это сделать? Конечно, в данном случае, можно было бы явно указать имя базового класса Geom и вызвать через него магический метод __init__ в инициализаторе класса Line следующим образом:

```
class Line(Geom):
    def __init__(self, x1, y1, x2, y2, color=0):
        Geom.__init__(self, x1, y1, x2, y2)
        print("инициализатор Line")
        self.color = color
    ...
```

Но явно указывать имена базовых классов не лучшая практика, так как их названия и иерархия наследования могут меняться. Поэтому в Python для обращения к «ближайшему» базовому классу используется специальная функция super. Эта функция возвращает ссылку на, так называемый, объект-посредник, через который происходит вызов метода базового класса. В нашем примере воспользоваться этой функцией можно следующим образом:

```
class Line(Geom):
    def __init__(self, x1, y1, x2, y2, color=0):
```

```
        super().__init__(x1, y1, x2, y2)  # делегированный вызов
        print("инициализатор Line")
        self.color = color
    ...
```

После запуска программы видим, что были вызваны оба инициализатора при создании объекта класса Line и сформированы требуемые локальные свойства:

```
инициализатор Geom для <class '__main__.Line'>
инициализатор Line
{'x1': 0, 'y1': 0, 'x2': 10, 'y2': 20, 'color': 0}
```

Такое прямое обращение к переопределенным методам базового класса называется **делегированием**. В данном случае, был делегирован вызов инициализатора класса Geom для создания в объекте line локальных свойств с координатами линии. Причем вызов метода __init__ базового класса лучше делать в первой же строчке инициализатора класса Line, чтобы случайно не переопределить какие-либо локальные свойства, формируемые в инициализаторе дочернего класса. Например, если прописать следующую реализацию классов Geom и Line:

```
class Geom:
    name = 'Geom'

    def __init__(self, x1, y1, x2, y2):
        print(f"инициализатор Geom для {self.__class__}")
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.color = 0


class Line(Geom):
    def __init__(self, x1, y1, x2, y2, color=0):
        print("инициализатор Line")
        self.color = color
        super().__init__(x1, y1, x2, y2) # будет обнулять color

    def draw(self):
        print("Рисование линии")
```

то, очевидно, свойство color будет неявно обнуляться при вызове __init__ базового класса в последней строчке инициализатора дочернего класса:

```
line = Line(0, 0, 10, 20, 255)
print(line.__dict__)  # {'color': 0, 'x1': 0, 'y1': 0, 'x2': 10, 'y2': 20}
```

По этой причине рекомендуется вызов инициализатора базового класса прописывать в первой же строчке:

```
class Line(Geom):
```

```
    def __init__(self, x1, y1, x2, y2, color=0):
        super().__init__(x1, y1, x2, y2)  # корректный вызов __init__
        print("инициализатор Line")
        self.color = color

    def draw(self):
        print("Рисование линии")
```

4.4. Поведение атрибутов private и protected при наследовании

Мы знаем, что в классах атрибуты могут быть записаны в трех режимах доступа: public (общедоступные), protected (защищенные) и private (закрытые, приватные). Также известно, что для обозначения protected-атрибутов перед их именами прописывается один символ подчеркивания, а для обозначения private-атрибутов – два символа подчеркивания. Однако, обратите внимание, если два подчеркивания записаны до и после имени, то это обозначение публичных dunder-атрибутов, к которым относятся магические методы и некоторые специальные свойства (например, object.__class__, self.__dict__ и т.п.). И здесь возникает вопрос, как ведут себя атрибуты разных режимов при наследовании классов? Давайте, разберемся.

Сразу отметим, что публичные (public) и защищенные (protected) атрибуты базового класса без ограничений доступны в соответствующих дочерних классах. А вот доступ к приватным (private) атрибутам по их исходным именам возможен только внутри того класса, где они определены.

4.4.1. Атрибуты private и protected объектов классов

Давайте, для примера, объявим в программе два класса: базовый Geom и дочерний Rect со следующими реализациями:

```
class Geom:
    name = 'Geom'
    _id_class = 1

    def __init__(self, x1, y1, x2, y2):
        print(f"инициализатор Geom для {self.__class__}")
        self.__x1 = x1
        self.__y1 = y1
        self.__x2 = x2
        self.__y2 = y2


class Rect(Geom):
    def __init__(self, x1, y1, x2, y2, fill='red'):
        super().__init__(x1, y1, x2, y2)
        self.__fill = fill
```

Здесь в инициализаторе базового класса Geom формируются приватные локальные свойства с координатами прямоугольника, а в инициализаторе

дочернего класса Rect создается приватное свойство __fill.

Где и как будут формироваться эти атрибуты при создании объекта класса Rect? Для этого создадим объект дочернего класса и выведем его локальные атрибуты в консоль:

```
rect = Rect(0, 0, 10, 20)
print(rect.__dict__)
```

После запуска программы на экране появятся следующие строчки:

```
инициализатор Geom для <class '__main__.Rect'>
{'_Geom__x1': 0, '_Geom__y1': 0, '_Geom__x2': 10, '_Geom__y2': 20,
'_Rect__fill': 'red'}
```

Видим, что локальные атрибуты __x1, __y1, __x2, __y2 имеют префикс _Geom, то есть, префикс того класса, в котором непосредственно создаются. Несмотря на то, что параметр self является ссылкой на объект класса Rect. Это особенность поведения (формирования) приватных атрибутов в базовых классах. У них всегда добавляется префикс соответствующего базового класса, а не класса объекта self. А вот последнее свойство __fill имеет ожидаемый префикс _Rect, так как оно было создано в классе Rect.

Что из этого следует? Во-первых, мы, конечно же, не можем обратиться к свойствам __x1, __y1, __x2, __y2 в дочернем классе Rect. Например, если в него добавить метод get_coords, который бы возвращал значения координат:

```
class Rect(Geom):
    def __init__(self, x1, y1, x2, y2, fill='red'):
        super().__init__(x1, y1, x2, y2)
        self.__fill = fill

    def get_coords(self):
        return (self.__x1, self.__y1, self.__x2, self.__y2)
```

а, затем, вызвать через ранее созданный объект класса Rect:

```
rect.get_coords()
```

Увидим ошибку:

```
AttributeError: 'Rect' object has no attribute '_Rect__x1'. Did you mean:
'_Geom__x1'?
```

Но если перенести этот метод в базовый класс Geom, то все сработает без ошибок, так как приватным свойствам будет добавлен правильный префикс _Geom.

На первый взгляд, это может показаться немного запутанным. Но если вспомнить, для чего вообще нужны приватные атрибуты, то все встает на свои места. Мы с вами ранее говорили, что private – это закрытые от внешнего доступа атрибуты текущего класса, доступные только внутри него, исключая дочерние классы. Именно поэтому приватные атрибуты жестко привязываются к текущему классу,

в котором создаются, так как по логике предполагается их использовать только внутри этого класса и больше нигде.

Если же нам нужно определить атрибуты, используемые только в текущем классе и во всех его дочерних классах, то для этого следует использовать режим protected. Поэтому в нашем примере в базовом классе Geom было бы правильнее создать переменные для координат с режимом доступа protected:

```
class Geom:
    name = 'Geom'
    _id_class = 1

    def __init__(self, x1, y1, x2, y2):
        print(f"инициализатор Geom для {self.__class__}")
        self._x1 = x1
        self._y1 = y1
        self._x2 = x2
        self._y2 = y2


class Rect(Geom):
    def __init__(self, x1, y1, x2, y2, fill='red'):
        super().__init__(x1, y1, x2, y2)
        self._fill = fill

    def get_coords(self):
        return (self._x1, self._y1, self._x2, self._y2)
```

Тогда, при вызове метода get_coords, никаких проблем не возникнет:

```
rect = Rect(0, 0, 10, 20)
coords = rect.get_coords()

print(rect.__dict__)
print(coords)
```

После запуска программы увидим следующие строчки:

```
инициализатор Geom для <class '__main__.Rect'>
{'_x1': 0, '_y1': 0, '_x2': 10, '_y2': 20, '_fill': 'red'}
(0, 0, 10, 20)
```

Еще раз обратите внимание, что режим доступа protected в реальности никак не ограничивает доступ к атрибутам объектов класса или самого класса. Например, вполне возможны команды:

```
print(rect._x2)     # 10
print(rect._fill)   # 'red'
```

Одно нижнее подчеркивание лишь предупреждает (сигнализирует) программиста о защищенном атрибуте, к которому напрямую лучше не обращаться. Так как этот атрибут, скорее всего, был создан для внутренней логики работы алгоритма в

классе и не предназначен для обращения извне. В дальнейшем, прямое обращение к нему может привести к проблемам, например, при изменении версии класса, в котором такого атрибута уже не будет, или он будет играть другую роль.

4.4.2. Атрибуты private и protected самих классов

Аналогичная ситуация с доступом к атрибутам private и protected, записанных непосредственно в классах. Вернемся к последней реализации классов Geom и Rect. После создания объекта rect:

```
rect = Rect(0, 0, 10, 20)
```

через него можно обращаться к protected-атрибуту _id_class базового класса Geom. Например, так:

```
print(rect._id_class)   # 1
```

Хотя, конечно, внешний доступ выполнять к таким свойствам не следует, а использовать исключительно внутри класса Geom или его дочерних классах.

Если же сделать этот атрибут приватным, то доступ к нему (по исходному имени) будет возможен только внутри класса Geom. Например:

```
class Geom:
    name = 'Geom'
    __id_class = 1

    def __init__(self, x1, y1, x2, y2):
        print(f"инициализатор Geom для {self.__class__}")
        self._x1 = x1
        self._y1 = y1
        self._x2 = x2
        self._y2 = y2
        self._id = self.__id_class  # ok


class Rect(Geom):
    def __init__(self, x1, y1, x2, y2, fill='red'):
        super().__init__(x1, y1, x2, y2)
        self._fill = fill
        self._id = self.__id_class  # AttributeError
```

При создании объекта Rect возникнет ошибка, т.к. в инициализаторе этого класса выполняется обращение к приватному атрибуту __id_class базового класса Geom. А вот объект базового класса Geom будет создаваться без ошибок, т.к. в его методах допустимо обращаться к приватным атрибутам текущего класса.

Так же работают ограничения доступа с методами, так как имена методов – это такие же атрибуты класса. Если в базовом классе Geom определить приватный метод, например, для проверки корректности значений координат, то он будет доступен только внутри этого класса и вызывать его, скажем, в дочернем классе

Rect уже не получится. Следующая программа демонстрирует такое поведение:

```
class Geom:
    name = 'Geom'
    __id_class = 1

    @staticmethod
    def __verify_coord(coord):
        return 0 <= coord <= 100

    def __init__(self, x1, y1, x2, y2):
        self._x1 = x1 if self.__verify_coord(x1) else None
        self._x2 = x2 if self.__verify_coord(x2) else None
        self._y1 = y1 if self.__verify_coord(y1) else None
        self._y2 = y2 if self.__verify_coord(y2) else None


class Rect(Geom):
    def __init__(self, x1, y1, x2, y2, fill='red'):
        super().__init__(x1, y1, x2, y2)
        super().__verify_coord(x1)       # AttributeError
        self._fill = fill


rect = Rect(0, 0, 10, 20)
```

При попытке создать объект rect возникнет ошибка:

```
AttributeError: 'super' object has no attribute '_Rect__verify_coord'
```

так как приватный метод __verify_coord, вызываемый в классе Rect, должен присутствовать в текущем классе.

С protected-методами таких ограничений нет. Они могут вызываться и в текущем классе и во всех его дочерних классах. Поэтому, если записать метод с одним подчеркиванием _verify_coord, то ошибка исчезнет, и объект rect будет успешно создан.

4.5. Полиморфизм и абстрактные методы

В ранних языках программирования, таких как C++ или Java, идея наследования классов открыла возможность оперировать разнотипными объектами через их единый базовый класс. Это позволило создавать единый интерфейс для обращения к общим свойствам и методам разнородных объектов, хранить их единым образом в коллекциях и, таким образом, унифицировать работу с ними. Такой подход в ООП получил название **полиморфизм** на уровне классов. В более краткой форме можно сказать, что:

Полиморфизм – это возможность работы с разными типами объектов (например, языка Python) через единый интерфейс (единым, унифицированным образом).

Давайте посмотрим, как идея полиморфизма реализуется в языке Python. Лучше всего это понять на конкретном примере. Для этого в программе объявим два класса Rectangle (прямоугольник) и Square (квадрат) следующим образом:

```
class Rectangle:
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def get_rect_pr(self):
        return 2*(self.w+self.h)


class Square:
    def __init__(self, a):
        self.a = a

    def get_sq_pr(self):
        return 4*self.a
```

В этих классах прописаны два метода-геттера get_rect_pr и get_sq_pr для получения периметров соответствующих фигур. Далее, можно создать экземпляры этих классов и вывести в консоль значения их периметров, например:

```
r1 = Rectangle(1, 2)
r2 = Rectangle(3, 4)
print(r1.get_rect_pr(), r2.get_rect_pr())

s1 = Square(10)
s2 = Square(20)
print(s1.get_sq_pr(), s2.get_sq_pr())
```

На экране увидим строчки:

```
6 14
40 80
```

Все работает, как и ожидалось. Однако если поместить все эти объекты в какую-либо коллекцию, например, список:

```
geom = [r1, r2, s1, s2]
```

а, затем, перебрать и посчитать для каждого периметр:

```
for g in geom:
    print(g.get_rect_pr())
```

то на объекте s1 возникнет ошибка:

```
AttributeError: 'Square' object has no attribute 'get_rect_pr'
```

указывающая, что в классе Square отсутствует метод get_rect_pr.

Конечно, зная, что в коллекции находятся объекты классов Rectangle и Square, в

цикле можно было бы записать следующую проверку:

```
for g in geom:
    if type(g) == Rectangle:
        print(g.get_rect_pr())
    else:
        print(g.get_sq_pr())
```

Тогда программа выполнится без ошибок. Но у такого подхода мало гибкости. Например, при добавлении еще одного класса геометрических фигур, допустим, треугольников:

```
class Triangle:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def get_tr_pr(self):
        return self.a + self.b + self.c
```

Тот же код перебора коллекции geom с объектами Triangle:

```
t1 = Triangle(4,3,5)
t2 = Triangle(7,8,11)
geom = [r1, r2, s1, s2, t1, t2]

for g in geom:
    if type(g) == Rectangle:
        print(g.get_rect_pr())
    else:
        print(g.get_sq_pr())
```

приведет снова к ошибке:

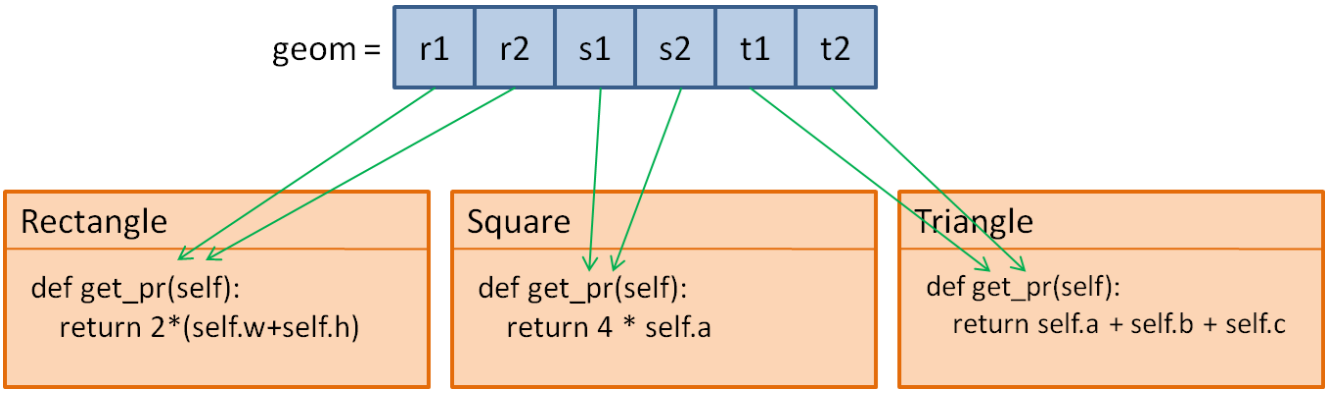
```
AttributeError: 'Triangle' object has no attribute 'get_sq_pr'. Did you
mean: 'get_tr_pr'?
```

Конечно, в цикл for можно добавить еще одну проверку на соответствие классу Triangle, но красоты и гибкости программе это не придаст. Можно сделать гораздо лучше, а именно, унифицировать во всех классах метод вычисления периметра фигуры и записывать его со следующей сигнатурой:

```
class Figure:
    def get_pr(): …
```

Тогда в цикле for достаточно обращаться к этому методу и получать значения периметров соответствующих фигур:

```
for g in geom:
    print(g.get_pr())
```



Получается, что на каждой итерации цикла объект g одного из классов геометрических фигур будет иметь доступ к своей реализации метода get_pr, т.к. переменная g будет ссылаться сначала на два объекта Rectangle, затем, на два объекта Square, и далее, на два объекта Triangle. Из каждого класса будет вызываться соответствующий метод get_pr со своим алгоритмом вычисления периметра, значение которого, затем, возвращается этим методом. Это и есть пример полиморфизма, когда с разными объектами работа осуществляется через единый интерфейс – переменную g с последующим вызовом геттера get_pr.

Обратите внимание, что список geom с объектами классов геометрических фигур можно сформировать непосредственно при его объявлении, например, так:

```
geom = [Rectangle(1, 2), Rectangle(3, 4),
        Square(10), Square(20),
        Triangle(1, 2, 3), Triangle(4, 5, 6)
        ]
```

В ряде случаев такой подход улучшает восприятие и понимание программного кода.

4.5.1. Использование исключения NotImplementedError

Полученная реализация классов Rectangle, Square, Triangle обладает одним существенным недостатком. Если забыть прописать в них метод get_pr или записать его с другой сигнатурой (например, добавить параметры), то дальнейшая программа завершится с ошибкой. Как можно было бы этого избежать? Один из вариантов определить базовый класс для Rectangle, Square, Triangle и в нем прописать реализацию геттера get_pr, используемую по умолчанию, например, следующим образом:

```
class Geom:
    def get_pr(self):
        return -1
```

А все остальные классы унаследовать от него:

```
class Rectangle(Geom):
    ...
```

```
class Square(Geom):
    ...


class Triangle(Geom):
    ...
```

Теперь, если в классе Triangle убрать метод get_pr, то для треугольников он будет вызываться из базового класса Geom и возвращать значение -1.

Но и это не самое лучшее решение. Все же, нам бы хотелось, чтобы каждый дочерний класс имел свою обязательную реализацию метода get_pr. Для этого можно использовать следующий прием. В методе get_pr базового класса Geom генерировать специальное исключение NotImplementedError, которое будет сигнализировать о необходимости переопределения метода в дочерних классах:

```
class Geom:
    def get_pr(self):
        raise NotImplementedError(f"В классе {self.__class__} должен быть
переопределен метод get_pr")
```

После запуска программы увидим следующее сообщение об ошибке:

```
NotImplementedError: В классе <class '__main__.Triangle'> должен быть
переопределен метод get_pr
```

Глядя на нее, программист сразу понимает, что было сделано не так и может оперативно, на этапе тестирования кода, внести необходимые исправления. В этом дополнительный плюс такого подхода.

4.5.2. Модуль abc (Abstract Base Classes)

В языках программирования методы, которые обязательно нужно переопределять в дочерних классах, и которые не имеют своей собственной реализации, называют **абстрактными**. Конечно, в языке Python нет чисто абстрактных методов, однако его стандартная библиотека содержит весьма полезный модуль abc (Abstract Base Classes), с помощью которого очень удобно имитировать абстрактные методы.

Полученная выше программа формирования объектов геометрических фигур с единым методом get_pr для вычисления их периметра, обладает тем недостатком, что ошибку NotImplementedError мы увидим не на этапе создания объектов, а на этапе вызова метода get_pr для соответствующего объекта. Такую ошибку можно не сразу отследить, а в более сложных случаях даже пропустить. Поэтому гораздо лучшим вариантом будет использование модуля abc для определения (имитации) абстрактных методов.

Вначале следует импортировать этот модуль, например, командой:

```
import abc
```

В этом модуле имеется класс-обертка ABC, который позволяет имитировать

объявление абстрактных методов с помощью декоратора abstractmethod:

```
class Geom(abc.ABC):
    @abc.abstractmethod
    def get_pr(self):
        pass
```

Теперь, если в каком-либо дочернем классе не переопределять абстрактный метод get_pr, например, в классе Triangle:

```
class Rectangle(Geom):
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def get_pr(self):
        return 2*(self.w+self.h)


class Square(Geom):
    def __init__(self, a):
        self.a = a

    def get_pr(self):
        return 4*self.a


class Triangle(Geom):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c
```

то при создании объектов этого класса:

```
t1 = Triangle(4, 3, 5)
t2 = Triangle(7, 8, 11)
```

возникнет ошибка:

```
TypeError: Can't instantiate abstract class Triangle with abstract method get_pr
```

Также теперь нельзя создавать отдельно объекты класса Geom, так как он содержит абстрактный метод:

```
g = Geom()
```

Появится ошибка:

```
TypeError: Can't instantiate abstract class Geom with abstract method get_pr
```

Если бы метод get_pr не был абстрактным, тогда никаких ошибок не возникало бы:

```
class Geom(abc.ABC):
    def get_pr(self):
        pass


g = Geom() # ok
```

Спрашивается, зачем тогда наследоваться от класса ABC? Дело в том, что декоратор abstractmethod и другие подобные работают только внутри классов, унаследованных от ABC. Если это наследование убрать:

```
class Geom:
    @abc.abstractmethod
    def get_pr(self):
        pass
```

то никакого эффекта от декоратора abstractmethod не будет. Он работает только в сочетании с базовым классом ABC.

Конечно, класс Geom может содержать не только абстрактные методы, но и любые другие конструкции, допустимые внутри классов, в том числе и обычные переменные и методы. Например:

```
class Geom(abc.ABC):
    def __init__(self, color=0, width=1):
        self.color = color
        self.width = width

    @abc.abstractmethod
    def get_pr(self):
        pass

    def get_data(self):
        return self.color, self.width
```

Здесь имеется один абстрактный метод get_pr, обычный метод get_data и стандартный инициализатор __init__.

Абстрактными можно также делать методы уровня класса и статические методы. Для этого нужно последовательно прописать два декоратора, например:

```
class Geom(abc.ABC):
    def __init__(self, color=0, width=1):
        self.color = color
        self.width = width

    @abc.abstractmethod
    def get_pr(self):
        pass

    @classmethod
    @abc.abstractmethod
    def get_limits(cls):
```

```
        pass

    @staticmethod
    @abc.abstractmethod
    def get_info():
        pass

    def get_data(self):
        return self.color, self.width
```

Соответственно, в дочерних классах все абстрактные методы должны быть переопределены. Например:

```
class Rectangle(Geom):
    MIN_VALUE = 0
    MAX_VALUE = 100

    def __init__(self, w, h, color=0, width=1):
        super().__init__(color, width)
        self.w = w
        self.h = h

    def get_pr(self):
        return 2 * (self.w + self.h)

    @classmethod
    def get_limits(cls):
        return cls.MIN_VALUE, cls.MAX_VALUE

    @staticmethod
    def get_info():
        return "Class Rectangle"
```

Использование абстрактных базовых классов весьма полезно при разработке больших проектов, где важно обеспечить согласованность структуры классов и интерфейсов. Это помогает избегать ситуаций, когда один разработчик ожидает определённый набор методов, а другой забывает реализовать какой-то важный элемент функциональности.

4.6. Множественное наследование

До сих пор мы с вами использовали цепочки наследования классов, когда один класс наследуется от другого, а тот в свою очередь, от третьего и т.д. Однако в языке Python допускается множественное наследование, когда один дочерний класс образуется сразу от нескольких базовых. В ряде задач такой подход дает свои дополнительные преимущества.

Несмотря на то, что множественное наследование применяется реже, чем обычное наследование от одного класса, тем не менее, это тоже важный механизм и некоторые подходы к программированию его активно используют. Например, идея (паттерн) миксинов (mixins) реализуется через множественное наследование.

И в нашем учебном примере мы рассмотрим пример его реализации.

Записывается множественное наследование в языке Python по следующему синтаксису:

```
class A(base1, base2, …, baseN):
…
```

Воспользуемся им и представим, что разрабатывается интернет-магазин по продаже товаров, например, ноутбуков, дисков, процессоров и т.п. Каждый товар определяется своим классом, а общим (базовым) для них всех будет класс Goods (товары). Также у каждого товара будут обязательные поля:

- name – наименование;
- weight – вес;
- price – цена.

Базовый класс Goods запишем в следующем виде:

```
class Goods:
    def __init__(self, name, weight, price):
        print("init Goods")
        self.name = name
        self.weight = weight
        self.price = price

    def __repr__(self):
        return f"{self.name}, {self.weight}, {self.price}"
```

И определим один дочерний класс для ноутбуков:

```
class Laptop(Goods):
    pass
```

Объект этого класса можно создать командой:

```
laptop = Laptop("Acer", 1.5, 51_000)
```

и вывести на экран:

```
print(laptop)
```

После запуска программы в консоли увидим строчки:

```
init Goods
Acer, 1.5, 51000
```

На данный момент вам все здесь должно быть понятно. Но потом, заказчик просит нас, например, добавить возможность логирования товаров. Как это лучше сделать? Первое, что приходит в голову – это прописать необходимые методы для логирования непосредственно в базовом классе Geom. Но, конечно же, лучше для этого создать еще один класс и добавить его в качестве еще одного базового для Laptop. Пусть в нашем примере класс, отвечающий за логирование, называется

MixinLog и имеет следующую (учебную) реализацию:

```
class MixinLog:
    ID = 0

    def __init__(self):
        print("init MixinLog")
        self.ID += 1
        self.id = self.ID

    def save_sell_log(self):
        print(f"{self.id}: товар продан в 00:00 часов")
```

Обратите внимание, что логика работы класса MixinLog совершенно независима от двух других классов: Goods и Laptop. Здесь лишь добавляется функционал по логированию товаров с добавлением атрибута id. Такие независимые базовые классы получили название миксинов (примесей).

Добавим этот класс в цепочку наследования:

```
class Laptop(Goods, MixinLog):
    pass
```

И после создания объекта laptop вызовем метод save_sell_log для формирования лога:

```
laptop = Laptop("Acer", 1.5, 51_000)
laptop.save_sell_log()
print(laptop)
```

Однако при выполнении этой программы получаем ошибку:

```
AttributeError: 'Laptop' object has no attribute 'id'. Did you mean: 'ID'?
```

Очевидно, она связана с тем, что у второго класса MixinLog не был вызван инициализатор. Почему так произошло? Как мы уже знаем, при создании объектов инициализатор ищется сначала в дочернем классе, но так как его там нет, то в первом базовом Goods. Так как он там прописан, то выполняется и на этом инициализация объекта Laptop завершается. Однако нам нужно также взывать инициализатор и второго базового класса MixinLog. В данном случае, сделать это можно с помощью объекта-посредника super, который перебирает базовые классы по цепочке наследования. Поэтому функция super, прописанная в инициализаторе базового класса Geom, возвратит ссылку на следующий класс MixinLog. Соответственно, через super можно вызвать инициализатор следующего класса в цепочке наследования, например, так:

```
class Goods:
    def __init__(self, name, weight, price):
        super().__init__()   # вызов инициализатора класса MixinLog
        print("init Goods")
        self.name = name
        self.weight = weight
```

```
        self.price = price
…
```

Теперь, после запуска программы, оба инициализатора срабатывают и на экране отображаются строчки:

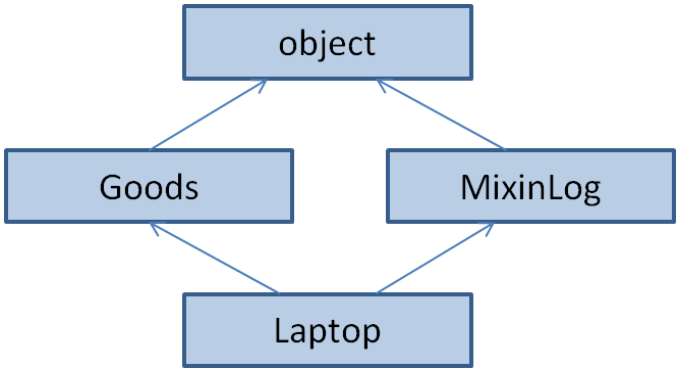
```
init MixinLog
init Goods
1: товар продан в 00:00 часов
Acer, 1.5, 51000
```

4.6.1. Порядок перебора базовых классов (MRO)

Но откуда функция super «знает», что нужно обратиться ко второму базовому классу MixinLog, а, скажем, не к базовому классу object, от которого неявно наследуются все классы верхнего уровня? В Python существует специальный алгоритм обхода базовых классов при множественном наследовании. Сокращенно, он называется:

MRO – Method Resolution Order

В нашем примере иерархия наследования классов имеет следующий вид:



А цепочка обхода базовых классов содержится в специальном атрибуте __mro__, который есть у каждого класса:

```
print(Laptop.__mro__)
```

После выполнения этой команды, в консоли отобразится кортеж с набором базовых классов, расположенных в порядке их обхода:

```
(<class '__main__.Laptop'>, <class '__main__.Goods'>, <class
'__main__.MixinLog'>, <class 'object'>)
```

То есть, атрибуты сначала ищутся в самом классе NoteBook, затем, в классе Goods, далее идет класс MixinLog и после него уже в классе object.

Конечно, это цепочка обхода будет меняться при формировании другой иерархии наследования. Но одно всегда неизменно – первый базовый класс, указанный при наследовании, выбирается первым (после дочернего). И это важный момент. Всегда можно быть уверенным, что инициализатор первого базового класса сработает в первую очередь. Почему это важно? Смотрите, при создании объекта

Laptop указываются три аргумента:

```
laptop = Laptop("Acer", 1.5, 51_000)
```

Они передаются в первый найденный инициализатор. В нашем примере – это инициализатор первого базового класса Goods. И, так как он гарантированно вызывается первым, то эти три аргумента попадают именно в него, а не в какой-то другой инициализатор других базовых классов. И какая бы цепочка наследования после Goods ни была, все равно первым будет вызываться метод __init__ класса Goods, потому что он записан первым. Это гарантирует работоспособность программы при разных иерархиях множественного наследования.

Давайте, для примера, поменяем местами базовые классы в Laptop:

```
class NoteBook(MixinLog, Goods):
    pass
```

В этом случае иерархия наследования в коллекции Laptop.__mro__ примет следующий вид:

```
(<class '__main__.Laptop'>, <class '__main__.MixinLog'>, <class
'__main__.Goods'>, <class 'object'>)
```

А при попытке создать объект этого класса:

```
laptop = Laptop("Acer", 1.5, 51_000)
```

возникает ошибка:

```
TypeError: MixinLog.__init__() takes 1 positional argument but 4 were given
```

так как в метод __init__ класса MixinLog передаются четыре аргумента, а он принимает только один. Поэтому порядок записи базовых классов при множественном наследовании играет важную роль. Первым должен идти «основной» класс и у него, как правило, инициализатор имеет несколько параметров. А далее, записываются классы, у которых, опять же, как правило, инициализаторы имеют только параметр self. И это второй важный момент. Когда реализуется множественное наследование, то структуру классов следует продумывать так, чтобы инициализаторы вспомогательных базовых классов имели только один параметр self, иначе возможны непредвиденные сложности их использования.

Чтобы это лучше понять, давайте, для примера, запишем следующий набор классов:

```
class Goods:
    def __init__(self, name, weight, price):
        super().__init__(1)
        self.name = name
        self.weight = weight
        self.price = price
```

```
class MixinLog:
    def __init__(self, p1):
        super().__init__(1, 2)


class MixinLog2:
    def __init__(self, p1, p2):
        super().__init__()


class Laptop(Goods, MixinLog, MixinLog2):
    pass
```

Здесь первый базовый класс Goods в своем инициализаторе вызывает метод __init__ следующего базового класса MixinLog с передачей одного аргумента, а следующий (по цепочке MRO) класс MixinLog в своем инициализаторе вызывает метод __init__ класса MixinLog2 уже с двумя аргументами. То есть, в этих классах жестко реализована последовательность их вызовов. И пока она соблюдается в описании дочернего класса Laptop, его объекты будут создаваться без каких-либо ошибок:

```
laptop = Laptop("Acer", 1.5, 51_000)
```

Однако если поменять порядок следования вспомогательных базовых классов MixinLog и MixinLog2, например, так:

```
class Laptop(Goods, MixinLog2, MixinLog):
    pass
```

то при создании объекта laptop получим ошибку:

```
TypeError: MixinLog2.__init__() missing 1 required positional argument: 'p2'
```

Или даже если убрать один последний базовый класс MixinLog2:

```
class Laptop(Goods, MixinLog):
    pass
```

то тоже возникнет ошибка:

```
TypeError: object.__init__() takes exactly one argument (the instance to
initialize)
```

Так вот, чтобы в программах при множественном наследовании не возникало подобных проблем с зависимостью последовательности наследования дополнительных базовых классов, их инициализаторы следует прописывать с одним параметром self и в каждом из них делать делегированный вызов инициализатора следующего класса. В нашем примере это будет выглядеть так:

```
class Goods:
    def __init__(self, name, weight, price):
        super().__init__()      # вызов инициализатора следующего класса
```

```
        self.name = name
        self.weight = weight
        self.price = price


class MixinLog:
    def __init__(self):         # инициализатор без параметров
        super().__init__()      # вызов инициализатора следующего класса


class MixinLog2:
    def __init__(self):         # инициализатор без параметров
        super().__init__()      # вызов инициализатора следующего класса
```

4.6.2. Вызов методов с одинаковыми именами из базовых классов

Следующий важный момент множественного наследования связан с возможностью появления атрибутов с одинаковыми именами в разных базовых классах. Как в такой ситуации будет отрабатывать программа? Чтобы ответить на этот вопрос, давайте в классы Goods и MixinLog добавим один и тот же метод с именем print_info:

```
class Goods:
    def __init__(self, name, weight, price):
        super().__init__()
        self.name = name
        self.weight = weight
        self.price = price

    def print_info(self):               # метод print_info класса Goods
        print("print_info класса Goods")


class MixinLog:
    def __init__(self):
        super().__init__()

    def print_info(self):               # метод print_info класса MixinLog
        print("print_info класса MixinLog")


class Laptop(Goods, MixinLog):
    pass
```

Создадим объект класса Laptop командой:

```
laptop = Laptop("Acer", 1.5, 51_000)
```

и через него вызовем метод print_info:

```
laptop.print_info()
```

На экране увидим строчку:

```
print_info класса Goods
```

То есть был вызван метод print_info класса Goods. И это логично, так как класс Goods записан первым в цепочке наследования и в соответствии с алгоритмом обхода MRO метод print_info сначала будет искаться в нем (разумеется, если такого метода нет в дочернем классе Laptop).

Но что если нужно вызвать метод print_info из второго базового класса MixinLog? Сделать это можно двумя способами. Либо напрямую вызвать этот метод через класс MixinLog командой:

```
MixinLog.print_info(laptop)
```

Обратите внимание, что в этом случае нам обязательно нужно указать первым аргументом ссылку на объект класса Laptop. Либо, определить такой же метод в классе Laptop и из него вызывать нужный метод базового класса. Например:

```
class Laptop(Goods, MixinLog):
    def print_info(self):
        MixinLog.print_info(self)
```

В этом случае команда:

```
laptop.print_info()
```

будет вызывать метод print_info класса Laptop, а он, в свою очередь, вызовет аналогичный метод из базового класса MixinLog.

4.7. Коллекция __slots__

В языке Python каждый объект класса имеет свой собственный набор локальных атрибутов, которые размещаются в отдельном словаре __dict__. Например, если объявить класс Point для представления точек на плоскости:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

то при создании его объекта:

```
pt = Point(1, 2)
```

в нем автоматически появляются локальные свойства x, y:

```
print(pt.__dict__)    # {'x': 1, 'y': 2}
```

Мало того, мы можем динамически добавлять новые локальные атрибуты:

```
pt.z = 4
```

и удалять существующие:

```
del pt.x
```

Это очень удобный инструментарий, который дает нам язык Python. Но за все приходится платить. Так как локальные атрибуты объекта размещаются в отдельном словаре __dict__, то, во-первых, происходит дополнительное выделение памяти под этот словарь (даже если он пустой) и, во-вторых, каждый словарь в Python реализуется на базе хэш-таблицы с доступом к атрибутам по их хэшам. Это не самый быстрый способ организации данных, но для обеспечения возможности динамического добавления/удаления атрибутов ничего лучшего пока не придумано. Однако если нам заранее известно, какой набор локальных свойств будет содержать объект определенного класса, и этот набор будет оставаться неизменным на всем протяжении существования объекта, то эти атрибуты можно было бы расположить непосредственно внутри объекта и обращаться к ним по адресным смещениям (по сути, по индексам), что значительно быстрее хэш-таблицы. Кроме того, здесь будет происходить некоторая экономия памяти, т.к. словарь __dict__ больше не понадобится. Именно этот функционал дает нам специальная коллекция __slots__, которая может быть прописана внутри любого класса.

Для примера объявим еще один класс Point2D, аналогичный классу Point, но с коллекцией __slots__, в которой в виде строк перечислены имена локальных атрибутов, размещаемых в его объектах:

```
class Point2D:
    __slots__ = ('x', 'y')

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Так как в коллекции __slots__ прописаны атрибуты с именами 'x' и 'y', то только такие локальные свойства могут существовать в экземплярах этого класса. Кроме того, его объекты не будут иметь коллекции __dict__. Давайте в этом убедимся. Создадим объект класса Point2D и выведем содержимое коллекции __dict__:

```
pt2 = Point2D(10, 20)
print(pt2.__dict__)
```

На экране появится ошибка:

```
AttributeError: 'Point2D' object has no attribute '__dict__'.
```

Действительно, объекты класса Point2D не содержат словаря __dict__, а все локальные атрибуты располагаются непосредственно в них. Поэтому команды:

```
x = pt2.x
y = pt2.y

pt2.x = 7
pt2.y = -5
```

позволяют совершенно спокойно читать и менять значения атрибутов, прописанных в коллекции __slots__. Но добавлять новые теперь не получится. Следующая команда приведет к ошибке:

```
pt2.z = 30  # AttributeError: 'Point2D' object has no attribute 'z'
```

Хотя, формально, удаление существующего атрибута выполнит можно, например:

```
del pt2.x   # имитация удаления атрибута
```

Но эта команда лишь пометит атрибут 'x' удаленным, в реальности он будет продолжать существовать в объекте pt2. Восстановить его можно путем присвоения какого-либо значения, например, так:

```
pt2.x = 5   # восстановление атрибута
```

Обратите внимание, что все приведенные операции справедливы исключительно для локальных атрибутов объектов, но не атрибутов класса. Сам класс Point2D будет иметь специальный словарь __dict__:

```
print(Point2D.__dict__)
```

а, значит, в него можно по-прежнему добавлять любые атрибуты и удалять существующие. Например, можно прописать переменную MAX_COORD со значением 100. Получим следующее объявление класса:

```
class Point2D:
    __slots__ = ('x', 'y')   # это для локальных атрибутов экземпляров
    MAX_COORD = 100          # это атрибут класса, а не объекта

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Соответственно, через объект этого класса:

```
pt2 = Point2D(10, 20)
```

можно обращаться к атрибуту класса MAX_COORD:

```
print(pt2.MAX_COORD)   # 100
```

Однако если имя атрибута класса будет совпадать с именем, записанным в коллекции __slots__, например, так:

```
class Point2D:
    __slots__ = ('x', 'y')   # это для локальных атрибутов экземпляров
    x = 100                  # это атрибут класса, а не объекта

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

то при ее выполнении получим ошибку:

```
ValueError: 'x' in __slots__ conflicts with class variable
```

Получается, что имена атрибутов классов не должны совпадать с именами локальных атрибутов, прописанных в __slots__. Это правило всегда следует соблюдать.

Как было сказано в самом начале, коллекция __slots__ помимо оптимизации доступа к локальным атрибутам, также приводит к уменьшению занимаемой объектом памяти. Например, если создать объект класса Point (без коллекции __slots__) и объект аналогичного класса Point2D (с коллекцией __slots__):

```
pt = Point(1, 2)
pt2 = Point2D(10, 20)
```

То размеры, которые занимают эти экземпляры в памяти устройства, можно вычислить следующими командами:

```
print("Размер объекта pt:", sys.getsizeof(pt) + sys.getsizeof(pt.__dict__))
print("Размер объекта pt2:", sys.getsizeof(pt2))
```

Получим значения:

```
Размер объекта pt: 352
Размер объекта pt2: 48
```

Таким образом, если в программе ожидается большое количество объектов какого-либо класса с фиксированным набором локальных атрибутов, то имеет смысл оптимизировать программу, воспользовавшись коллекцией __slots__.

Подводя общий итог, коллекция __slots__ меняет поведение экземпляров класса следующим образом:

- ограничивает набор локальных атрибутов;
- ускоряет доступ к локальным атрибутам;
- уменьшает размер занимаемой объектами памяти.

4.8. Коллекция __slots__ с объектами property и дескрипторами

Для работы с локальными атрибутами объектов в классе можно прописывать объекты-свойства property и/или дескрипторы. Ранее мы с вами их подробно уже рассматривали и отмечали, что приоритет обращения к дескрипторам данных (data descriptor) и объектам property выше, чем к локальным атрибутам экземпляров класса. Давайте посмотрим, как совместно работают эти инструменты с коллекцией __slots__.

Для примера воспользуемся все тем же классом Point2D, в котором имеется коллекция __slots__, но теперь с тремя именами локальных атрибутов:

```
class Point2D:
    __slots__ = ('x', 'y', 'length')
```

```
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.length = (x*x + y*y) ** 0.5
```

Как вы понимаете, если создать объект этого класса:

```
pt = Point2D(1, 2)
```

то в нем автоматически будет создан атрибут length с вычисленным значением:

```
print(pt.length)   # 2.23606797749979
```

Но что если в этом же классе прописать объект-свойство property с тем же именем length? Посмотрим, что произойдет:

```
class Point2D:
    __slots__ = ('x', 'y', 'length')

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.length = (x*x + y*y) ** 0.5

    @property
    def length(self):
        return self.length

    @length.setter
    def length(self, value):
        self.length = value
```

При выполнении этой программы получим ошибку:

```
ValueError: 'length' in __slots__ conflicts with class variable
```

И это ожидаемо, так как имена атрибутов классов не должны совпадать с именами, прописанными в __slots__. Поправить это достаточно просто, записав локальный атрибут length, как приватный (с двумя подчеркиваниями):

```
class Point2D:
    __slots__ = ('x', 'y', '__length')

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.length = (x*x + y*y) ** 0.5

    @property
    def length(self):
        return self.__length

    @length.setter
```

```
    def length(self, value):
        self.__length = value
```

Это позволит избежать конфликта имен, а через объект pt можно будет обращаться к публичному атрибуту класса length следующим образом:

```
pt = Point2D(1, 2)

res = pt.length     # обращение к объекту-свойству property
pt.length = 5       # обращение к объекту-свойству property
```

То же самое и с дескрипторами. Если для примера записать простой дескриптор данных:

```
class Digit:
    def __init__(self, name):
        self.name = name

    def __set__(self, instance, value):
        setattr(instance, self.name, value)

    def __get__(self, instance, owner):
        return getattr(instance, self.name)
```

то в классе Point2D он не должен конфликтовать с именами коллекции __slots__, поэтому реализацию класса запишем в виде:

```
class Point2D:
    x = Digit('_x')
    y = Digit('_y')

    __slots__ = ('_x', '_y')

    def __init__(self, x, y):
        self.x = x       # обращение к дескриптору
        self.y = y       # обращение к дескриптору
```

Обратите внимание, что при создании дескриптора Digit нужно правильно указывать имя локального атрибута. Оно должно присутствовать в коллекции __slots__. В данном случае передаются имена '_x' и '_y'. Это ключевой момент. Если указать любое другое имя, то получим ошибку AttributeError.

Далее с локальными атрибутами объекта класса Point2D можно работать напрямую через дескрипторы, например, так:

```
pt = Point2D(1, 2)

pt.y = 8
x = pt.x
```

Также обратите внимание, что в дескрипторе Digit доступ к локальным атрибутам объекта класса Point2D осуществляется через функции setattr и getattr. Все они

корректно работаю совместно с коллекцией __slots__.

4.9. Коллекция __slots__ при наследовании классов

Следующий важный момент – это особенности работы коллекции __slots__ при наследовании классов. Рассмотрим это на конкретных примерах.

Пусть определены два класса Point2D и дочерний от него класс Point3D со следующими реализациями:

```
class Point2D:
    __slots__ = ('x', 'y')

    def __init__(self, x, y):
        self.x = x
        self.y = y


class Point3D(Point2D):
    pass
```

Спрашивается, будет ли распространяться действие коллекции __slots__ на объекты дочернего класса Point3D? Давайте проверим. Создадим объект этого класса и выведем его словарь __dict__:

```
pt3 = Point3D(10, 20)
print(pt3.__dict__)
```

После выполнения программы, на экране увидим пустой словарь:

```
{}
```

Отсюда можно сделать вывод, что локальные атрибуты x, y объекта pt3 располагаются непосредственно в нем, а не в словаре __dict__ со всеми вытекающими отсюда последствиями:

```
print(pt3.x)   # 10
print(pt3.y)   # 20
```

Но наличие __dict__ означает, что в объекте pt3 можно динамически создавать новые атрибуты, например:

```
pt3.z = 7
```

которые размещаются в __dict__. То есть, получается, что объекты класса Point3D будут содержать в себе локальные свойства x, y из коллекции __slots__, но дополнительно могут иметь произвольное количество других атрибутов, которые располагаются в словаре __dict__.

Если требуется, чтобы объекты класса Point3D вели себя так же, как и объекты класса Point2D, то есть, не имели бы коллекции __dict__, то достаточно в класс поместить __slots__ с пустым кортежем:

```
class Point3D(Point2D):
    __slots__ = ()
```

В этом случае объекты класса Point3D смогут работать исключительно с атрибутами x, y, указанные в коллекции __slots__ базового класса:

```
pt3 = Point3D(10, 20)

print(pt3.x)  # 10
print(pt3.y)  # 20
```

То есть, коллекция __slots__ дочернего класса лишь расширяет аналогичную коллекцию базового класса. Поэтому, разрешив в дочернем классе Point3D еще один локальный атрибут, например, с именем z:

```
class Point3D(Point2D):
    __slots__ = ('z', )
```

объект pt3 сможет оперировать тремя атрибутами x, y, z:

```
pt3 = Point3D(10, 20)
pt3.x = 1
pt3.y = 2
pt3.z = 3
```

Обратите внимание на висячую запятую в коллекции __slots__. Она означает, что z – это элемент кортежа, а не просто строка в круглых скобках.

Раз в классе Point3D появился дополнительный разрешенный атрибут, то имеет смысл добавить в него инициализатор для записи значений в x, y, z:

```
class Point3D(Point2D):
    __slots__ = ('z', )

    def __init__(self, x, y, z):
        super().__init__(x, y)    # вызов инициализатора класса Point2D
        self.z = z
```

В результате, экземпляры этого класса можно создавать командой:

```
pt3 = Point3D(10, 20, 30)
```

А что будет, если в базовом классе Point2D убрать коллекцию __slots__? И записать все в следующем виде:

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y


class Point3D(Point2D):
    __slots__ = ('z', )
```

```
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z
```

В этом случае при создании объекта класса Point3D никаких ошибок не возникнет и, кроме того, появится словарь __dict__, в который попадут свойства x, y:

```
pt3 = Point3D(10, 20, 30)
print(pt3.__dict__)       # {'x': 10, 'y': 20}
```

Локальный атрибут 'z' будет располагаться внутри объекта pt3. Этот пример показывает, что как только в иерархии наследования появляется класс без __slots__, то итоговый объект (дочернего класса) приобретает словарь __dict__ с возможностью динамического создания локальных атрибутов. Исключение здесь представляет базовый класс object, от которого неявно наследуются все классы.

ГЛАВА V. ДАТА-КЛАССЫ (DATA CLASSES)

В программах на Python нередко приходится объявлять небольшие классы для хранения данных в экземплярах с минимальной логикой их обработки. При этом каждый раз приходится прописывать инициализатор для формирования локальных атрибутов, возможно, операцию сравнения на равенство (магический метод __eq__) и понятный вывод информации об объекте (магический метод __repr__). Если таких классов много, то все это превращается в настоящую рутинную работу, которую неплохо бы было автоматизировать. Именно для этого в Python 3.7 появился модуль dataclasses, который заметно упрощает описание таких классов, а сами классы, объявленные через него, стали называться дата-классами (data classes).

В этой главе мы подробно рассмотрим порядок объявления и особенности работы дата-классов, используя инструментарий модуля dataclasses.

5.1. Введение в дата-классы (data classes)

Довольно часто при объявлении классов для хранения определенного набора данных в программах на Python приходится прописывать инициализатор следующего вида:

```
class Thing:
    def __init__(self, name, weight, price):
        self.name = name
        self.weight = weight
        self.price = price
```

В данном случае при создании объекта класса Thing в инициализатор передаются три аргумента: name, weight, price и формируются локальные свойства с аналогичными именами внутри объекта. Соответственно, объект этого класса можно создать командой:

```
t = Thing("Учебник по Python", 100, 1024)
```

и после вывода его на экран:

```
print(t)
```

увидим малоинформативную строку:

```
<__main__.Thing object at 0x000001B120CB7A10>
```

Поэтому нередко в таких классах с данными прописывается магический метод __repr__, например, следующим образом:

```
class Thing:
    def __init__(self, name, weight, price):
        self.name = name
        self.weight = weight
        self.price = price
```

```
    def __repr__(self):
        return f"Thing: {self.name=}, {self.weight=}, {self.price=}"
```

В результате получается большой объем однотипного программного кода, который было бы неплохо оптимизировать с точки зрения описания таких «классов с данными» в тексте программы. Поэтому, начиная с версии Python 3.7, появился инструмент в виде стандартного модуля dataclasses, позволяющий значительно упростить описание дата-классов (data classes). Для этого из модуля dataclasses следует импортировать декоратор dataclass, например, командой:

```
from dataclasses import dataclass
```

А затем использовать его для упрощенного описания дата-класса. В результате, аналог класса Thing можно объявить следующим образом:

```
@dataclass
class ThingData:
    name: str
    weight: int
    price: float
```

Получилось довольно компактное описание класса ThingData с набором полей name, weight, price. Как это работает? Декорирование с помощью dataclass автоматически добавляет в класс ThingData инициализатор с тремя параметрами (name, weight, price), магический метод __repr__ и метод __eq__ для сравнения на равенство объектов между собой. Чтобы декоратор dataclass «знал», какие локальные атрибуты должны создаваться внутри каждого объекта класса, они перечисляются в виде атрибутов класса с обязательным аннотированием их типа данных. Таким образом, класс ThingData обладает необходимым базовым функционалом для сохранения в своих объектах данных name, weight и price. Чтобы в этом убедиться, создадим экземпляр класса ThingData командой:

```
td = ThingData("Учебник по Python", 100, 1024)
```

Обратите внимание, в качестве первого аргумента передается строка, а вторые два – это вес и цена. То есть, параметры в инициализаторе по умолчанию следуют в том же порядке, что и атрибуты, указанные при описании класса ThingData.

Если теперь вывести на экран объект td с помощью функции print:

```
print(td)
```

то увидим вполне информативную строку:

```
ThingData(name='Учебник по Python', weight=100, price=1024)
```

Это сработал магический метод __repr__, который был автоматически добавлен в класс ThingData декоратором dataclass. Благодаря этому нам не нужно тратить время на написание типового, рутинного кода. Да и само объявление класса в программе визуально выглядит куда лучше. Главное правильно понимать такую запись.

Помимо метода __repr__ в класс был добавлен метод __eq__, который сравнивает объекты на равенство. Давайте проверим его работу. Создадим еще один объект с тем же самым содержимым:

```
td2 = ThingData("Учебник по Python", 100, 1024)
```

и выполним команду:

```
print(td == td2)  # True
```

Увидим значение True. То есть, здесь выполняется сравнение данных в объектах, а не их идентификаторов, как это делается по умолчанию методом __eq__ класса object.

Если же, хотя бы одно поле объекта будет содержать другие данные, то такие объекты считаются не равными, например:

```
td = ThingData("Учебник по Python", 100, 1024)
td2 = ThingData("Учебник по Python", 200, 1024)
print(td == td2)   # False
```

По сути, эта операция работает по аналогии сравнения двух кортежей между собой:

```
res = (name1, weight1, price1) == (name2, weight2, price2)
```

Если это стандартное поведение функции __eq__ нас не устраивает, то ее можно явно прописать в дата-классе ThingData с требуемым функционалом. Пусть, например, сравнение выполняется только по полю name, получим:

```
@dataclass
class ThingData:
    name: str
    weight: int
    price: float

    def __eq__(self, other):
        return self.name == other.name
```

В этом случае, явно объявленный метод __eq__, не меняется декоратором dataclass. И сравнение двух объектов будет выполняться только по полю name:

```
td = ThingData("Учебник по Python", 100, 1024)
td2 = ThingData("Учебник по Python", 200, 1024)
print(td == td2)   # True
```

И так происходит со всеми явно прописанными методами, которые допустимо определять внутри дата-классов, – они не меняются декоратором dataclass.

5.2. Атрибуты со значениями по умолчанию в data classes

Следующий важный базовый момент при определении data classes, состоит в порядке объявления атрибутов с начальными значениями. Давайте предположим,

что цена по умолчанию принимает значение 0. В этом случае класс ThingData можно объявить следующим образом:

```
@dataclass
class ThingData:
    name: str
    weight: int
    price: float = 0
```

После создания объекта этого класса и вывода его на экран:

```
td = ThingData("Учебник по Python", 100)
print(td)
```

в консоли увидим строку:

```
ThingData(name='Учебник по Python', weight=100, price=0)
```

Ожидаемо, атрибут price принимает значение по умолчанию 0. Если же третьим аргументом указать какое-либо значение, например:

```
td = ThingData("Учебник по Python", 100, 512)
```

то теперь переменная price будет равна 512:

```
ThingData(name='Учебник по Python', weight=100, price=512)
```

Но здесь есть один маленький нюанс: все атрибуты со значениями по умолчанию должны идти последними. Например, если поменять атрибуты weight и price местами:

```
@dataclass
class ThingData:
    name: str
    price: float = 0
    weight: int
```

то получим ошибку:

```
TypeError: non-default argument 'weight' follows default argument
```

Почему так произошло? Почему атрибуты с начальными значениями обязательно должны записываться последними? Вы, наверняка, уже догадались, в чем дело. Так как эти атрибуты, по сути, являются параметрами метода __init__ и идут в том порядке, в котором записаны, то при указанной их записи в классе, получаем следующее объявление инициализатора:

```
class ThingData:
    def __init__(self, name: str, price: float = 0, weight: int):
        ...
```

Однако параметры со значениями по умолчанию должны идти последними при описании методов. Отсюда и возникает ошибка. Поэтому, при объявлении дата-классов через декоратор dataclass, все атрибуты с начальными значениями

должны быть записаны в конце.

Также следует иметь в виду, что в качестве начальных значений лучше использовать неизменяемые типы данных. Почему это так важно? Мы с вами об этом уже говорили. В качестве примера рассмотрим объявление класса Thing, у которого в инициализаторе прописан параметр dims со значением в виде пустого списка:

```
class Thing:
    def __init__(self, name, weight, price, dims=[]):
        self.name = name
        self.weight = weight
        self.price = price
        self.dims = dims
```

Создадим объект этого класса и через атрибут dims в список добавим значение 10:

```
t = Thing("Учебник по Python", 100, 1024)
t.dims.append(10)   # [10]
```

Пока все ожидаемо, список содержит единственное значение 10. Но, если создать еще один объект этого же класса:

```
t2 = Thing("Учебник по Python", 100, 1024)
```

и вывести его вариант списка dims:

```
print(t2.dims)   # [10]
```

то и у него увидим значение 10.

Чаще всего инициализация параметров изменяемыми типами данных приводит к непредвиденным ошибкам, т.к. начальный пустой список подразумевается независимым для каждого объекта класса Thing. Тогда как в приведенном примере он оказался общим.

По этой причине при объявлении data classes атрибутам нельзя присваивать изменяемые типы данных в качестве значений по умолчанию. Например, следующее объявление приведет к ошибке:

```
@dataclass
class ThingData:
    name: str
    weight: int
    price: float = 0
    dims: list = []    # ValueError
```

5.3. Функция field()

Однако если нам все же требуется использовать пустой список в качестве начального значения параметра, причем так, чтобы он был независимым для каждого объекта класса ThingData, то как это сделать? Для таких целей в модуль dataclasses была добавлена функция field, которую вначале нужно импортировать,

например, командой:

```
from dataclasses import dataclass, field
```

А затем с ее помощью определить начальное значение параметра, например, следующим образом:

```
@dataclass
class ThingData:
    name: str
    weight: int
    price: float = 0
    dims: list = field(default_factory=list)
```

Как работает эта функция? После создания объекта класса она вызывает функцию, прописанную в параметре default_factory, для формирования начального значения соответствующего параметра. В нашем примере, параметр dims будет ссылаться на пустой список. Причем этот список будет формироваться независимо для каждого объекта класса ThingData. В этом легко убедиться, выполнив следующие строчки:

```
th1 = ThingData("Учебник по Python", 100, 1024)
th1.dims.append(10)
th2 = ThingData("Учебник по Python", 100, 1024)

print(th1.dims)   # [10]
print(th2.dims)   # []
```

В параметр default_factory можно передавать ссылки на любые функции, которые вызываются без аргументов. Например, в классе ThingData начальное значение атрибута dims можно формировать с помощью лямбда-функции:

```
@dataclass
class ThingData:
    name: str
    weight: int
    price: float = 0
    dims: list = field(default_factory=lambda: [0, 0, 0])
```

Если теперь сформировать объект этого класса и вывести его на экран:

```
th = ThingData("Python", 100, 1024)
print(th)
```

то увидим строчку:

```
ThingData(name='Python', weight=100, price=1024, dims=[0, 0, 0])
```

Конечно, вместо лямбда-функции допустимо указывать ссылку на аналогичную обычную функцию, например, такую:

```
def initial_dims():
    return [0, 0, 0]
```

```
@dataclass
class ThingData:
    name: str
    weight: int
    price: float = 0
    dims: list = field(default_factory=initial_dims)
```

А вот использовать статический метод текущего класса не получится. Например, следующая программа выполнится с ошибкой NameError:

```
@dataclass
class ThingData:
    name: str
    weight: int
    price: float = 0
    dims: list = field(default_factory=ThingData.initial_dims)

    @staticmethod
    def initial_dims():
        return [0, 0, 0]
```

так как класс ThingData на момент вызова функции field еще не сформирован, а значит, не существует. Как вариант, можно создать вложенный класс и из него передавать ссылки на статические методы. Например, так:

```
@dataclass
class ThingData:
    class GoodsMethodsFactory:
        @staticmethod
        def initial_dims():
            return [0, 0, 0]

    name: str
    weight: int
    price: float = 0
    dims: list = field(default_factory=GoodsMethodsFactory.initial_dims)
```

Какой варианты выбрать? Решает сам программист, исходя из удобства описания программного кода.

Вообще, функция field предоставляет богатый функционал по управлению объявляемых атрибутов в дата-классах и содержит следующие основные параметры:

- default_factory – фабрика по формированию начальных значений (обычно изменяемых типов) с помощью вызываемых объектов (например: list, dict);
- default – значение по умолчанию с использованием неизменяемых типов (например: строки, числа);
- init – булево значение True/False указывает использовать ли атрибут, как параметр в инициализаторе __init__() (по умолчанию True);

- repr – булево значение True/False указывает использовать ли атрибут в магическом методе __repr__() (по умолчанию True);
- compare – булево значение True/False указывает использовать ли атрибут при сравнении объектов (по умолчанию True).

Смысл этих параметров достаточно прост. Давайте для примера объявим класс Vector3D с четырьмя следующими атрибутами:

```
@dataclass
class Vector3D:
    x: int = field(repr=False)
    y: int
    z: int = field(compare=False)
    length: float = field(init=False, compare=False, default=0.0)
```

Первый атрибут x будет исключен из метода __repr__, третий атрибут z исключен из метода __eq__, а четвертый length исключен из инициализатора, метода сравнения и будет иметь начальное значение 0.0. Давайте в этом убедимся. Создадим два следующих объекта класса Vector3D:

```
v1 = Vector3D(1, 2, 3)
v2 = Vector3D(1, 2, 5)
```

выведем на экран первый объект и результат их сравнения на равенство:

```
print(v1)
print(v1 == v2)
```

Получим строчки:

```
Vector3D(y=2, z=3, length=0.0)
True
```

Как видите, метод __repr__ пропустил значение x при выводе информации об объекте; значения атрибутов z и length не участвуют при сравнении объектов; по умолчанию переменная length принимает значение 0.0 и не включена в параметры инициализатора. Все так, как было определено параметрами функции field.

5.4. Метод __post_init__()

Давайте теперь представим, что требуется определить дата-класс, который бы повторял логику работы следующего класса Vector_3D:

```
class Vector_3D:
    def __init__(self, x: int, y: int, z: int):
        self.x = x
        self.y = y
        self.z = z
        self.length = (x * x + y * y + z * z) ** 0.5
```

Обратите внимание, что здесь локальный атрибут length вычисляется на основе параметров x, y, z. Спрашивается, как это можно сделать при объявлении data

classes? Так как атрибут length не участвует в параметрах инициализатора, то начальное описание дата-класса Vector3D можно сделать следующим:

```
@dataclass
class Vector3D:
    x: int
    y: int
    z: int
    length: float = field(init=False, compare=False, default=0.0)
```

Однако нам нужно, чтобы значение length вычислялось на основе переданных данных в параметры x, y, z инициализатора класса. То есть, вычислять length нужно после создания объекта класса в момент его инициализации. Для этого метод __init__, сформированный декоратором dataclass, последней строчкой содержит вызов еще одного вспомогательного метода __post_init__, который можно явно прописать в дата-классе. В нашем примере его реализация будет иметь вид:

```
@dataclass
class Vector3D:
    x: int
    y: int
    z: int
    length: float = field(init=False, compare=False, default=0.0)

    def __post_init__(self):
        self.length = (self.x*self.x + self.y*self.y + self.z*self.z)**0.5
```

Теперь, после создания объекта и вывода его на экран:

```
v = Vector3D(1, 2, 3)
print(v)
```

увидим следующие значения:

```
Vector3D(x=1, y=2, z=3, length=3.7416573867739413)
```

Мало того, атрибут length можно динамически создать непосредственно в методе __post_init__, не прописывая его в атрибутах класса:

```
@dataclass
class Vector3D:
    x: int
    y: int
    z: int

    def __post_init__(self):
        self.length = (self.x*self.x + self.y*self.y + self.z*self.z)**0.5
```

Правда, тогда он не будет выводиться методом __repr__, но будет присутствовать в коллекции __dict__:

```
v = Vector3D(1, 2, 3)
```

```
print(v)           # Vector3D(x=1, y=2, z=3)
print(v.__dict__)  # {'x': 1, 'y': 2, 'z': 3, 'length': 3.7416573867739413}
```

И это понятно, т.к. реализация метода __repr__ строится декоратором dataclass на основе указанных атрибутов класса. И когда атрибут length убирается, то и в методе __repr__ он не учитывается. Поэтому, если нужен вывод атрибута, то его лучше указывать в атрибутах класса, либо, прописывать свою реализацию метода __repr__. Правда, тогда смысл определения дата-классов будет нивелироваться (уменьшаться), т.к. мы вручную начинаем повторять работу декоратора dataclass.

5.4.1. Параметры типа InitVar

Иногда при определении классов в инициализаторе прописываются вспомогательные параметры, которые не сохраняются внутри объекта. Например, нам нужно в классе Vector_3D вычислять значение атрибута length, если параметр calc_len равен True, а иначе приравнять length нулю. Здесь вспомогательный параметр calc_len не нужно сохранять в объекте, а использовать непосредственно в методе __init__. При обычном определении класса это можно реализовать следующим образом:

```
class Vector_3D:
    def __init__(self, x: int, y: int, z: int, calc_len: bool = True):
        self.x = x
        self.y = y
        self.z = z
        self.length = (x * x + y * y + z * z) ** 0.5 if calc_len else 0
```

А как это сделать при объявлении дата-классов? Для определения параметров, участвующих только в инициализации (таких, как calc_len), в модуле dataclasses предусмотрен специальный класс (тип) InitVar:

```
from dataclasses import dataclass, field, InitVar
```

Если при объявлении атрибут аннотируется этим типом, то он, во-первых, не сохраняется в объекте и, во-вторых, автоматически передается как параметр в метод __post_init__, чтобы им можно было воспользоваться при формировании вычисляемых свойств:

```
@dataclass
class Vector3D:
    x: int
    y: int
    z: int
    calc_len: InitVar[bool] = True
    length: float = field(init=False, compare=False, default=0)

    def __post_init__(self, calc_len: bool):
        if calc_len:
            self.length = (self.x*self.x+self.y*self.y+self.z*self.z)** 0.5
```

В результате, после создания объекта класса и отображения его на экран:

```
v = Vector3D(1, 2, 3)
print(v)
print(v.__dict__)
```

увидим следующие строчки:

```
Vector3D(x=1, y=2, z=3, length=3.7416573867739413)
{'x': 1, 'y': 2, 'z': 3, 'length': 3.7416573867739413}
```

Здесь присутствуют только четыре атрибута x, y, z, length. Атрибут calc_len отсутствует в объекте v, так как он был аннотирован специальным типом InitVar.

5.5. Параметры декоратора dataclass

До сих пор мы с вами использовали декоратор dataclass с параметрами по умолчанию. Однако ему можно передавать различные аргументы и управлять процессом формирования итогового класса. Вот основные параметры, которые принимает декоратор dataclass:

Параметр	**Описание**
init = [True | False]	Принимает булево значение, по умолчанию True. Если значение True, то в классе объявляется инициализатор, иначе – не объявляется.
repr = [True | False]	Принимает булево значение, по умолчанию True. Если значение True, то в классе объявляется магический метод __repr__(), иначе – не объявляется.
eq = [True | False]	Принимает булево значение, по умолчанию True. Если значение True, то в классе объявляется магический метод __eq__(), иначе – не объявляется.
order = [True | False]	Принимает булево значение, по умолчанию False. Если значение True, то в классе объявляются магические методы для операций сравнения <; <=; >; >=, иначе – не объявляются.
unsafe_hash = [True | False]	Влияет на формирование магического метода __hash__()
frozen = [True | False]	Принимает булево значение, по умолчанию False. Если значение True, то атрибуты объектов класса становятся неизменными (можно только проинициализировать один раз в инициализаторе).
slots = [True | False]	Принимает булево значение, по умолчанию False. Если значение True, то атрибуты объявляются в коллекции __slots__.

Существуют и другие параметры. Подробно о них можно почитать на странице официальной документации:

https://docs.python.org/3/library/dataclasses.html

Давайте, для примера, рассмотрим основные из них. Если в декоратор передать аргумент init=False:

```
@dataclass(init=False)
class Vector3D:
    x: int
    y: int
    z: int
```

то класс будет сформирован без собственного инициализатора (будет использован инициализатор базового класса). Поэтому создание объектов такого класса возможно только командой:

```
v = Vector3D()
print(v.__dict__)   # {}
```

Обратите внимание, что магический метод __repr__ при отображении объекта v будет обращаться к локальным свойствам x, y, z, которых не существует. В результате, команда:

```
print(v)
```

приведет к ошибке:

```
AttributeError: 'Vector3D' object has no attribute 'x'
```

Это следует учитывать, прописывая аргумент init=False. Однако, прописав атрибуты класса со значениями по умолчанию:

```
@dataclass(init=False)
class Vector3D:
    x: int = 0
    y: int = 0
    z: int = 0
```

ошибка пропадет, так как в самом классе Vector3D будут присутствовать атрибуты x, y, z:

```
v = Vector3D()
print(v)           # Vector3D(x=0, y=0, z=0)
print(v.__dict__)  # {}
```

То есть, метод __repr__ использует атрибуты класса, а не объекта.

Чтобы такой неопределенной ситуации не происходило, дополнительно из дата-класса можно убрать метод __repr__ с помощью аргумента repr=False:

```
@dataclass(init=False, repr=False)
class Vector3D:
```

```
    x: int
    y: int
    z: int
```

Тогда команда отображения объекта на экране будет использовать метод __repr__ базового класса object:

```
v = Vector3D()
print(v)           # <__main__.Vector3D object at 0x00000209A1F46FD0>
print(v.__dict__)  # {}
```

То же самое и с магическим методом __eq__. Так как локальные атрибуты внутри объекта отсутствуют, то команда сравнения на равенство:

```
v1 = Vector3D()
v2 = Vector3D()
print(v1 == v2)  # AttributeError: 'Vector3D' object has no attribute 'x'
```

приведет к ошибке:

```
AttributeError: 'Vector3D' object has no attribute 'x'
```

Поэтому данный метод нужно либо переопределить, либо исключить из дата-класса путем передачи в декоратор аргумента eq=False:

```
@dataclass(init=False, repr=False, eq=False)
class Vector3D:
    x: int
    y: int
    z: int
```

В результате объекты будут сравниваться по их идентификаторам с помощью метода __eq__ базового класса object.

Следующий параметр order позволяет добавлять в дата-класс набор магических методов для различных операций сравнения. Он может принимать значение True только совместно с eq=True. Например, следующее объявление:

```
@dataclass(eq=False, order=True)
class Vector3D:
    x: int
    y: int
    z: int
```

приведет к ошибке:

```
ValueError: eq must be true if order is true
```

Здесь либо нужно убрать параметр eq (по умолчанию принимает значение True), либо у него прописать значение True:

```
@dataclass(eq=True, order=True)
class Vector3D:
    x: int
```

```
    y: int
    z: int
```

Это связано с тем, что различные операции сравнения используют метод __eq__ в своей работе, поэтому он должен присутствовать в классе.

Итак, с объектами класса Vector3D теперь можно дополнительно выполнять сравнения на «больше» и «меньше». Например:

```
v1 = Vector3D(1, 2, 30)
v2 = Vector3D(1, 3, 4)

print(v1 < v2)   # True
print(v1 > v2)   # False
```

Сравнение выполняется на уровне кортежей, содержащих значения атрибутов в порядке их объявления в классе, например:

```
res = (x1, y1, z1) < (x2, y2, z2)
```

То есть, происходит последовательное сравнение сначала значений `x` между собой, затем, `y` и в конце `z`. Как только встречается пара, для которой можно вычислить значение True или False, проверка завершается. В данном примере `y1 < y2`, поэтому операция «меньше» возвращает True.

Если из операций сравнений нужно исключить какие-либо атрибуты, то для этого следует использовать функцию field с параметром compare=False, например:

```
@dataclass(eq=True, order=True)
class V3D:
    x: int
    y: int = field(compare=False)
    z: int
```

В результате, объекты будут сравниваться по локальным атрибутам x и z:

```
v1 = Vector3D(1, 2, 30)
v2 = Vector3D(1, 3, 4)

print(v1 < v2)   # False
print(v1 > v2)   # True
```

Обратите внимание на то, что если в классе объявить какой-либо магический метод сравнения на «больше», «меньше», «больше либо равно» или «меньше либо равно», то возникнет исключение TypeError:

```
@dataclass(order=True)
class Vector3D:
    x: int
    y: int
    z: int

    def __lt__(self, other):
```

```
        return self.x < other.x and self.y < other.y
```

При выполнении этой программы увидим ошибку:

```
TypeError: Cannot overwrite attribute __lt__ in class Vector3D. Consider
using functools.total_ordering
```

Однако переопределять методы сравнения недопустимо только при аргументе `order=True`. Если же он принимает значение по умолчанию False, то никаких ошибок возникать не будет.

Последний параметр frozen, который мы рассмотрим, позволяет «замораживать» значения атрибутов класса, то есть, делает их неизменяемыми. Например:

```
@dataclass(frozen=True)
class Vector3D:
    x: int
    y: int
    z: int
    dims: list = field(init=False, default_factory=list)
```

Тогда для любого объекта этого класса:

```
v = Vector3D(1, 2, 3)
```

команда изменения значения локального атрибута:

```
v.x = 5
```

приведет к ошибке:

```
dataclasses.FrozenInstanceError: cannot assign to field 'x'
```

Но, если меняется не сам атрибут, а состояние объекта, на который он ссылается, например, список:

```
v.dims.append(5)
```

то никаких ошибок возникать не будет. Причем ограничение касается именно атрибутов объектов, а не класса, поэтому вполне возможна команда:

```
Vector3D.attr = 'python'
```

Эти нюансы следует иметь в виду при «заморозке» локальных атрибутов экземпляров класса.

5.6. Наследование дата-классов (data calsses)

Дата-классы – это обычные классы языка Python, только объявленные через декоратор dataclass. Поэтому к ним применимы все ранее изученные подходы ООП, включая механизм наследования.

Давайте в качестве примера объявим базовый класс с именем Goods для представления различных товаров интернет-магазина:

```
from dataclasses import dataclass, field, InitVar
from typing import Any


@dataclass
class Goods:
    uid: Any
    price: Any = None
    weight: Any = None
```

И определим дочерний от него дата-класс Book для книг:

```
@dataclass
class Book(Goods):
    title: str = ""
    author: str = ""
    price: float = 0
    weight: int | float = 0
```

Смотрите, здесь в дочернем классе повторяются атрибуты price и weight. Спрашивается, какие инициализаторы в итоге будут формироваться в первом и во втором классах? Для первого базового класса Goods все очевидно. Будет объявлен инициализатор с сигнатурой:

```
class Goods:
    def __init__(self, uid: Any, price: Any = None, weight: Any = None):
        ...
```

А вот дочерний класс Book будет иметь инициализатор вида:

```
class Book:
    def __init__(self, uid: Any, price: float = 0,
                 weight: int | float = 0,
                 title: str = "", author: str = ""):
        ...
```

То есть, вначале идут параметры в соответствии с базовым классом Goods, а затем, параметры текущего класса Book. Причем их типы и начальные значения берутся из дочернего класса Book (если какие-либо параметры переопределяются в нем). Поэтому, создавая объект класса Book командой:

```
b = Book(1)
print(b)
```

увидим строчку:

```
Book(uid=1, price=0, weight=0, title='', author='')
```

Или же, можно создать объект, указав весь набор аргументов:

```
b = Book(1, 1000, 100, "Python ООП", "Балакирев С.М.")
print(b)
```

В консоли увидим строчку:

```
Book(uid=1, price=1000, weight=100, title='Python ООП', author='Балакирев С.М.')
```

Понимая теперь, как формируются параметры в инициализаторах дочерних классов, атрибуты должны быть записаны так, чтобы вначале шли параметры без значений по умолчанию, а затем, все остальные. Например, если дочерний класс Book записать в виде:

```
@dataclass
class Book(Goods):
    title: str            # параметр title без начального значения
    author: str = ""
    price: float = 0
    weight: int | float = 0
```

то при выполнении программы получим ошибку:

```
TypeError: non-default argument 'title' follows default argument
```

так как параметры в инициализаторе класса Book принимают вид:

```
class Book:
    def __init__(self, uid: Any, price: float = 0,
                 weight: int | float = 0,
                 title: str, author: str = ""):
        ...
```

Здесь параметр title без значения идет после параметров со значениями. На этот момент следует обращать внимание, при описании дочерних дата-классов.

5.6.1. Вызов метода __post_init__() в иерархии дата-классов

Давайте несколько усложним нашу программу и будем формировать значение uid автоматически при создании каждого нового объекта. Сделаем это следующим образом:

```
from dataclasses import dataclass, field, InitVar
from typing import Any, ClassVar


@dataclass
class Goods:
    current_uid: ClassVar[int] = 0

    uid: int = field(init=False)
    price: Any = None
    weight: Any = None

    def __post_init__(self):
        print("Goods: post_init")
        Goods.current_uid += 1
        self.uid = Goods.current_uid
```

Здесь в класс Goods добавлен атрибут current_uid типа ClassVar и с начальным значением 0. Тип ClassVar означает, что объявляется атрибут класса, а не объекта. Соответственно, он не попадает в параметры инициализатора и локальные атрибуты экземпляров класса. Далее, атрибут uid исключается из параметров инициализатора и вычисляется в методе __post_init__. В этом же методе выводится строчка `"Goods: post_init"` (сделано в учебных целях, чтобы показать вызов данного метода).

Дочерний класс Book оставим без изменений. Создадим объект этого класса и выведем его в консоль:

```
b = Book(1000, 100, "Python ООП", "Балакирев С.М.")
print(b)
```

Увидим следующие строчки:

```
Goods: post_init
Book(uid=1, price=1000, weight=100, title='Python ООП', author='Балакирев С.М.')
```

Первая строка показывает, что был вызван метод __post_init__ базового класса, в котором сформировалось свойство uid со значением 1.

Давайте посмотрим, что произойдет, если добавить такой же метод __post_init__ в дочерний класс Book, например, так:

```
@dataclass
class Book(Goods):
    title: str = ""
    author: str = ""
    price: float = 0
    weight: int | float = 0

    def __post_init__(self):
        print("Book: post_init")
```

При запуске программы получаем ошибку:

```
AttributeError: 'Book' object has no attribute 'uid'
```

сообщающую, что свойство uid отсутствует в объекте b. Почему так произошло? Дело в том, что после добавления метода __post_init__ в дочерний класс именно он и будет вызывается инициализатором класса Book. При этом вызов такого же метода базового класса не происходит. Чтобы это поправить, нужно прописать делегированный вызов __post_init__ для базового класса следующим образом:

```
@dataclass
class Book(Goods):
    title: str = ""
    author: str = ""
    price: float = 0
    weight: int | float = 0
```

```
    def __post_init__(self):
        super().__post_init__()
        print("Book: post_init")
```

Теперь никаких проблем с начальной инициализацией вычисляемых свойств базового класса не возникает, и на экране отображаются строчки:

```
Goods: post_init
Book: post_init
Book(uid=1, price=1000, weight=100, title='Python ООП', author='Балакирев С.М.')
```

Обратите внимание, что для формирования локальных атрибутов не нужно самим прописывать вызов инициализатора базового класса. Это делается автоматически декоратором dataclass. Если бы мы формировали цепочку наследования без использования декоратора dataclass (обычным способом), то вызов инициализаторов базовых классов нужно было бы прописывать явно.

5.7. Функция make_dataclass()

Помимо явного объявления дата-классов через параметр dataclass, есть еще один способ создания таких классов с помощью функции make_dataclass, которая имеет следующие основные параметры:

- cls_name – название нового класса (строка);
- fields – поля (локальные атрибуты) объектов класса;
- bases – список базовых классов;
- namespace – словарь для определения атрибутов самого класса (например, так можно объявлять методы класса).

Преимущество использования функции make_dataclass перед явным объявлением в том, что она формирует классы в процессе работы программы, а не заранее в тексте программы.

Давайте, для примера, с помощью функции make_dataclass объявим класс аналогичный следующему:

```
class Car:
    def __init__(self, model, max_speed, price):
        self.model = model
        self.max_speed = max_speed
        self.price = price

    def get_max_speed(self):
        return self.max_speed
```

Это можно сделать следующей командой:

```
from dataclasses import make_dataclass, field


CarData = make_dataclass("CarData",
                         [("model", str),
```

```
                           "max_speed",
                           ("price", float, field(default=0))],
                 namespace={'get_max_speed': lambda self: self.max_speed})
```

Здесь "CarData" имя создаваемого класса; затем идет список из кортежей или строк – набор локальных атрибутов объекта класса; параметр namespace определяет атрибут класса get_max_speed в виде метода.

Создадим объект этого класса:

```
car = CarData("BMW", 256, 4096)
```

Выведем его на экран и вызовем метод get_max_speed:

```
print(car)
print(car.get_max_speed())
```

В консоли увидим строчки:

```
CarData(model='BMW', max_speed=256, price=4096)
256
```

Конечно, функцию make_dataclass имеет смысл использовать лишь для динамического создания классов в процессе работы программы. В противном случае, гораздо удобнее их прописывать явно через декоратор dataclass.

Если вам на практике потребуется использовать функцию make_dataclass, то более подробную информацию о ней можно посмотреть на странице официальной документации:

https://docs.python.org/3/library/dataclasses.html

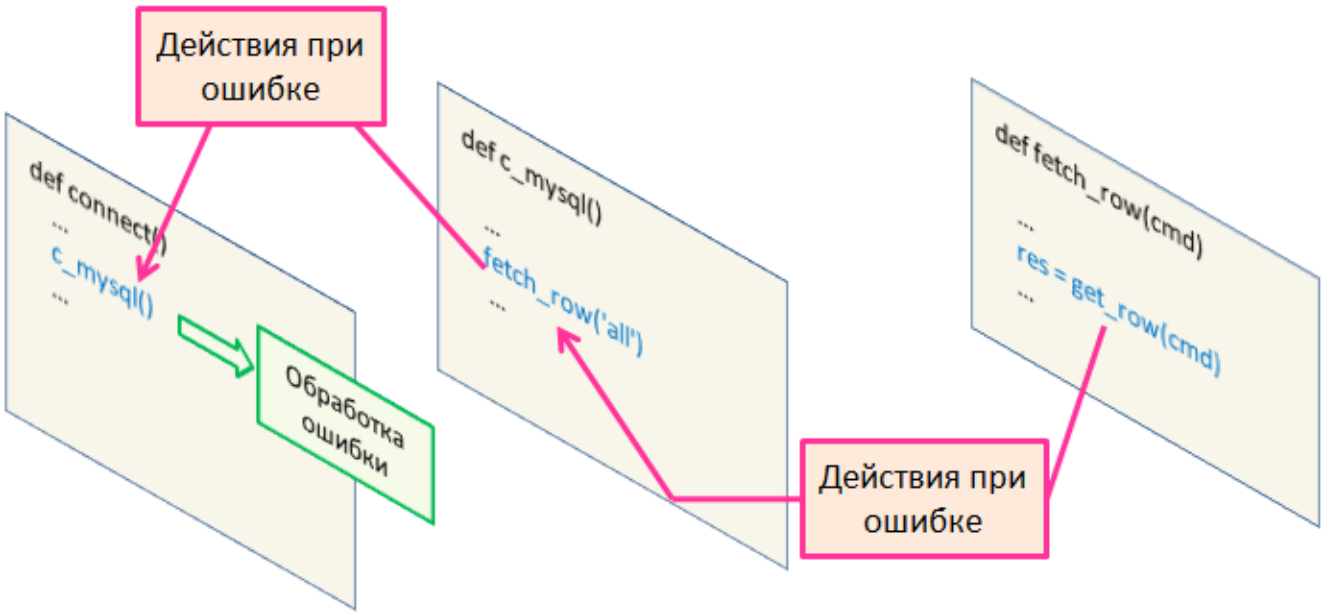
ГЛАВА VI. ОБРАБОТКА ИСКЛЮЧЕНИЙ. МЕНЕДЖЕРЫ КОНТЕКСТА

При разработке программ мы с вами постоянно сталкивались с возникающими ошибками, например, AttributeError, TypeError, SyntaxError и т.п. Эти ошибки также называются исключениями, которые можно разбить на две группы:

- исключения в момент исполнения программы (например: IndexError, KeyError, AttributeError, ValueError);
- исключения при компиляции (до выполнения программы, обычно это SyntaxError).

Так вот, ошибки первой группы можно перехватывать на программном уровне и выполнять определенные действия, при их возникновении. Это называется обработкой исключений. Реализуется она операторами try/except/finally, о которых пойдет речь в этой главе.

Первый закономерный вопрос, зачем вообще понадобился такой отдельный инструмент для обработки ошибок? Смотрите, при выполнении прикладных программ, зачастую, происходит вложенный вызов различных функций. И глубина этих вызовов может быть довольно большой.



Предположим, что на каком-либо уровне вызова возникла ошибочная ситуация. Например, деление на ноль, или не найден файл, или ошибка связи с сервером, и так далее. Мало зафиксировать ошибку, нужно еще указать программе, как ей выполняться дальше. И вот здесь кроется главная проблема. Часто отдельные функции лишь выполняют свои локальные задачи, фиксируя возможные ошибки. Однако не всегда имеют возможность сразу же их обработать, то есть, «сказать» программе, что нужно сделать, при их возникновении. Например, не найден указанный файл, или недостаточно прав для записи в него данных. Что в этом случае должна делать вспомогательная функция? Часто ответ можно дать лишь на уровне общей логики программы или специально предназначенных для этого функциях. То есть, возникшую ошибку нужно передать обратно по стеку вызова,

пока не найдется такая функция, которая в состоянии ее обработать. При этом выполнение других функций должно быть прекращено.

Это довольно типовая ситуация и до появления механизма обработки исключений решалась либо возвратом специального значения из функции, либо через специальные глобальные переменные, либо каким-либо другим изощренным способом. Дополнительные сложности здесь представляют вызовы методов через объекты классов. Каждый объект и класс – это отдельная и независимая программная единица и вопрос обработки возникающих ошибок становится особенно актуальным. Все это, в итоге, привело к необходимости разработки отдельного механизма обработки исключительных ситуаций, который бы брал на себя реализацию логики выполнения программы при той или иной непредвиденной ошибке.

6.1. Введение в обработку исключений. Операторы try/except

Давайте предположим, что в программе строчка за строчкой выводится некоторый многострочный текст:

```
print("Я к вам пишу – чего же боле?")
print("Что я могу еще сказать?")
print("Теперь, я знаю, в вашей воле")
print(a)
print("Меня презреньем наказать.")
print("Но вы, к моей несчастной доле")
print("Хоть каплю жалости храня,")
print("Вы не оставите меня.")
```

Здесь в четвертой строчке выводится переменная `a`, которая не существует. В результате возникает ошибка (исключение) NameError:

```
Я к вам пишу – чего же боле?
Что я могу еще сказать?
Теперь, я знаю, в вашей воле
NameError: name 'a' is not defined
```

Это пример исключения в момент выполнения программы. Причем типы исключений могут быть самыми разными, в зависимости от возникающей ошибки. Например, если попробовать выполнить деление на ноль:

```
10/0
```

то отобразится исключение типа ZeroDivisionError:

```
ZeroDivisionError: division by zero
```

которое так же возникает при выполнении программы.

Другой вид ошибок часто связан неверным синтаксисом команд. Если, например, записать строчки:

```
if True
```

```
    print("True")
```

то увидим исключение:

```
SyntaxError: expected ':'
```

которое возникает в момент компиляции текста программы (язык Python перед выполнением программ компилирует их в байт-код). При синтаксических ошибках программа не выполняется, поэтому перехватывать их не представляется возможным.

На первый взгляд может показаться, что исключения возникают по вине программиста. Но это не всегда так. Классический пример обработки исключений связан с открытием файлов на чтение или запись. Когда мы с вами рассматривали тему файлов, то говорили, что команда:

```
file = open("myfile.txt")
```

таит в себе потенциальную ошибку FileNotFoundError, связанную с недоступностью указанного файла. Поэтому такой критический код следует помещать в специальный блок try с обработкой исключения FileNotFoundError, например:

```
try:
    file = open("myfile2.txt")
except FileNotFoundError:
    print("Невозможно открыть файл")
```

Или, при делении одного числа на другое, заранее нельзя предугадать значение переменной y, которая может быть равна нулю. В этом случае возникает исключение ZeroDivisionError, которое также можно перехватить:

```
try:
    x, y = map(int, input().split())
    res = x / y
except ZeroDivisionError:
    print("Делить на ноль нельзя!")
```

Причем здесь может появиться еще один тип исключения ValueError, если с клавиатуры ввести не целые числа, а что-то другое. Чтобы отловить сразу два исключения допустимо прописывать несколько блоков except, например, так:

```
try:
    x, y = map(int, input().split())
    res = x / y
except ZeroDivisionError:
    print("Делить на ноль нельзя!")
except ValueError:
    print("Ошибка типа данных")
```

И таких блоков except может быть множество. Причем они просматриваются в порядке их записи сверху-вниз и, как только, какой-либо из них срабатывает,

остальные пропускаются. То есть, каждый раз выполняться будет только какой-то один блок except из всего набора, а все остальные после него игнорируются.

Чтобы отловить в одном блоке except сразу несколько типов исключений, их следует записывать в круглых скобках через запятую после ключевого слова except следующим образом:

```
try:
    x, y = map(int, input().split())
    res = x / y
except (ZeroDivisionError, ValueError):
    res = "деление на ноль или нечисловое значение"
```

Но чаще используют раздельную запись, так как при разных ошибках выполняются разные действия.

Если при возникновении ошибок необходимо взаимодействовать с объектами классов исключений, то доступ к ним записывается через ключевое слово as, например, так:

```
try:
    x, y = map(int, input().split())
    res = x / y
except ZeroDivisionError as z:
    print(z)
except ValueError as z:
    print(z)
```

То есть, после имени класса-исключения ставится ключевое слово as и, затем, переменная, которая будет ссылаться на объект соответствующего класса (в нашем примере ValueError или ZeroDivisionError).

Если сейчас запустить эту программу и с клавиатуры набрать два числа `"1 0"`, то на экране появится строчка:

```
division by zero
```

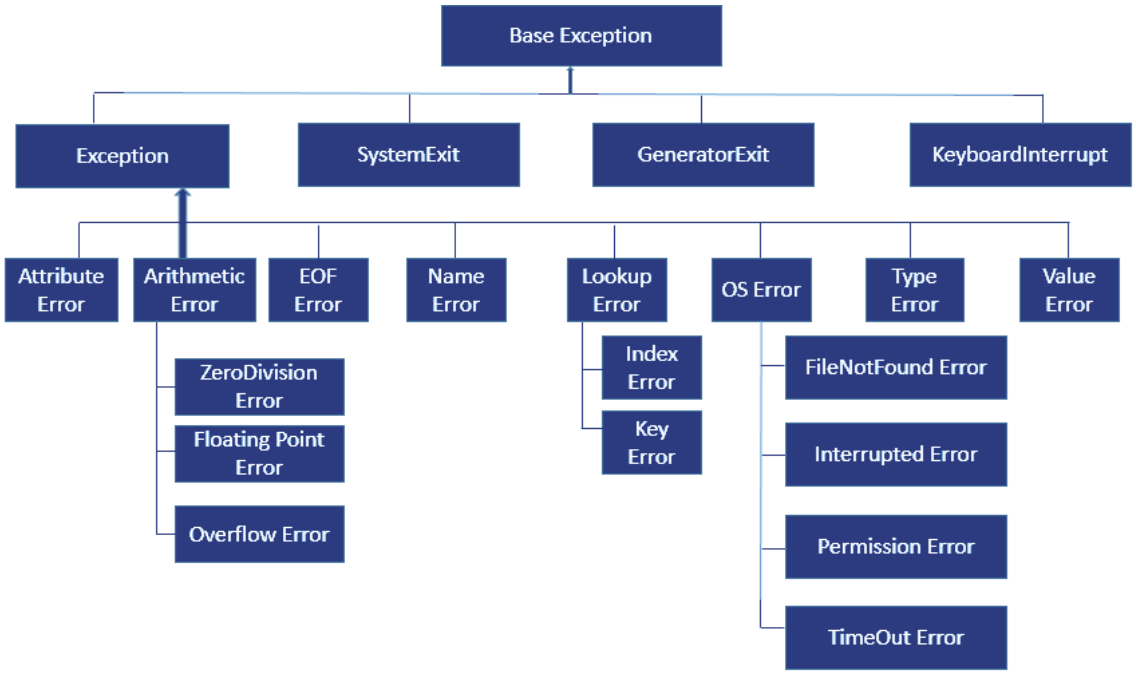
Это результат вывода объекта z класса ZeroDivisionError.

В общем случае существует иерархия классов исключений, во главе которой стоит базовый класс BaseException. Ниже приведен фрагмент этой иерархии. Самих классов, конечно же, значительно больше. Со всем списком подробно можно ознакомиться на странице официальной документации:

https://docs.python.org/3/builtins/exceptions.html

Из приведенного рисунка хорошо видно, что большая часть классов наследуется от базового класса Exception, и именно эта ветка нас будет интересовать, так как в ней расположены типы исключений, возникающие в процессе работы программы. Но на что влияет эта иерархия и как ее можно использовать при обработке исключений? Например, из рисунка видно, что исключение деления на ноль

(ZeroDivisionError) относится к базовому классу ArithmeticError, а он, в свою очередь, к базовому классу Exception.



Благодаря этой цепочке наследования, в блоках except вместо указания конкретного типа ZeroDivisionError можно прописывать любой из базовых классов, например, ArithmeticError и этот тип исключения будет срабатывать при возникновении любых дочерних от него типов: ZeroDivisionError, FloatingPointError и других.

Давайте, для примера, в блоке except вместо ZeroDivisionError запишем его базовый класс ArithmeticError:

```
try:
    x, y = map(int, input().split())
    res = x / y
except ArithmeticError as z:
    print(z)
except ValueError as z:
    print(z)
```

Если при выполнении этой программы набрать два числа `"1 0"`, то на экране появится все та же ошибка:

```
division by zero
```

В действительности, переменная z в этом примере ссылается на класс ZeroDivisionError, а не на ArithmeticError первого блока except. Но, благодаря тому, что ArithmeticError является базовым по отношению к ZeroDivisionError, то объекты этого типа будут отлавливаться первым блоком except.

Но здесь есть один важный нюанс. Если вместо ArithmeticError прописать более общий класс Exception, который является базовым для ArithmeticError и

ValueError:

```
try:
    x, y = map(int, input().split())
    res = x / y
except Exception as z:
    print(z)
except ValueError as z:
    print(z)
```

то в первый блок except будут попадать практически все виды исключений. Следовательно, второй блок except с ValueError никогда не будет срабатывать. Поэтому здесь есть одно простое правило. Сначала прописываются блоки со специализированными классами исключений, а затем, с более общими (базовыми). В соответствии с этим правилом, в программе выше следует поменять блоки except местами, получим:

```
try:
    x, y = map(int, input().split())
    res = x / y
except ValueError as z:
    print(z)
except Exception as z:
    print(z)
```

Теперь блок с ValueError будет отлавливать свой тип исключения, а второй блок – все остальные, унаследованные от класса Exception.

Если же требуется записать блок except, который отлавливает все виды программных ошибок, то он прописывается без каких-либо классов, например, следующим образом:

```
try:
    x, y = map(int, input().split())
    res = x / y
except:
    print("Ошибка")
```

Конечно, такой блок всегда следует записывать самым последним, если используется набор из нескольких блоков except.

6.2. Блоки finally и else оператора try. Вложенные блоки try

На данный момент мы с вами познакомились с основами работы оператора try/except, а также увидели, как правильно использовать иерархию классов исключений для их корректной обработки. Однако простой конструкции try/except часто бывает недостаточно для корректной обработки возникающих ошибок. Поэтому на практике нередко дополнительно используют блоки else и finally. Давайте посмотрим, как они работают и для чего нужны.

Итак, блок try может содержать необязательный блок else, который выполняется

при штатном выполнении кода внутри блока try, то есть, когда не произошло никаких ошибок. Например, при запуске следующей программы и ввода с клавиатуры ненулевых целых чисел:

```
try:
    x, y = map(int, input().split())
    res = x / y
except ZeroDivisionError as z:
    print(z)
except ValueError as z:
    print(z)
else:
    print("Исключений не произошло")
```

увидим сообщение:

```
Исключений не произошло
```

Если же возникает какая-либо ошибка, то блок else не выполняется.

Другим необязательным блоком является блок finally, который, наоборот, выполняется всегда после блока try, вне зависимости от того, произошла ошибка или нет. Например, при выполнении следующей программы (при вводе любых данных):

```
try:
    x, y = map(int, input().split())
    res = x / y
except ZeroDivisionError as z:
    print(z)
except ValueError as z:
    print(z)
else:
    print("Исключений не произошло")
finally:
    print("Блок finally выполняется всегда")
```

всегда будем видеть сообщение:

```
Блок finally выполняется всегда
```

Здесь часто возникает вопрос, зачем нужен этот блок, если он выполняется всегда после try? Кажется, что с тем же успехом можно записать команды из блока finally сразу после блока try/except, например, так:

```
try:
    x, y = map(int, input().split())
    res = x / y
except ZeroDivisionError as z:
    print(z)
except ValueError as z:
    print(z)
else:
```

```
    print("Исключений не произошло")

print("Блок finally выполняется всегда")
```

Однако между последними двумя реализациями есть существенное отличие. Если, например, убрать блок except с исключением ValueError и запустить программу с вводом нечисловых значений, то возникнет необработанное исключение ValueError, но блок finally все равно выполнился! А вот команды, записанные после блока try, выполняться уже не будут.

Но все же, в каких ситуация может потребоваться использовать блок finally? Классический пример – это работа с файлами. Вначале, для открытия файла следует вызвать функцию open в блоке try, например, так:

```
try:
    f = open("myfile.txt")
    f.write("hello")
except FileNotFoundError as z:
    print(z)
except:
    print("Другая ошибка")
```

Если файл «myfile.txt» существует и успешно открывается, то из файла f можно только читать данные, т.к. по умолчанию функция open открывает файловый поток на чтение данных. Поэтому при выполнении следующей команды `f.write()` возникнет ошибка и файловый поток останется открытым. А мы с вами уже знаем, что любой файл нужно закрывать даже при возникновении программных ошибок. Как раз эту логику удобно прописать в блоке finally следующим образом:

```
f = None

try:
    f = open("myfile.txt")
    f.write("hello")
except FileNotFoundError as z:
    print(z)
except:
    print("Другая ошибка")
finally:
    if f:  # проверка, что файловый поток вообще существует
        f.close()
        print("Файл закрыт")
```

После запуска программы (при наличии файла «myfile.txt» в рабочем каталоге), увидим, что файл закрывается при любых исходах, что очень удобно.

Конечно, при работе с файлами такую громоздкую конструкцию заменяют более компактной с использованием файлового менеджера, который автоматически закрывает файл:

```
try:
    with open("myfile.txt") as f:
        f.write("hello")
except FileNotFoundError as z:
    print(z)
except:
    print("Другая ошибка")
```

Предыдущий вариант с блоком finally приведен исключительно в учебных целях, чтобы показать особенности и ценность использования этого блока. Существует множество задач, где применение блока finally позволяет устранять дублирование кода, повышает читаемость текста программ и упрощает ее последующее редактирование.

Следующая особенность работы блока finally связана с обработкой исключений внутри функций. Предположим, что в функции вводятся два целых числа, которые возвращаются в виде кортежа:

```
def get_values():
    try:
        x, y = map(int, input().split())
        return x, y
    except ValueError as v:
        print(v)
        return 0, 0
    finally:
        print("finally выполняется до return")


x, y = get_values()
print(x, y)
```

Здесь может возникнуть исключение ValueError, когда вводятся не целые числа. В этом случае возвращается кортеж с двумя нулями. Также прописан блок finally с выводом сообщения «finally выполняется до return», чтобы мы убедились, что этот блок действительно выполняется до оператора return. После запуска программы, и ввода значений (неважно каких), видим, что блок finally действительно срабатывает до оператора return. Это очень важный момент, который нужно хорошо знать, при использовании блоков try/except/finally. Причем внутри блока finally оператор return обычно не прописывают, это считается плохой практикой программирования.

Последняя особенность блоков try/except, которую мы рассмотрим, заключается в том, что их можно вкладывать один в другой. Например, следующая программа отдельным вложенным блоком try отлавливает ошибку деления на ноль и обрабатывает ее в блоке except. Если же возникает ошибка ValueError, то она обрабатывается внешним блоком try:

```
try:
    x, y = map(int, input().split())
```

```
    try:
        res = x / y
    except ZeroDivisionError:
        print("Деление на ноль")
except ValueError as z:
    print("Ошибка ValueError")
```

Вложения в программе могут появляться и в неявном виде. Например, внутренний блок try располагается в некоторой отдельной функции:

```
def div(a, b):
    try:
        return x / y
    except ZeroDivisionError:
        return "Деление на ноль"
```

А затем она вызывается в другом, внешнем блоке try:

```
try:
    x, y = map(int, input().split())
    res = div(x, y)
except ValueError as z:
    print("Ошибка ValueError")

print(res)
```

Получаем схожую ситуацию вложенных блоков try. Работают они очевидным образом. Сначала отрабатывает внутренний блок. Если ошибка им перехватывается, то дальше она не распространяется. Иначе, исключение переходит к внешнему блоку try.

6.3. Инструкция raise и пользовательские исключения

Во всех наших предыдущих примерах исключения автоматически возникали в результате ошибочных ситуаций, например, деления на ноль:

```
print("Куда ты скачешь, гордый конь,")
print("И где опустишь ты копыта?")
print("О мощный властелин судьбы!")
1/0
print("Не так ли ты над самой бездной")
print("На высоте, уздой железной")
print("Россию поднял на дыбы?")
```

Но как эта операция деления формирует само исключение? Для этого в языке Python имеется специальный оператор `raise`, который порождает различные типы исключений. Например, чтобы сгенерировать ошибку ZeroDivisionError достаточно записать команду:

```
raise ZeroDivisionError("Деление на ноль")
```

И если такая ошибка не обрабатывается, то в консоли увидим:

```
ZeroDivisionError: Деление на ноль
```

Выполнение программы при этом останавливается на конструкции raise. В качестве сообщения об ошибке отображается та строка, которая указана в ZeroDivisionError.

Вообще, после оператора raise можно прописывать любой класс исключения с определенным набором аргументов, либо совсем не указывать аргументы, например, так:

```
raise ZeroDivisionError
```

В этом случае сообщение об ошибке будет иметь вид:

```
ZeroDivisionError
```

В последнем примере также создается экземпляр класса, но без аргументов. И, разумеется, никто нам не мешает сформировать объект-исключения отдельной командой:

```
e = ZeroDivisionError("Деление на ноль")
```

а затем прописать его в команде raise:

```
raise e
```

Причем в качестве исключения можно использовать любой класс, унаследованный от базового класса BaseException. Например, если записать недопустимый, объект-строку:

```
raise "деление на ноль"
```

то интерпретатор Python как раз на это и укажет:

```
TypeError: exceptions must derive from BaseException
```

То есть, после raise должен находиться экземпляр класса исключения, а не какой-то произвольный объект.

Когда может потребоваться явно прописывать в программе оператор raise? И разве сам язык Python не генерирует автоматически все необходимые исключения? Конечно, во всех типовых ситуациях именно так и происходит. Например, если выполнять операции:

```
1 + "2"         # TypeError
(1, 2, 3)[4]    # IndexError
```

то автоматически возникают ошибки заданного типа. Однако прописать исключения на все случаи жизни невозможно. Например, разрабатывается класс PrintData, отвечающий за работу с принтером (гипотетический, учебный пример):

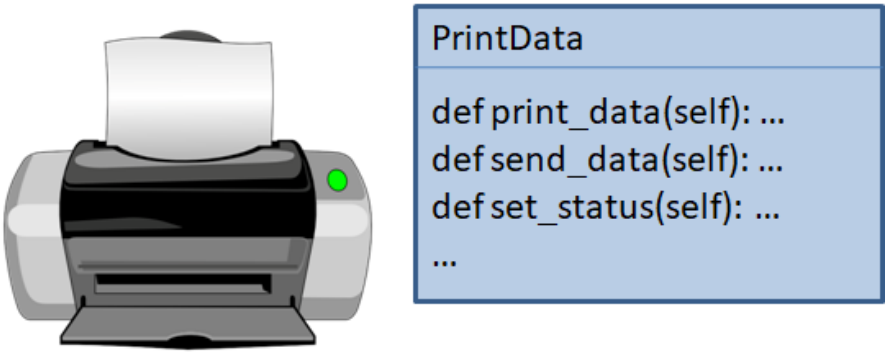
```
class PrintData:
    def print_data(self, data):
        self.send_data(data)
```

```
        print(f"печать: {data}")

    def send_data(self, data):
        if not self.send_to_print(data):
            raise Exception("принтер не отвечает")

    def send_to_print(self, data):
        return False
```

Здесь метод send_data генерирует исключение с помощью класса Exception, если по каким-то причинам данные не были отправлены принтеру.



Далее, это исключение может быть обработано на любом уровне стека вызова функций. Например, если создать экземпляр этого класса и вызвать метод print:

```
p = PrintData()
p.print_data("123")
```

то увидим следующий стек возникшей ошибки:

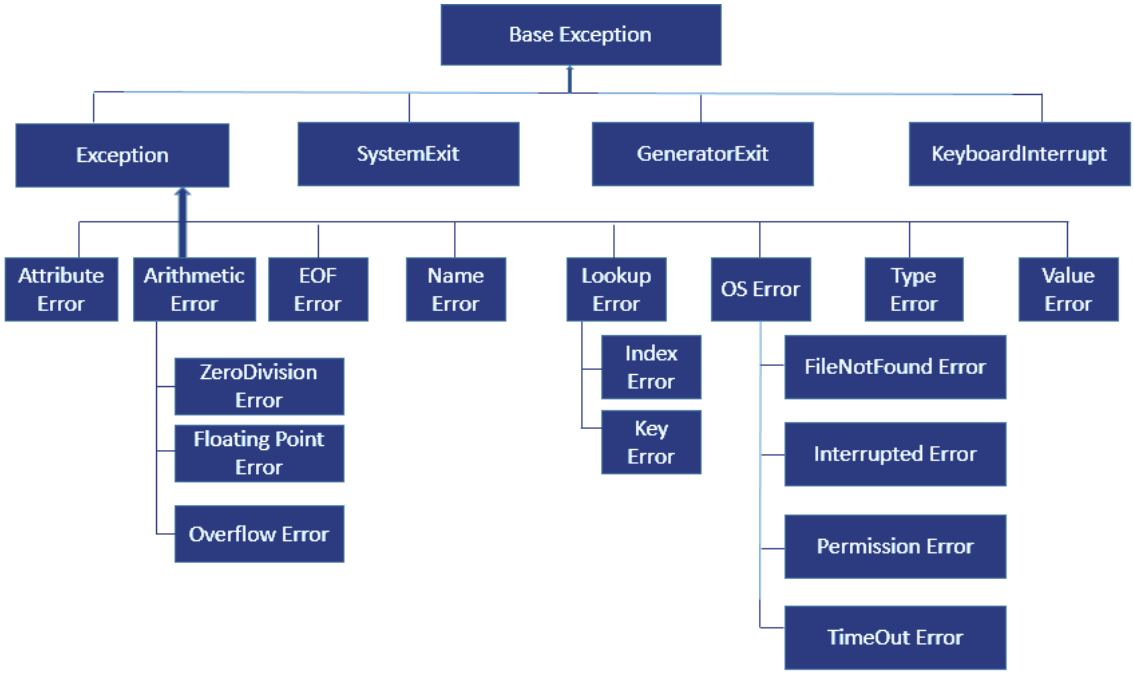
```
Traceback (most recent call last):
  File "D:\Python\Projects\stl\main.py", line 15, in <module>
    p.print_data("123")
    ~~~~~~~~~~~~^^^^^^^
  File "D:\Python\Projects\stl\main.py", line 3, in print_data
    self.send_data(data)
    ~~~~~~~~~~~~~~^^^^^^
  File "D:\Python\Projects\stl\main.py", line 8, in send_data
    raise Exception("принтер не отвечает")
Exception: принтер не отвечает
```

Отсюда хорошо видно, что изначально ошибка произошла в методе send_data и, так как не была обработана (перехвачена в блоках оператора try/except), то переместилась на уровень выше до метода print_data класса PrintData. Здесь она также не обрабатывалась, поэтому перешла на корневой уровень, где интерпретатор языка Python вывел ее в консоль с остановкой работы программы. Так происходит распространение любых исключений, сгенерированных оператором raise.

6.3.1. Пользовательские классы исключений

В приведенном примере класса PrintData исключение генерировалось с типом

класса Exception. Почему был выбран именно он? Если еще раз посмотреть на иерархию классов-исключений языка Python, то здесь во главе стоит базовый класс BaseException. Остальные классы наследуются от него и имеют строгую специализацию, кроме, разве что, класса Exception, который является общим для большого разнообразия типов ошибок, которые могут возникать в момент выполнения программы. Поэтому имеет смысл выбирать класс Exception для формирования новых собственных (пользовательских) классов исключений, что мы сейчас и сделаем.



Итак, чтобы создать в программе новый тип исключения достаточно объявить какой-либо класс с базовым классом Exception. В самом простом варианте это можно сделать, например, так:

```
class PrintSendDataError(Exception):
    """Класс исключения при отправке данных принтеру"""
```

И далее в программе использовать его при генерации соответствующих ошибок:

```
class PrintData:
    def print_data(self, data):
        self.send_data(data)
        print(f"печать: {data}")

    def send_data(self, data):
        if not self.send_to_print(data):
            raise PrintSendDataError("принтер не отвечает")

    def send_to_print(self, data):
        return False
```

Если теперь запустить эту программу, создав объект класса PrintData и вызвав метод print_data:

```
p = PrintData()
p.print_data("123")
```

то увидим следующее сообщение об ошибке:

```
PrintSendDataError: принтер не отвечает
```

То есть, в программе появился новый тип исключения, который можно обрабатывать с помощью оператора try/except, например:

```
p = PrintData()
try:
    p.print_data("123")
except PrintSendDataError as e:
    print(e)
```

Получается красивый программный код с собственными осмысленными типами ошибок. Благодаря такому подходу, становится легче понимать и отличать ошибки передачи данных принтеру от каких-либо других ошибок. Причем это же исключение можно поймать, указав его базовый класс Exception:

```
p = PrintData()
try:
    p.print_data("123")
except Exception as e:
    print(e)
```

Результат будет тем же самым. Но, конечно, лучше и понятнее, когда указывается конкретный тип отлавливаемого исключения. Поэтому в ряде случаев создают свою иерархию классов-исключений. Например, в нашем случае можно прописать общий класс для принтера PrintError:

```
class PrintError(Exception):
    """Общий класс исключения принтера"""
```

А затем остальные более узкие типы наследовать от него:

```
class PrintSendDataError(PrintError):
    """Класс исключения при отправке данных принтеру"""
```

В результате, в блоках except становится возможным отлавливать, как частные типы ошибок, так и общие, связанные с принтером. Например:

```
p = PrintData()

try:
    p.print_data("123")
except PrintSendDataError as e:
    print(e)
except PrintError:
    print("Ошибка печати")
```

Помимо этого можно расширить функционал каждого класса исключений. В

качестве примера добавим в класс ExceptionPrintSendData инициализатор с параметром сообщения об ошибке и два магических метода __str__ и __repr__:

```
class PrintSendDataError(PrintError):
    """Класс исключения при отправке данных принтеру"""
    def __init__(self, msg):
        self.message = msg

    def __str__(self):
        return self.__repr__()

    def __repr__(self):
        return f"Ошибка: {self.message}"
```

В результате выполнения этой программы будет сгенерировано исключение PrintSendDataError сообщением:

```
Ошибка: принтер не отвечает
```

Как видите, механизм обработки ошибок совместно с возможностью формировать свои собственные классы исключений, является удобным и гибким инструментом для обработки различных исключительных ситуаций при работе больших программ, где, как правило, отдельные модули лишь сигнализируют о возможных ошибках, а основная логика должна их обрабатывать. Такое разделение ролей заметно упрощает программный код и, как следствие, уменьшает ошибки при разработке крупных проектов.

6.4. Менеджеры контекста. Оператор with

Довольно часто при работе с внешними ресурсами, такими как, файлы, базы данных, сокеты и т.п. необходимо выполнять определенные завершающие действия, вне зависимости от возникновения различных программных ошибок. Например, ранее мы с вами уже видели, что любой открытый файловый поток желательно закрывать, после работы с ним. Если реализовывать эту логику через конструкцию try/except/finally, то получаем реализацию вида:

```
fp = None
try:
    fp = open("myfile.txt")
    for t in fp:
        print(t)
except Exception as e:
    print(e)
finally:
    if fp is not None:
        fp.close()
```

Здесь, благодаря блоку finally, гарантированно закрывается файл, даже если в блоке try возникли какие-либо исключения. И это типовой подход при работе с файлами. Поэтому было бы хорошо оптимизировать этот процесс, чтобы каждый раз специально не прописывать блок finally для закрытия файла. Именно для

этого и был разработан менеджер контекста, который задается с помощью оператора with. Например, та же самая программа с его использованием примет следующий вид:

```
try:
    with open("myfile.txt") as fp:
        for t in fp:
            print(t)
except Exception as e:
    print(e)
```

Здесь записано ключевое слово with, после которого указана функция open, открывающая файловый поток. Ключевое слово «as» позволяет сформировать переменную fp на открытый файл и в теле менеджера контекста работать с файлом через эту переменную.

Что же дает этот контекстный менеджер и почему здесь отсутствует блок finally? Дело в том, что при работе с файлами формируется менеджер контекста, создаваемый функцией open, т.е. файловый менеджер контекста. Его работа заключается в том, что когда он (менеджер) завершает свою работу, то автоматически закрывает файловый поток. Причем делает это вне зависимости от возникающих ошибок. В результате, отдельно прописывать блок finally больше нет необходимости.

Как же это все в деталях работает? Чтобы лучше понять принцип работы контекстных менеджеров, мы с вами создадим свой собственный. Делается это достаточно просто. Объявляется класс, в котором определяются два магических метода:

- __enter__() – срабатывает при запуске контекстного менеджера;
- __exit__() – срабатывает при завершении работы контекстного менеджера (вне зависимости, от возникших исключений).

Давайте для примера объявим класс менеджера контекста, который бы контролировал работу изменения списка, а именно, если программа в теле менеджера приводит к исключению (ошибке), то список должен оставаться прежним (без изменений). Использоваться он будет, например, следующим образом:

```
v1 = [1, 2, 3]
v2 = [1, 2]

with DefenderVector(v1) as dv:
    for i, _ in enumerate(dv):
        dv[i] += v2[i]

print(v1)
```

Здесь оператор with создает объект класса DefenderVector и автоматически вызывает метод __enter__. То, что вернет этот метод, на то и будет ссылаться

переменная dv. Далее в теле контекстного менеджера выполняются некоторые действия со списком. Если при этом возникают какие-либо ошибки, то обрабатываемый список v1 должен оставаться прежним (не меняться). Эту логику удобно прописать в магическом методе __exit__, который автоматически вызывается при завершении работы менеджера контекста. Причем всегда, вне зависимости от возникших ошибок. В результате можно записать следующую реализацию класса DefenderVector:

```
class DefenderVector:
    def __init__(self, v):
        self.__v = v

    def __enter__(self):
        self.__temp = self.__v[:]  # делаем копию вектора v
        return self.__temp

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is None:
            self.__v[:] = self.__temp
        return False
```

Здесь вначале, при вызове инициализатора, выполняется сохранение ссылки на переданный список (список v1) в приватном локальном свойстве __v. Затем, контекстный менеджер автоматически вызывает метод __enter__, который возвращает копию переданного списка, и дальнейшая работа внутри менеджера будет выполняться с копией, а не с исходным списком, через переменную dv. После того, как менеджер завершает свою работу, вызывается метод __exit__. Это может произойти либо при возникновении какой-либо ошибки, либо при штатном завершении работы контекстного менеджера. Если никаких ошибок не произошло, то параметр exc_type будет принимать значение None. Тогда измененный список __temp копируется в список __v. Иначе метод __exit__ возвращает значение False, что означает передачу возникшего исключения вышестоящему оператору. В данном случае ошибка будет передаваться в корень и отображаться интерпретатором языка Python в консоли.

После запуска программы, на экране появится сообщение об ошибке:

```
IndexError: list index out of range
```

Если же в методе __exit__ возвращать значение True, то ошибка будет обработана внутри менеджера контекста и на экране увидим значения списка v1:

```
[1, 2, 3]
```

В данном случае он не изменился, так как возникли ошибки. Если же немного поменять программу так, чтобы никаких исключений не возникало, например:

```
v1 = [1, 2, 3]
v2 = [1, 2, 5]

with DefenderVector(v1) as dv:
```

```
    for i, _ in enumerate(dv):
        dv[i] += v2[i]

print(v1)
```

то после запуска увидим измененный список v1:

```
[2, 4, 8]
```

Конечно, метод __exit__ обычно возвращает значение False при появлении каких-либо ошибок, а сам менеджер контекста помещают в блок try/except. В нашем случае его можно записать следующим образом:

```
v1 = [1, 2, 3]
v2 = [1, 2]

try:
    with DefenderVector(v1) as dv:
        for i, _ in enumerate(dv):
            dv[i] += v2[i]
except IndexError as e:
    print(e)

print(v1)
```

При выполнении этой программы будет обработано исключение IndexError и на экране отобразится исходный (не измененный) список v1.

При необходимости в одном операторе with можно создавать сразу несколько контекстных менеджеров, записывая их через запятую, например, так:

```
v1 = []

try:
    with DefenderVector(v1) as dv, open("my_file.txt") as fp:
        dv.append("Строки из файла:")

        for line in fp:
            dv.append(line)
except Exception as e:
    print(e)

print(v1)
```

В результате, список v1 будет содержать какие-либо элементы только при успешном чтении данных из файла. Если произойдет ошибка, то список v1 останется пустым благодаря использованию разработанного менеджера DefenderVector.

Другой аналогичный вариант совместного использования нескольких контекстных менеджеров – это вложить один в другой, например, следующим образом:

```
v1 = []

try:
    with DefenderVector(v1) as dv:
        with open("my_file.txt", encoding='utf-8') as fp:
            dv.append("Строки из файла:")

            for line in fp:
                dv.append(line)
except Exception as e:
    print(e)

print(v1)
```

Отличие такого подхода в том, что если какой-либо менеджер в этой вложенной цепочке будет обрабатывать (блокировать) возникающие исключения в своих вложенных блоках, то соответствующие ошибки не станут доходить до внешнего блока try. Это может стать причиной некорректной работы программы. Поэтому классы менеджеров следует проектировать так, чтобы они сами не перехватывали исключения. Допускается только генерация своих собственных, при внутренней обработке данных.

ГЛАВА VII. МЕТАКЛАССЫ

Программы на языке Python оперируют объектами. Например, булевы значения, строки, числа, списки, словари и т. п. – все это объекты. И эти объекты образованы от соответствующих классов (типов данных): bool, str, int, float, list, dict и т.д. Но эти классы так же являются объектами, потому что все в Python – объекты, даже классы. Да, классы – это объекты, которые позволяют создавать другие объекты с конкретным содержимым. А раз так, то должно быть «нечто», что создает объекты-классы! И это «нечто» в Python называется **метаклассом**. Причем метакласс – это тоже объект (в Python всё объекты), но это объект особого рода, который нельзя динамически порождать каким-нибудь другим мета-метаклассом. Он является вершиной, отправной точкой для создания обычных классов и, как следствие, их объектов.

Что является метаклассом в языке Python? Наверное, вас немного удивит, но это давно знакомый нам объект type, который мы использовали для определения типов объектов. Когда он вызывается с одним аргументом, например:

```
a = "hello"
b = 5

type(a)  # <class 'str'>
type(b)  # <class 'int'>
```

то возвращает тип переданного объекта. Если же ему передаются три аргумента:

```
type(<имя класса>, <кортеж родительских классов>, <словарь с атрибутами и
их значениями>)
```

то type начинает работать совершенно иным образом, а именно, создает (динамически) новый класс, новый тип данных в программе.

В действительности, все стандартные типы данных, функции и обычные классы – порождены метаклассом type. Это видно при выводе типов, например:

```
type(int)    # <class 'type'>
type(str)    # <class 'type'>
```

Всюду видим обозначение `'type'`, которое указывает, что все эти объекты сформированы метаклассом type.

Возможно, на данном этапе вы задаетесь извечным вопросом, зачем все это надо? Мы же и так можем объявлять классы? Для чего их создавать динамически в процессе работы программы? На этот вопрос хорошо ответил гуру языка Python Тим Питерс. Он сказал:

Метаклассы – это глубокая магия, о которой 99% пользователей даже не нужно задумываться. Если вы думаете, нужно ли вам их использовать — вам не нужно (люди, которым они реально нужны, точно знают, зачем они им, и не нуждаются в объяснениях, почему).

Однако чтобы лучше понимать, как функционирует язык Python, все же лучше ознакомиться с этой темой. И, кроме того, возможно в будущем именно вам это пригодится.

7.1. Введение в метаклассы. Объект type

Давайте первым делом попробуем создать свой класс с помощью метакласса type. Для простоты – это будет знакомый нам класс Point для представления точек на плоскости, который в явном виде выглядит так:

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0
```

А создание его через объект type будет иметь следующий вид:

```
A = type('Point', (), {'MAX_COORD': 100, 'MIN_COORD': 0})
```

Здесь переменная A – это ссылка на новый созданный класс с именем Point. Так как классы – это тоже объекты, то мы можем на них ссылаться через разные переменные. И, далее, можно создать экземпляр этого класса, используя переменную A, например:

```
pt = A()
```

При выводе этого объекта на экран:

```
print(pt)
```

увидим строчку:

```
<__main__.Point object at 0x000001D2430A6F90>
```

То есть, класс Point действительно был успешно создан с помощью type, а затем сформирован объект этого класса pt.

При необходимости, можно дополнительно прописывать базовые классы, передавая их список вторым аргументом метаклассу type. Например, следующим образом:

```
class B1: pass
class B2: pass

A = type('Point', (B1, B2), {'MAX_COORD': 100, 'MIN_COORD': 0})
pt = A()
```

Если теперь вывести коллекцию __mro__ класса Point, используя ссылку A:

```
print(A.__mro__)
```

то увидим всю цепочку наследования:

```
(<class '__main__.Point'>, <class '__main__.B1'>, <class '__main__.B2'>,
```

```
<class 'object'>)
```

Также, при динамическом формировании новых классов, в них можно добавлять и методы. Делается это, по крайней мере, двумя способами. В первом случае используется уже готовая функция с указанием ссылки на нее, например:

```
def method1(self):
    print(self.__dict__)


A = type('Point', (), {'method1': method1})
```

После создания экземпляра класса Point, через него можно вызвать этот метод:

```
pt = A()
pt.method1()   # {}
```

Во втором случае, метод определяется непосредственно в словаре через лямбда-функцию:

```
A = type('Point', (), {'MAX_COORD': 100,
                       'method1': lambda self: self.MAX_COORD})

pt = A()
print(pt.method1())   # 100
```

Однако таким способом обычно определяются только простейшие методы. Функции со сложным поведением лучше прописывать отдельно с последующим указанием ссылок на них.

Вот, что из себя представляет метакласс type, и как с его помощью можно динамически создавать новые классы. Однако описывать сложный алгоритм формирования новых классов непосредственно через объект type не очень удобно. Поэтому обычные классы проще объявлять в программе привычным нам образом.

7.2. Пользовательские метаклассы. Параметр metaclass

В Python можно описывать свои собственные метаклассы, которые, конечно же, явно или неявно наследуются от основного метакласса type. Давайте предположим, что мы хотели бы создавать классы для точек, в которых автоматически появлялись бы атрибуты MAX_COORD и MIN_COORD. Для этого определим метакласс в виде функции (да, обычные функции тоже можно использовать как метаклассы), следующим образом:

```
def create_class_point(name, base, attrs):
    attrs.update({'MAX_COORD': 100, 'MIN_COORD': 0})
    return type(name, base, attrs)
```

Обратите внимание, функция-метакласс должна иметь три параметра: name – имя создаваемого класса; base – кортеж из базовых классов; attrs – словарь с атрибутами класса.

Функция create_class_point автоматически добавляет в коллекцию attrs два атрибута MAX_COORD и MIN_COORD, а затем, с помощью метакласса type формирует новый класс.

Теперь, эту функции можно использовать в качестве метакласса, при объявлении класса. Для этого прописывается специальный параметр metaclass с указанием ссылки на функцию-метакласс, например:

```
class Point(metaclass=create_class_point):
    def get_coords(self):
        return 0, 0
```

Что в итоге у нас получилось? Если сейчас создать объект класса Point, то в нем появятся оба атрибута MAX_COORD и MIN_COORD, а также метод get_coords:

```
pt = Point()
print(Point.__dict__)
print(pt.get_coords())
```

На экране увидим строчки:

```
{'__module__': '__main__', '__firstlineno__': 6, 'get_coords': <function
Point.get_coords at 0x0000020F935FC720>, '__static_attributes__': (),
'MAX_COORD': 100, 'MIN_COORD': 0, '__dict__': <attribute '__dict__' of
'Point' objects>, '__weakref__': <attribute '__weakref__' of 'Point'
objects>, '__doc__': None}
(0, 0)
```

Фактически, класс Point был создан функцией create_class_point, в которую сам Python передал нужный набор аргументов с соответствующими значениями, а далее, с помощью вызова метакласса type был создан класс Point. Такой подход намного удобнее прописывания всех атрибутов непосредственно в словаре attrs.

Однако использование функции в качестве метакласса – это, скорее, учебный пример для лучшего понимания сущности метаклассов. На практике чаще используют отдельные классы, которые играют роль метаклассов.

Давайте повторим нашу программу, но с использованием пользовательского метакласса. Вначале объявим класс с именем Meta (имя придумываем сами), который должен наследоваться от метакласса type:

```
class Meta(type):
    def __init__(cls, name, base, attrs):
        super().__init__(name, base, attrs)
        cls.MAX_COORD = 100
        cls.MIN_COORD = 0
```

Внутри этого класса прописан инициализатор с четырьмя параметрами. Первый cls – это ссылка на новый уже созданный класс, а три остальных нам уже знакомы. Внутри инициализатора обязательно нужно вызвать инициализатор базового метакласса type, а затем, динамически добавляются два атрибута MAX_COORD и MIN_COORD в новый созданный класс. Теперь, в классе Point

вместо ссылки на функцию указывается ссылка на метакласс:

```
class Point(metaclass=Meta):
    def get_coords(self):
        return 0, 0


pt = Point()
print(Point.__dict__)
print(pt.get_coords())
```

После выполнения программы увидим прежний результат:

```
{'__module__': '__main__', '__firstlineno__': 8, 'get_coords': <function
Point.get_coords at 0x000001F8104BC720>, '__static_attributes__': (),
'__dict__': <attribute '__dict__' of 'Point' objects>, '__weakref__':
<attribute '__weakref__' of 'Point' objects>, '__doc__': None, 'MAX_COORD':
100, 'MIN_COORD': 0}
(0, 0)
```

Однако инициализатор __init__ в метаклассе Meta вызывается, когда класс Point полностью создан. Для более тонкой работы лучше переопределить магический метод __new__, который срабатывает непосредственно перед созданием класса. Аналогичная реализация может быть записана в следующем виде:

```
class Meta(type):
    def __new__(cls, name_class, base, attrs):
        attrs.update({'MAX_COORD': 100, 'MIN_COORD': 0})
        return type.__new__(cls, name_class, base, attrs)
```

Здесь магический метод __new__ имеет аналогичные четыре параметра: cls – ссылка на текущий класс Meta (класса Point на данный момент еще нет); name_class – имя создаваемого класса; base – кортеж из базовых классов; attrs – словарь атрибутов создаваемого класса.

Так как метод __new__ вызывается до создания нового класса, то добавляемые атрибуты MAX_COORD и MIN_COORD прописываются непосредственно в словаре attrs. После этого вызывается аналогичный метод __new__ метакласса type. Обратите внимание, что метод __new__ метакласса Meta должен вернуть ссылку на созданный класс, поэтому здесь обязательно прописывается оператор return.

Возможно, у вас возникает вопрос, в чем преимущество использования классов в качестве метаклассов перед функциями? Принципиальных отличий нет. Любой алгоритм можно прописать и на уровне функции и на уровне класса. Однако при использовании классов, у нас в руках оказывается вся мощь ООП. В частности, можно создавать иерархии для формирования сложного поведения метаклассов. В крупных проектах это заметно облегчает код и делает его гораздо понятнее. Поэтому на практике, когда возникает необходимость, чаще используют метаклассы из-за их удобства.

7.3. Учебный пример разработки метакласса

Чтобы тема метаклассов была более понятна рассмотрим пример, когда при создании объектов класса, будут автоматически добавляться в него локальные свойства с теми же именами, что и атрибуты самого класса.

Вначале определим сам метакласс Meta со следующей реализацией:

```
class Meta(type):
    def create_local_attrs(self, *args, **kwargs):
        for key, value in self.class_attrs.items():
            setattr(self, key, value)

    def __init__(cls, name_class, base, attrs):
        cls.class_attrs = attrs
        cls.__init__ = Meta.create_local_attrs
```

Здесь метод create_local_attrs будет играть роль инициализатора в создаваемом классе. В методе перебираются атрибуты класса из коллекции class_attrs и создаются в его объекте аналогичные локальные свойства с соответствующими значениями. Чтобы словарь class_attrs существовал, он формируется в инициализаторе метакласса Meta со всеми атрибутами создаваемого класса и их значениями. Фактически, метакласс Meta конструирует классы следующего вида:

```
class Women:
    class_attrs = {'title': 'заголовок', 'content': 'контент', 'photo':
'путь к фото'}
    title = 'заголовок'
    content = 'контент'
    photo = 'путь к фото'

    def __init__(self, *args, **kwargs):
        for key, value in self.class_attrs.items():
            self.__dict__[key] = value
```

Но записываются они в программе гораздо проще, например, так:

```
class Women(metaclass=Meta):
    title = 'заголовок'
    content = 'контент'
    photo = 'путь к фото'
```

Если теперь создать объект этого класса и вывести его локальные атрибуты на экран:

```
w = Women()
print(w.__dict__)
```

то увидим строку:

```
{'__module__': '__main__', '__qualname__': 'Women', '__firstlineno__': 11,
'title': 'заголовок', 'content': 'контент', 'photo': 'путь к фото',
```

```
'__static_attributes__': ()}
```

Действительно, объект w в своих локальных атрибутах повторяет все атрибуты класса.

Конечно, метаклассы, как правило, содержат гораздо более изощренную логику работы, чем в приведенном примере. Мы здесь лишь рассмотрели саму идею их использования. Однако применять их нужно только тогда, когда другие инструменты дают слишком сильное усложнение программного кода. Здесь всегда следует держать в голове высказывание гуру языка Python Тима Питерса:

Метаклассы – это глубокая магия, о которой 99% пользователей даже не нужно задумываться. Если вы думаете, нужно ли вам их использовать — вам не нужно (люди, которым они реально нужны, точно знают, зачем они им, и не нуждаются в объяснениях, почему).

В действительности, метаклассы реализуют очень простую идею формирования новых классов по некоторому алгоритму. Однако чаще всего стандартной логики объявления классов вполне достаточно для реализации самых разных задач. Существуют лишь отдельные исключительные ситуации, когда действительно возникает целесообразность использования метаклассов. И если вы на практике столкнетесь с ними, то теперь знаете, как в целом они работают.

Курсы серии «Добрый, добрый» на платформе Stepik
(для отработки практических навыков)

1. Python: https://stepik.org/course/100707
2. Python ООП: https://stepik.org/a/116336
3. Стандартная библиотека Python: https://stepik.org/a/259466
4. Django + Docker: https://stepik.org/a/183363
5. C/C++: https://stepik.org/a/193691
6. ООП C++: https://stepik.org/a/205781
7. Машинное обучение: https://stepik.org/course/209247/
8. Нейронные сети на PyTorch: https://stepik.org/a/227582

Телеграм-канал: https://t.me/selfedu_sb

Телеграм-чат: https://t.me/python_selfedu

Видео-каналы: Rutube | Youtube

Быть или не быть?
Вот в чем вопрос.
Достойно ли идти в IT?
Иль остановиться?
Что лучше выбрать?
Жизнь? Биенье сердца? Пенье птиц?
Ручей, журчащий музыкой? Слепец!
Иное суждено тебе.
Сидеть. И день за днем
Сухим, логическим пером
Писать команды, чтоб потом
Вообразить себя творцом!
Что лучше выбрать?
Знаешь только ты...