МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ
Кафедра системного анализа и информационных технологий

Еникеев Разиль Радикович

ОСНОВЫ И БАЗОВЫЕ ТИПЫ ДАННЫХ В PYTHON

Учебное пособие

КАЗАНЬ, 2021

УДК 004.432.2
ББК 3973.22

Рецензенты:
кандидат физико-математических наук, старший преподаватель кафедры математической статистики **А. Д. Романенко**
младший научный сотрудник научно-исследовательской лаборатории «Квантовые методы обработки данных» **И. М. Маннапов**

Еникеев Р. Р.
Основы и базовые типы данных в Python / Р. Р. Еникеев — Казань: КФУ, 2021. — 129 с.

Учебное пособие предназначено для самостоятельной работы по проблемным темам курса «Программирование на Python» для студентов, обучающихся в бакалавриате по направлениям «Фундаментальная информатика и информационные технологии», «Информационная безопасность». В учебном пособии изложены основы работы с языком программирования Python. Подробно изучаются основные операторы, динамическая типизация, базовые типы данных: числа, списки, строки, кортежи, словари и файлы. Учебное пособие содержит большое количество теоретического материала с достаточным количеством примеров и поясняющих рисунков.

Содержание

1. Основы Python

1.1. История Python

Язык программирования Python был создан Гвидо ван Россумом (Guido van Rossum), который назвал язык в честь английской комик-группы Монти Пайтон (Monty Python), поэтому название языка программирования произносится как «П*ай*тон».

Главными задачами при разработке Python были повышение производительности труда программиста и повышение читаемости кода. Хорошая читаемость кода помогает в сопровождении программ в течении всего времени эксплуатации программы.

На текущий момент существуют две главные версии Python: 2.7 и 3.9. Часто используются обозначения Python 2 и Python 3 (или Python 3.X), которые указывают на всё семейство версий. Версия Python 2.7 является более старой версией нежели Python 3. И хотя Python 3 появился очень давно, многие программисты не переходят на новую версию, используя старую версию Python 2.7. Таким образом, для создания новых приложений лучшим решением является Python 3, а Python 2.7 следует оставить на случай сопровождения существующей кодовой базы. В текущем пособии будет рассматриваться Python 3.9.

1.2. О Python

Язык Python имеет большу́ю область применения.

- Python является одним из ведущих языков программирования для исследований в таких научных областях, как машинное обучение, компьютерное зрение, компьютерная алгебра, анализ и визуализация данных и так далее.

- В Python присутствует широкий выбор встроенных библиотек, которые позволяют, в частности, решать следующие задачи. Системное программирование — написание программ с помощью предоставляемых операционной системой функций для работы с ней, скрывая детали работы с самой операционной системой. Сетевые и клиент-серверные приложения достаточно просто создаются в Python.

- Для Python разработано большое количество прикладных библиотек, которые позволяют, в частности, разрабатывать веб-приложения (библиотеки flask и django), создавать настольные (библиотеки pyqt и tkinter) и мобильные приложения (библиотека Kivy).

Python реализует следующие парадигмы программирования.

- Императивное программирование. Программа на языке, который поддерживает эту парадигму, представляет собой алгоритм в виде набора последовательных шагов для исполнения.

- Структурное программирование. Языки с поддержкой структурного программирования состоят из ветвлений, циклов, функций и линейных областей кода без переходов.

- Объектно-ориентированное программирование (ООП). Возможности ООП в Python позволяют создавать классы с поддержкой инкапсуляции, полиморфизма и наследования.

- Функциональное программирование. В этой парадигме функции можно передавать в качестве аргумента, возвращать в качестве результата и выполнять другие операции над функциями. Эта парадигма позволяет упростить написание многих фрагментов кода.

Далее перечислены основные черты Python.

- Интерпретируемость. Программа на Python не компилируется, а выполняется строчка за строчкой специальной программой, называемой *интерпретатором*. Интерпретатор обрабатывает строку кода и транслирует её в одну или несколько машинных команд.

- Переносимость программ. Благодаря предыдущему пункту программы на Pyhton можно переносить на все платформы и операционные системы, для которых реализован интерпретатор Python.

- Динамическая типизация. Нет необходимости объявлять тип переменной, интерпретатор самостоятельно выведет тип объекта во время выполнения.

- Автоматическое управление памятью. В Python не требуется освобождать выделенную память, *сборщик мусора*, являющийся отдельным компонентом интерпретатора, делает это автоматически.

Среди преимуществ языка можно выделить простоту написания и чтения кода, большое количество разработанных библиотек.

Вследствие интерпретируемости Python главным его недостатком является невысокая производительность: программы на Python выполняются медленнее программ, написанных на C++. Но часто высокая скорость разработки программ на Python является более важной, чем высокая производительность программ.

1.3. Hello world

Начнём рассмотрение возможностей языка Python с вывода на экран строки `'hello world'`. Именно эту задачу часто решают первой при изучении нового языка программирования. Такая простая задача позволяет проверить правильность установки и настройки интерпретатора Python, его библиотек и операционной системы. Кроме этого, такая простая программа даёт базовое понимание того, каким образом создаются и выполняются программы на языке программирования. Решение задачи представлено в следующем коде, который нужно разместить в файле с расширением `.py` и запустить с помощью интерпретатора (подробнее см. [10]):

```
print('hello world')
```

Одна простая строчка кода позволяет вывести строку `'hello world'` в консоль. Python является интерпретируемым, что означает, что интерпретатор начинает выполнять файл с программой с первой строчки шаг за шагом, не требуя определения функции `main`, которая в компилируемых языка программирования является точкой входа всей программы.

Теперь рассмотрим сам пример. Функция `print` после вызова с применением круглых скобок выводит переданный аргумент на экран, при необходимости преобразуя параметр в строку. В качестве аргумента передаётся строка, которая может быть задана с помощью одинарных кавычек.

Предыдущий пример, ко всему прочему, показал одну из особенностей интерпретируемых языков программирования: отсутствие `main`. Одну из возможностей динамической типизации поможет раскрыть следующая задача. Необходимо вывести на экран строку, хранящуюся в переменной. Решение представлено ниже:

```
s = 'hello world'
print(s)
```

В этом примере строка `'hello world'` сохраняется в переменную `s`. В отличие от языка C++ в Python не нужно указывать тип данных переменной, интерпретатор самостоятельно выведет тип данных.

Python по отношению к названиям переменных чувствителен к регистру. Хотя названия переменных и других элементов языка (функции и классы) можно указывать на русском (или другом иностранном языке) и использовать довольно необычные символы, предпочтительнее всего в названии переменных указывать только буквы английского алфавита, цифры, знак нижнего подчеркивания, при этом название переменной не может начинаться с цифры. Не следует использовать имена встроенных функций и классов в качестве имени своих переменных, так как происходит перекрытие имён. Кроме этого, нельзя использовать ключевые слова, например, `if`. Также названия файлов с исходным кодом должны подчиняться описанному выше шаблону, например, `a.py`.

Одним из способов повышения читаемости кода является использование комментариев. *Комментарии* — часть кода, которая не выполняется и пропускается интерпретатором. Комментарии пишутся разработчиками для разработчиков, чтобы разъяснить код или описать замечания к коду. Для написания комментариев в Python используется символ `#`. Комментарии действуют от начала символа `#` до конца строки:

```
print('hello world') # вывод: hello world
```

В данном пособии в комментариях будет указываться результат, получаемый после выполнения соответствующей строчки кода.

Далее попробуем вывести на экран введённую пользователем строку:

```
s = input('input your text:\n')
print(s)
```

Функция `input` сначала выводит свой аргумент на экран, а затем ждёт ввода одной строки (line) от пользователя. Введённая строка затем возвращается функцией `input`, присваивая это значение в переменную `s`. В этом примере интерпретатор также выводит тип данных переменной `s`.

Рассмотрим задачу нахождения суммы целых чисел `a` и `b`, которые вводятся пользователем с клавиатуры:

```
s1 = input('input a:\n')
s2 = input('input b:\n')
a = int(s1)
b = int(s2)
my_sum = a + b
print(my_sum)
```

Функция `input` возвращает считанную строку, поэтому перед сложением эту строку нужно преобразовать в целое число путём передачи её в `int`. Функция `input` считывает строку (line) до символа перевода строки. Для перевода этой строки в другой тип данных, необходимо её передать в конструктор класса, как это было сделано в этом примере. Перепишем это решение, используя возможность поместить выражение (результат вызова функции и применение операций) внутрь вызова функции:

```
a = int(input('input a:\n'))
b = int(input('input b:\n'))
print(a + b)
```

В этом коде сначала вычисляются внутренние выражения: вызов функции `input` и использование операции `+`; а затем результат их выполнения передаётся в качестве аргумента во внешнюю функцию.

1.4. Булевский тип и `None`

Класс `bool` служит для работы с булевскими константами `True` и `False`, представляющими «истину» и «ложь» соответственно.

Константа `None` является аналогом `null` в C++ и означает отсутствие значения. Для проверки, является ли объект `None` или нет, *необходимо применять* операторы `is` и `is not`:

```
a is None
b is not None
```

1.5. Основы чисел

В Python основными числовыми типами данных являются целые числа (класс `int`) и числа с плавающей запятой (класс `float`). Используя эти классы, можно получить числа из строк: `int(s)` или `float(s)`. Числа задаются по аналогии с другими языками программирования:

```
a = 1
b = 2.3
```

Функция `print` выводит числа на экран. Числа поддерживают операции сравнения (`==`, `!=`, `<`, ...) и стандартные математические операции (`+`, `-`, `*`, `/`, `%`). Однако в Python оператор деления `/` вычисляет результат как число с плавающей точкой даже при делении чисел нацело, например, `6/2` вернёт `3.0`. Для целочисленного деления присутствует уникальный для Python оператор `//`, например, `6//2` вернёт `3`.

1.6. Основы строк

Строки используются для хранения и обработки текстовой информации и реализованы классом `str`. Строки представляют собой неизменяемую последовательность символов, в котором каждый символ также является строкой, то есть в Python отсутствует тип `char`.

Строка задаётся в одинарных либо двойных кавычках. Эти способы полностью эквиваленты:

```
s = 'abc' # или s = "abc"
```

Внутри кавычек можно указать экранированные символы `'\n'`, `'\t'`, `'\r'`, ...

Вывести на печать строку можно при помощи функции `print`, которая выводит содержимое строки без кавычек:

```
print(s) # > abc
```

Функция `len` позволяет выяснить длину строки: `len('abc')` вернёт `3`. Длина строки хранится в виде переменной экземпляра, поэтому получение длины строки выполняется очень быстро.

Символы в строке пронумерованы с нуля. Для получения символа по его индексу следует воспользоваться индексацией, указав в квадратных скобках индекс:

```
print(s[0]) # > a
```

Конкатенация при помощи операции + позволяет соединить две строки:

```
a = 'Hello, ' + 'world!' # a → 'Hello, world!'
```

Оператор == сравнивает посимвольно две строки на равенство:

```
'hello' == 'hello' # → True
s2 = 'a'
s2 == s2[0] # → True; нулевой элемент возвращает строку
```

Для приведения строки к нижнему регистру применяется метод строки `lower`, который может применяться для проверки строк вне зависимости от регистра их символов:

```
'heLLo'.lower() == 'Hello'.lower() # →True
```

Обход строки выполняется оператором `for` при помощи цикла по всем индексам строки:

```
hello = 'hello'
for i in range(0, len(s), 1):
    print(s, end=', ')
# > h, e, l, l, o,
```

Функция `range` возвращает по очереди числа от `0` до `len(s)`, *не включая* его, с шагом `1`. Оператор `for` присваивает эти значения в переменную `i` и выполняет тело (вложенный блок кода).

Двоеточие в конце строки с оператором `for` является требованием синтаксиса Python. Следующая строка после `for` является телом цикла и *обязана иметь отступ* (табуляцию или пробелы) согласно синтаксису языка. Тело цикла `for` завершается, когда встречается блок кода, имеющий одинаковый отступ с заголовком (в примере выше это `print`). Отсутствие двоеточий и отступов является распространённой ошибкой начинающих изучать Python программистов.

Для преобразования объекта в строку нужно передать объект в `str`: `str(1)` вернёт `'1'`. Этот способ используется для получения строкового представления объекта для пользователя (например, для вывода на экран). Для получения строкового представления для программиста используется функция `repr`:

```
print(repr('abc')) # вывод: 'abc' - вывод с кавычками
print('')          # вывод: в консоль ничего не выводится
print(repr(''))    # вывод: ''
```

1.7. Функция `range`

В этом разделе подробно рассмотрим функцию `range`, имеющую несколько вариантов вызова.

Функция `range(start, stop, step)` возвращает последовательно числа в диапазоне `[start; stop)` с шагом `step`, то есть числа `start`, `start+step`, `start+2*step`, ... , `start+n*step`, где для `n` выполняется условие `start+n*step` < `stop` ≤ `start+(n+1)*step`. Например, следующий код печатает каждый символ с чётным индексом:

```
s = 'hello'
for i in range(0, len(s), 2):
    print(s[i], end='')
# вывод: hlo
```

При задании отрицательного шага числа будут возвращаться уже в обратном порядке, при этом нужно помнить, что первый параметр указывает начало диапазона включительно, а второй — конец диапазона, не включая его. Далее представлен пример обхода строки в обратном порядке:

```
for i in range(len(s)-1, -1, -1):
    print(s[i], end='')
# вывод: olleh
```

Шаг можно не указывать, тогда значение шага будет установлено в 1. Таким образом, `range(start, stop)` возвращает последовательно числа в диапазоне `[start; stop)` с шагом 1.

Кроме того, функцию `range(stop)` можно вызвать с одним аргументом, обозначающим открытую верхнюю границу диапазона. В этом случае будут возвращаться последовательно числа в диапазоне `[0; stop)` с шагом 1. Таким образом, обход всех элементов строки по очереди можно выполнить с помощью следующего кода:

```
for i in range(len(s)):
    print(s[i])
```

Функция `range` возвращает объект диапазана, у которого можно получить значения атрибутов `start`, `stop` и `step`:

```
r = range(6)
print(r.start, r.stop, r.step) # вывод: 0 6 1
```

При этом только в контексте цикла `for` объект диапазона начинает возвращать последовательность целых чисел.

1.8. Основы списков и кортежей

В Python отсутствуют массивы, вместо них используются списки. Списки хранят последовательность элементов. Для создания списка можно воспользоваться следующим способом:

```
L = [6, -2, 11]
```

В этом примере задаётся список, состоящий из чисел 6, -2 и 11.

Вывести список на экран можно с помощью функции `print`:

```
print(L) # вывод: [6, -2, 11]
```

Размер списка можно получить, применив встроенную функцию `len`: `len(L)` вернёт 3.

Элементы в списке пронумерованы, начиная с 0. Получение или изменение элемента списка осуществляется путём индексации:

```
a = L[1]            # a равно -2
L[0] = 7            # L равно [7, -2, 11]
L[2] = L[0] + L[1] # L равно [7, -2, 5]
```

При индексации списка следует избегать выхода за границы списка с помощью предварительной проверки условия `0 <= i < len(L)`.

Для обхода элементов списка можно воспользоваться оператором `for`, указав верхнюю границу итерации как `len(L)` не включительно:

```
s = 0
for i in range(len(L)):
    s += L[i]
print(s) # вывод: 10
```

Для создания списка во время выполнения можно воспользоваться двумя стратегиями. Первая — создать состоящий из 0 список размера n, применив операцию *:

```
L = [0]*5 # L равняется [0, 0, 0, 0, 0]
```

Второй способ — создать пустой список, добавляя в цикле требуемые значения. Для этого можно воспользоваться методом append, добавляющим элемент в конец списка:

```
L = []
for i in range(5):
    L.append(0)
# L равно [0, 0, 0, 0, 0]
```

Далее продемонстрировано считывание списка целых чисел из консоли с помощью двух способов, описанных выше (необходимо ввести размер списка в самом начале):

```
n = int(input('Введите размер списка:\n'))
L = [0] * n
for i in range(n):
    L[i] = int(input('Введите элемент списка:\n'))
# или
L = []
for i in range(n):
    a = int(input('Введите элемент списка:\n'))
    L.append(a)
```

При запуске этого кода каждое число должно вводиться с новой строки, иначе произойдёт ошибка.

В дополнение к спискам в Python присутствует *кортеж*. Кортеж является аналогом списка, который нельзя изменять. Кортеж задаётся в круглых скобках:

```
a = (1, 2, 3)
a[0]      # вернёт 1
a[1] = 2 # ошибка: изменение кортежа
```

1.9. Оператор if/elif/else

Python поддерживает требуемые в структурном программировании ветвление и циклы благодаря операторам. Поддержка ветвлений обеспечивается операторами if/elif/else. Оператор if выполняет проверку условия и, в случае истинности условия, выполняет тело:

```
n = int(input('input some integer:\n'))
if n > 0:
    print('n больше нуля')
```

Ещё раз стоит отметить, что тело составных операторов (в частности, if и for) обязано иметь отступы, иначе это считается синтаксической ошибкой. Кроме этого, заголовок составных операторов в конце должен иметь двоеточие.

Блок else выполняет своё тело, когда условие в заголовке if не было выполнено:

```
n = int(input('input some integer:\n'))
if n % 2 == 0:
    print('n - чётное')
else:
    print('n - нечётное')
```

В этом примере тело `else` выполняется, когда условие `n % 2 == 0` неверно, то есть при нечётном `n`. Операторов `if` и `else` достаточно для реализации ветвлений в программе, но для удобства написания программ имеется оператор `elif`, который помещается между блоками `if` и `else`. Интерпретатор по очереди проверяет условие каждого оператора, если условие истинно, тогда исполняется тело оператора, иначе осуществляется переход к следующему оператору. Это продолжается, пока не будет встречен `else` или не закончится блок операторов. Рассмотрим задачу вывода знака числа на экран. Далее представлены два способа решения этой задачи: слева — решение с помощью `elif`, справа — без `elif`:

```
a = int(input('Input a:'))
if a > 0:
    print('a > 0')
elif a == 0:
    print('a = 0')
else:
    print('a < 0')
```

```
a = int(input('Input a:'))
if a > 0:
    print('a > 0')
else:
    if a == 0:
        print('a = 0')
    else:
        print('a < 0')
```

Количество блоков `elif` после `if` может быть произвольным.

В предыдущих разделах уже был рассмотрен цикл `for`, позволяющий с помощью `range` обходить диапазоны целых чисел. Кроме этого существует цикл с предусловием `while`, но отсутствует цикл с постусловием. Для управления потоком выполнения цикла применяются операторы `break` и `continue`, которые позволяют выйти из цикла и перейти на следующую итерацию соответственно. Далее представлен пример `while`:

```
while True:
    n = int(input('Угадайте число от 1 до 10'))
    if n == 7:
        print('правильно')
        break
    else:
        print('попробуйте ещё раз')
```

1.10. Основы словарей

Списки представляют собой последовательность объектов, доступ к которым осуществляется по позициям. *Словарь* (*dictionary*) позволяет хранить значения по ключам. Например, рассмотрим задачу хранения набора товаров и цен на них. С помощью списков эту задачу можно решить с применением двух списков для отдельного хранения товаров и цен или с помощью одного списка, элементами которого являются кортежи пар товар/цена, что не очень удобно. Словари предназначены для решения подобных задач:

```
prices = {'banana': 50, 'apple': 100}
```

В этом примере создаётся словарь, где у товаров `'banana'` и `'apple'` имеется цена `50` и `100` соответственно: `'apple'` является *ключом* (*key*), `100` — *значением* (*value*). Ключи

в списке являются уникальными. В нашем примере товар может появляться только один раз, то есть у товара может быть только одна цена. Однако значения могут повторяться, что в нашем примере означает, что одинаковая цена может быть у нескольких товаров. Ключами могут быть только неизменяемые объекты, а значениями — любые объекты.

Одной из основных операций над словарями является операция получения значения по ключу, выполняемая аналогично индексации в списках с помощью указания ключа в квадратных скобках. Словари являются изменяемыми. Это означает, что можно изменять значения и добавлять новые элементы:

```
prices = {'banana': 50, 'apple': 100}
# получение значения по ключу
a = prices['apple']     # a → 100
prices['banana'] = 70   # изменение элемента
prices['orange'] = 120 # добавление нового элемента
# prices → {'banana': 70 , 'apple': 100, 'orange': 120}
```

Вывод словаря осуществляется с помощью функции `print`, которая выводит всё содержимое словаря:

```
prices = {'banana': 50, 'apple': 100}
print(prices) # > {'banana': 50, 'apple': 100}
```

Длину словаря (количество элементов) можно извлечь путём вызова функции `len`:

```
print(len(prices)) # > 2
```

1.11. Основы обработки исключений

Обработка ошибок в Python производится с помощью исключений: возбуждения и обработки исключений. Код, который может выбросить исключение, необходимо поместить внутрь оператора `try`, а обработку исключений внутрь `except`:

```
try:
    1/0
except:
    print('error') # вывод: error
```

В этом примере при делении на ноль возбуждается исключение, которое перехватывается `except`. После перехвата начинает выполняться тело `except`. После блока `except` код продолжает выполнение со следующего после `try` оператора. Если после `except` ничего не указано, тогда этот блок обрабатывает все выброшенные исключения. Чтобы сузить действие оператора `except` можно указать исключение, которое будет обрабатываться этим блоком:

```
try:
    1/0
except ZeroDivisionError:
    print('error') # вывод: error
```

В этом примере блок `except` перехватывает только ошибки деления на ноль, игнорируя остальные.

После `try` можно указать произвольное количество блоков `except`, которые обрабатывает разные исключения.

Кроме блока `except` можно указать блок `finally`, который должен быть указан самым последним. Тело оператора `finally` выполняется как при нормальном завершении тела `try` (без исключений), так и при возникновении исключения (даже если исключение обработано не будет):

```
try:
    1/0
except ZeroDivisionError:
    print('error', end=' ')
finally:
    print('finally')
# вывод: error finally

try:
    1/1
except ZeroDivisionError:
    print('error', end=' ')
finally:
    print('finally')
# вывод: finally

try:
    1/0
finally:
    print('finally')
# вывод: finally; исключение не обработано
```

Исключение можно выбросить вручную с помощью оператора `raise`, указав тип выбрасываемого исключения:

```
raise ZeroDivisionError
```

1.12. Основы функций

В Python можно объявлять собственные функции, однако при этом нужно помнить, что в языке отсутствует возможность перегрузки функций.

Для объявления функций нужно воспользоваться оператором `def`:

```
def f():
    pass
```

В этом примере объявляется функция `f`, которая не принимает никаких аргументов, на что указывают пустые круглые скобки после имени функции. При этом не нужно указывать тип возвращаемого значения благодаря динамической типизации. Ключевое слово `pass` применяется в качестве заглушки тела функции (или другого составного оператора) и обозначает пустое тело.

Чтобы вызвать функцию, требуется указать название функции с последующими круглыми скобками, внутри которых перечисляются аргументы функции. Вызов предыдущей функции осуществляется через `f()`.

Чтобы вернуть значение из функции, следует воспользоваться оператором `return`:

```
def g():
    return 1
```

```
a = g() # a равняется 1
```

Функция g при вызове будет возвращать 1. Если в теле функции не указан возвращаемый результат, тогда эта функция возвращает None, таким образом, f() вернёт None.

Чтобы указать, что функция принимает аргументы, нужно в круглых скобках перечислить названия всех параметров без их типа:

```
def mysum(a, b):
    return a + b
mysum(1, 2) # возвращает 3
```

Кроме этого, из функции можно вернуть сразу несколько значений, указав их после return и разделив их запятыми:

```
def h(a, b):
    return a + b, a * b
```

Для получения значений из подобной функции можно воспользоваться следующим кодом для вызова функции:

```
m, n = h(2, 3) # m равняется 5, n равняется 6
```

Обычно во время вызова аргументы передаются по позициям. Порядок аргументов при вызове функции должен совпадать с порядком аргументов в объявлении функции. Вместе с тем в Python можно передавать аргументы во многие функции по их имени. Для этого нужно указать название переменной, знак равно и значение переменной:

```
m, n = h(a=2, b=3)
```

Подобный способ передачи аргументов позволяет улучшить читаемость кода.

1.13. Функция print

В предыдущих разделах описаны основы различных аспектов Python. В этом разделе будет подробно рассмотрена функция print, знание особенностей которой необходимо для получения требуемого вывода.

Функция print принимает произвольное количество объектов, которые нужно вывести на экран:

```
print(1, 2, 3) # вывод: 1 2 3
```

Все переданные объекты преобразуются в строку и выводятся в консоль с пробелом между ними. После вывода всех объектов осуществляется переход на новую строку. Функция print позволяет настроить строку, выводимую между объектами, и строку, выводимую после всех объектов, с помощью ключевых параметров sep и end соответственно. Значениями по умолчанию для sep является строка ' ' (пробел), для end — '\n':

```
print(1, 2, 3, sep=', ', end='.\n') # Вывод: 1, 2, 3.\n
```

1.14. Импортирование модулей

В Python присутствует большое количество полезных функций, которые находятся во встроенных модулях. Для вызова таких функций сначала необходимо импортировать содержащий их модуль. *Модули* — это обычные файлы с программой на Python, которые содержат различные функции.

Рассмотрим способы импортирования модуля на примере использования функции вычисления квадратного корня `sqrt` из модуля с математическими функциями `math`. Для подключения модуля применяется оператор `import`:

```
import math
```

После подключения можно вызывать любую функцию из модуля `math`. Для вызова функции необходимо после имени модуля указать название функции:

```
a = math.sqrt(9) # a равняется 3.0
```

Чтобы не было необходимости каждый раз использовать название модуля перед именем функции, можно подключить функцию без подключения всего модуля. Для этого применяется конструкция `from`/`import`:

```
from math import sqrt
a = sqrt(9) # a равняется 3.0
```

Кроме этого, можно указать псевдонимы для подключённых модуля и функции с помощью `as`:

```
import math as m
m.sqrt(9)  # возвращается 3.0
from math import sqrt as s
s(9)       # возвращается 3.0
```

1.15. Создание собственных модулей

Рассмотрим создание собственных модулей с функциями, которые могут быть вызваны из других файлов. Сначала необходимо создать файл .py с названием, который соответствует шаблону, указанному в разделе 1.3, например, `numbers_functions.py`. Затем в этот файл необходимо переместить функцию, которая будет вызываться из разных файлов, например:

```
# файл numbers_functions.py
def is_prime(n):
    # тело функции
```

Для вызова функции из созданного модуля требуется первым делом создать файл (например, `a.py`) в той же папке, в которой находится импортируемый модуль. Далее в этом файле можно применить способы, указанные в предыдущем разделе:

```
# файл a.py
from numbers_functions import is_prime
is_prime(5)
```

1.16. Справка

Для получения справки необходимо воспользоваться функцией `help`. Чтобы узнать подробную информацию о функции, в функцию `help` необходимо передать ссылку на функцию, указав название функции без круглых скобок. Для извлечения документации к методу необходимо после названия класса и точки указать название метода без круглых скобок. Пример получения справки представлен ниже:

```
help(len)       # документация по функции len
help(str.lower) # документация по методу строки lower
```

Для получения документации к классу/модулю нужно указать название класса/модуля в качестве аргумента `help`, но модуль должен быть перед этим импортирован:

```
help(int)  # документация к классу int
import math
help(math) # документация к модулю math
```

Для получения всех методов, доступных в классе можно воспользоваться функцией `dir`, которая принимает название класса и возвращает список названий методов:

```
dir(str) # возвращает список методов строки
```

1.17. Обозначения в пособии

В этом разделе описаны обозначения, которые будут использоваться далее в этом пособии при описании исходного кода.

```
a = f() # a → 1
```

Выражение в комментариях «`a` → `1`» обозначает, что после выполнения показанной выше строки кода в переменной `a` будет храниться значение `1`, то есть результатом вызова функции `f` будет значение `1`, которое присваивается в переменную `a`.

Следующий код использует обозначение «→ `3`», чтобы показать, что выражение вызова функция `g` возвращает `3`, при этом данное значение никуда не присваивается:

```
mysum(1, 2) # → 3
```

Символ > будет использоваться для отображения содержимого, выведенного в консоль при помощи `print`. Переход на новую строку, выполняемый функцией `print` после вывода всех объектов, при `end='\n'` не отображается. Далее представлены некоторые примеры обозначений информации, выводимой в консоль:

```
print('hello')          # > hello
print('hello\n')        # > hello\n
print(repr('hello\n')) # > 'hello\n'
```

Во второй строке выводится `'hello'` с переходом на новую строку, а в третьей такого перехода не выполняется.

Для уменьшения количества кода договоримся о том, что переменная будет действительна до конца раздела, например, в следующем коде используется переменная, вычисленная в самом начале текущего раздела:

```
print(a) # > 1
```

1.18. Резюме

Язык программирования Python («Пайтон») является интерпретируемым языком с динамической типизацией, который поддерживает парадигмы императивного, структурного, функционального и объектно-ориентированного программирования.

При работе с переменными не требуется объявление и указание их типа, переменные создаются в момент первого присваивания. Функция `print` выводит переданный аргумент в консоль, позволяя вывести даже сложные объекты, например, списки. Функция `input` считывает строку с консоли и возвращает её в качестве результата.

Числа, строки, списки и кортежи являются базовыми типами данных, которые доступны в Python. Числа поддерживают все операции, используемые и в других языка программирования. Строки являются последовательностью символов, а списки — последовательностью произвольных объектов. Обе эти коллекции поддерживают индексацию и поэлементный обход с помощью оператора `for` и функции `range`.

В Python присутствуют операторы ветвления `if`/`elif`/`else` и операторы циклов `for` и `while`.

Обработка исключений выполняется при помощи операторов `try`/`except`/`finally`, а возбуждается исключение применением оператора `raise`.

Для определения собственной функции можно воспользоваться оператором `def`. Тип возвращаемого значения и аргументов указывать не нужно. В круглых скобках после имени функции задаются аргументы, перечисленные через запятую. Оператор `return` прерывает выполнение функции и возвращает значение, указанное после `return`. Чтобы вернуть несколько объектов из функции, после `return` нужно указать все возвращаемые значения через запятую.

Для определения функции, которую можно вызвать из разных файлов, необходимо поместить эту функцию в модуль. Для импортирования функции из модуля можно воспользоваться конструкцией `import` или `from`/`import`.

1.19. Рекомендованная литература

Два тома книги Лутца «Изучаем Python» (см. [1]) являются одними из наиболее исчерпывающих книг по языку программирования Python. Также дополнительную информацию можно подчерпнуть из книг [2–6].

Документацию и туториал по языку Python можно найти по адресам, указанным в списках литературы в [8, 9].

2. Как выполняются программы

В этой главе подробно рассматривается работа интерпретатора Python во время выполнения программ. Программисту не обязательно знать внутреннее устройство интерпретатора, потому что выполняемая интерпретатором работа скрыта от программиста. Но знание особенностей функционирования интерпретаторов может помочь в выборе интерпретатора Python, например, для повышения производительности программ.

2.1. Основы интерпретатора Python

Под Python, кроме языка программирования, понимается также и интерпретатор, который исполняет Python-программы. Сам стандарт языка не накладывает ограничения на способ реализации интерпретатора, а в одном из последующих разделов будут рассмотрены различные интерпретаторы Python.

После установки интерпретатора в системе будут установлены сам интерпретатор и стандартная библиотека Python.

2.2. Интерпретаторы и компиляторы

В текущем разделе рассматриваются основные виды интерпретаторов и компиляторов.

Классический компилятор — программа, преобразующая исходный код программы в набор машинных команд. Результатом компиляции является исполняемая программа, запускаемая на компьютере напрямую. Компиляция выполняется на машины, поддерживающие один набор команд, с конкретной операционной системой. Для машины с другим набором команд или с другой операционной системой нужна отдельная компиляция. В случае наличия ошибок в тексте (например, синтаксические ошибки, ошибки в названии переменных) компиляция прерывается с ошибкой. Программы доставляются пользователям в виде исполняемых программ. Примерами классического компилятора являются компиляторы языка C++.

Классический интерпретатор — программа, построчно обрабатывающая и выполняющая исходный код программы, то есть программа выполняется на компьютере не напрямую, а с помощью интерпретатора. Интерпретируемые программы имеют большу́ю переносимость, то есть могут выполняться на любой машине и в любой операционной системе, для которых разработан интерпретатор. Любые ошибки в тексте будут обнаружены лишь во время выполнения команды. В плане производительности интерпретируемые программы выполняются гораздо медленнее компилируемых программ из-за необходимости обработки исходного кода во время выполнения программы. Существенный недостаток интерпретатора — отсутствие оптимизации кода. Пользователям программы поставляются в виде исходного кода или фиксированных двоичных файлов (см. 2.8). В настоящее время классические интерпретаторы практически не используются в виду низкой производительности.

Для улучшения производительности интерпретируемых программ применяются транслирующие интерпретаторы. *Транслирующий интерпретатор* — программа, транслирующая исходный код программы в байт-код и выполняющая полученный байт-код на компьютере. *Байт-код* — специальный кроссплатформенный язык, который является промежуточной формой между языком высокого уровня и машинным кодом. Байт-код выполняется так называемой *виртуальной машиной* строчка за строчкой. Чаще всего файл с программой транслируется в байт-код полностью, благодаря чему можно уже на этом этапе обнаружить синтаксические ошибки (но не ошибки в названиях переменных). Программы на транслирующих интерпретаторах выполняются быстрее, чем на классических

интерпретаторах, потому что программа переводится в байт-код один раз, а выполнение байт-кода происходит быстрее, чем исполнение исходного кода. Трансляция в байт-код может выполняться как во время запуска исходного кода программы, так и до выполнения программы. Программа доставляется до пользователей в виде исходного кода, байт-кода или фиксированных двоичных файлов (см. 2.8). Примером транслирующего интерпретатора является CPython (стандартная реализация Python) (см. 2.3).

Транслирующий компилятор — программа, транслирующая исходный код в байт-код и выполняющая во время исполнения программы компиляцию байт-кода в машинный код, который в дальнейшем и будет выполняться во время вызова скомпилированного байт-кода. Байт-код выполняется виртуальной машиной, а компиляцию производит *JIT-компилятор*. Результат компиляции программы сохраняется в оперативной памяти и удаляется после завершения программы. Сама компиляция выполняется перед первым вызовом кода, а последующее обращение к этому коду приведёт к вызову уже скомпилированного кода. Хотя JIT-компилятор тратит время на компиляцию байт-кода, это компенсируется ускорением, достигаемым за счёт уменьшения времени выполнения этого кода при последующих вызовах. Транслирующий компилятор обладает как высокой переносимостью, так и высокой производительностью, выполняя программы быстрее транслирующего интерпретатора. Транлсяция в байт-код всегда выполняется до начала выполнения самой программы. Доставка программ осуществляется в виде байт-кода или фиксированных двоичных файлов (см. 2.8). Примерами транслирующих компиляторов являются современные компиляторы: CLR (система выполнения кода .Net) и JVM (виртуальная машина Java).

Стоит отметить, что создание функций и классов в интерпретаторах происходит во время выполнения программы.

2.3. Реализация CPython

Стандартной и самой популярной реализацией Python является CPython [7]. В случае установки Python с официального сайта устанавливается именно этот интерпретатор. CPython реализован на ANSI C, что позволяет запустить интерпретатор на любой системе, для которой существует компилятор с языка C.

В момент запуска программы весь файл с программой транслируется в байт-код. Поэтому при наличии синтаксической ошибки, выполнение программы даже не начнётся. Во время запуска программы её байт-код создаётся и хранится в оперативной памяти, а после завершения программы байт-код удаляется из памяти.

Если во время выполнения программы импортируется другой модуль, тогда он преобразуется в байт-код, который сохраняется с именем `filename.cpython-39.pyc`. Сам рус-файл создаётся в папке `__pycache__`, которая находится в той же папке, что и импортированный файл. К названию файла добавляется название интерпретатора и его версия. Во время следующего запуска программы, когда модуль будет импортироваться, шаг трансляции модуля в байт-код будет пропущен, и будет использован уже существующий байт-код. Если рус-файлы нельзя записать на диск, тогда байт-код сохраняется в памяти и удаляется в конце выполнения программы.

Транслированный байт-код исполняется *виртуальной машиной Python* (*Python's virtual machine*, *PVM*), которая является частью и последним шагом интерпретатора. Машина PVM в цикле проходит по инструкциям в байт-коде и выполняет их по очереди, переводя байт-код в машинный код. Создание переменных, функций и классов происходит во время выполнения программы.

Производительность CPython не может сравниться со скоростью работы скомпилированных программ в виду необходимости трансляции байт-кода в машинный код во вре-

мя выполнения. Однако встроенные функции стандартной библиотеки Python являются скомпилированными и потому выполняются со скоростью C++. Стандартная библиотека реализована на `C`, поэтому во время вызова стандартной функции будет выполняется скомпилированный код без шага интерпретации байт-кода. Кроме того, некоторые сторонние библиотеки написаны на C, вследствие чего они тоже выполняются со скоростью C++. Например, популярная библиотека NumPy написа на C и является стандартом де-факто для научных вычислений.

2.4. Альтернативные реализации Python

В этом разделе будут рассмотрены различные реализации языка Python, применяемые для специфических целей.

Целью создания PyPy (см. [13]) была высокая производительность, которая достигается за счёт использования JIT-компилятора. Реализация PyPy транслирует исходный код в байт-код, затем байт-код компилируется JIT-компилятором во время выполнения. Функция может быть вызвана с разными типами аргументами, поэтому JIT-компилятор для каждого набора типов компилирует функцию в отдельный машинный код. Для этого JIT-компилятор отслеживает тип данных параметров функции. Кроме этого, запущенные интерпретатором PyPy программы занимают меньше памяти (по сравнению с запуском в CPython). Однако написанные для CPython программы для запуска на PyPy придётся переписывать.

Реализация Stackless Python является интерпретирующим траслятором, разработанным для обеспечения параллелизма [18].

Jython и IronPython применяются для выполнения Python-программы на JVM (Java Virtual Machine) и CLR (виртуальная машина платформы .NET). Программа полностью запускается на Jython/IronPython, имеет доступ к внутренним библиотекам Java/.NET и может взаимодействовать с другими Java/.NET-программами. Кроме того, модули `JPype` и `Python for .NET` применяются для доступа к компонентам Java и .NET, но сама Python-программа выполняется интерпретатором Python.

2.5. Дополнительные возможности оптимизации

Повышения производительности можно добиться не только изменением модели выполнения программы, но и путём изменения самого языка.

Cython (не путать с CPython) является языком программирования, аналогичным Python, с добавлением конструкций языка C. Среди прочего, в Cython требуется определять тип переменных. Код на языке Cython можно транслировать в код на языке C, после чего можно выполнить компиляцию программы. Благодаря отсутствию интерпретации программы, написанные на Cython, выполняются быстрее.

Кроме того, существует траслятор Shed Skin, который транслирует статически типизированный код на Python в код на C++ с последующей компиляцией в машинный код. Однако на данный момент проект ShedSkin был заброшен, а часть наработок перешла в другие проекты, например, numba.

2.6. Система управлений пакетами `pip`

Внешние пакеты являются важной составляющей Python. Большинство пакетов можно найти в системе PyPI (Python Package Index) [13]. Их можно установить вручную, но это не удобно. Для быстрой и удобной установки сторонних библиотек (пакетов) используется система управления пакетами `pip`:

```
pip install packagename
```

Кроме этого, система `pip` позволяет обновлять и удалять уже установленные библиотеки.

В CPython `pip` устанавливается автоматически.

Большинство пакетов написаны на Python и легко устанавливаются с помощью `pip`. Но некоторые библиотеки написаны на C. Эти пакеты во время установки требуют компиляции, из-за чего может возникнуть сложность во время установки таких пакетов.

2.7. Дистрибутивы Python с пакетами

В предыдущем разделе были рассмотрены проблемы установки библиотек, написанных на языке C. Кроме этого, разрабтчику часто требуется большое количество внешних пакетов, которые иногда тяжело устанавливать вручную. Для решения этих двух проблем созданы сторонние дистрибутивы Python, содержащие стандартный интерпретатор CPython и часто используемые пакеты: `Anaconda` [14], `Python(x, y)` [15] и `Pyzo` (основан на `Anaconda`) [16]. Эти дистрибутивы устанавливают интерпретатор Python и большое число сторонных библиотек.

2.8. Фиксированные двоичные файлы

Для запуска написанной на Python программы требуется, чтобы на компьютере были установлены интерпретатор и все используемые программой библиотеки. Однако это сильно ограничивает возможность запуска программы на произвольном компьютере, так как не на всех комьпютерах присутствует интерпертатор Python. Для удобного запуска Python-программ на любых компьютерах применяются фиксированные двоичные файлы.

Фиксированный двоичный файл — исполняемый файл, в котором хранятся объединённые машина PVM, все необходимые библиотеки и программа в виде байт-кода. Исходный код при помещении в фиксированный двоичный файл транслируется в байт-код, а не компилируется. Фиксированный двоичный файл создаётся при помощи специальных инструментов, например, `py2exe` для Windows (см. [19]). В операционной системе Windows фиксированные двоичные файлы создаются в виде exe-файлов. В момент запуска фиксированного двоичного файла запускается машина PVM, которая начинает выполнять программу оператор за оператором. Таким образом, фиксированные двоичные файлы не повышают производительность программы, а дают возможность простого и удобного распространения и запуска программы. Недостаток использования фиксированного двоичного файла состоит в том, что размер полученного двоичного файла становится больши́м (по сравнению с самой программой), потому что в программу добавляется машина PVM.

Инструмент `py2exe` создаёт фиксированный двоичный файл только для Windows. Для создания фиксированного двоичного файла для Windows, Linux, Mac OS можно воспользоватся инструментом `cx_Freeze` (см. [20]).

2.9. Резюме

Существует четыре основных типа моделей выполнения кода: классический компилятор, транслирующий компилятор, транслирующий интерпретатор и классический интерпретатор. Полученный код этими системами имеет разную степень переносимости и производительности. Трансляция в байт-код (промежуточный низкоуровневый язык) выполняется для большей переносимости и производительности. JIT-компилятор обеспечивает компиляцию во время выполнения в целях ускорения кода.

Стандартный интерпретатор CPython представляет собой интепретирующий транслятор, который компилирует исходный код в байт-код и исполняет его во время выполнения. Интерпретатор позволяет запустить уже транслированный байт-код.

Для повышения производительности Python-программу можно запустить с помощью PyPy, переписать в программу на языке Cython или транслировать в код C++ при помощи Shed Skin.

3. Операции, выражения и операторы

Английские термины в области программирования в различной литературе могут переводиться на русский язык по-разному. В этом учебном пособии используется следующий перевод для английских терминов: operator — операция, expression — выражение, statement — оператор.

3.1. Компоненты программы

Рассмотрим, из каких строительных блоков в Python строятся программы.

- Программы состоят из модулей.
- Модули включают в себя операторы.
- Операторы состоят из выражений.
- Выражения конструируются из объектов и операций.

Далее отдельно рассмотрим каждый компонент программы.

3.2. Операции (operators)

Данные в Python хранятся в виде объектов. Объекты могут быть заданы либо в виде литералов (например, `1`, `'a'`), либо в виде переменных. Для манипулирования объектами используются *операции* (*operators*), возвращающие в качестве результата другой или тот же самый объект, например, операция `+` для чисел возвращает их сумму. Объекты, над которыми выполняется операция, называются *операндами* (*operands*).

Полный список операций в Python представлен в таблице 1. В этой таблице операторы без указания способа применения (без использования переменной `a`) являются *бинарными*, то есть они принимают два операнда, например, `*` и `/`. Дополнительно в таблице для многих операций указаны разделы, в которых более подробно описаны эти операции.

В таблице 1 представлено lambda-выражение, которое используется для создания анонимных функций. Операция `@`, добавленная в Python 3.5, предназначена для умножения матриц, однако встроенные типы данных Python не поддерживают эту операцию. Операция `in` выполняет проверку вхождения элемента в коллекцию (например, в список), а для строк осуществляет проверку наличия подстроки. Операция `not in` инвертирует результат предыдущей операции. Под отображением (display) в таблице понимается либо заданные вручную список (см. 7.2), кортеж (см. 7.12), словарь (см. 9.1), множество (см. 10.2), либо включения списков (см. 7.10), словарей (см. 9.7) и множеств (см. 10.3).

Операции для разных типов могут работать по-разному: операция `+` для чисел выполняет операцию сложения, а для строк — конкатенацию. Пользовательский класс может определить собственный способ выполнения операций над объектами этого класса.

3.3. Операции сравнения

В Python имеются различные операции сравнения: `<`, `<=`, `>`, `>=`, `!=`, `==`. Они возвращают либо `True`, либо `False`. Действие этих операций зависит от типа данных объекта, к которому применяются эти операции: операция `<` числа сравнивает математически, а строки — лексикографически.

Исключая числовые типы, объекты разных типов при сравнении на равенство будут возвращать `False`.

Таблица 1 — Таблица операций в порядке увеличения приоритета

Операция	Описание
`:=`	Выражение присваивания (см. 3.10)
`lambda`	lambda
`if - else`	Условное выражение (тернарный оператор) (см. 3.12)
`or`	Логическое ИЛИ (см. 3.5)
`and`	Логическое И (см. 3.5)
`not a`	Логическое НЕ (см. 3.5)
`in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	Проверки вхождения элемента в коллекцию и равенства ссылок (см. 4.3), сравнения (см. 3.3)
`|`	Побитовый ИЛИ (см. 5.9)
`^`	Побитовый XOR (см. 5.9)
`&`	Побитовый И (см. 5.9)
`<<`, `>>`	Сдвиги влево и вправо (см. 5.9)
`+`, `-`	Сложение и вычитание
`*`, `@`, `/`, `//`, `%`	Умножение, умножение матриц, деление, деление нацело, вычисление остатка от деления (см. 5.2)
`+a`, `-a`, `~a`	Значение `a` (сохранение знака), отрицательное `a` (смена знака), побитовое НЕТ (см. 5.9)
`**`	Возведение в степень (см. 5.2)
`await a`	await-выражение
`a[index]`, `a[i1:i2]`, `f(arguments)`, `a.attr`	Индексация, срез, вызов, ссылка на атрибут
`(expressions)`, `[expressions]`, `{key:value, ...}`, `{expressions...}`	Группировка, генераторное выражение, отображения кортежей, списков, словарей и множеств

В отличие от других языков программирования, операции сравнения можно сцеплять вместе: проверка `a < b < c` будет выполняться традиционным для математики способом:

```
a < b and b < c
```

В сцепленных сравнениях могут быть задействованы любые операции сравнения:

```
a < b == c  # аналогично a < b and b == c
a == b == c # аналогично a == b and b == c
a < b > c   # аналогично a < b and b > c
```

Сцепленные сравнения могут иметь произвольную длину. Далее продемонстрировано, как длинное сцепленное сравнение преобразуется в обычное:

```
a_1 op_1 a_2 op_2 a_3 ... a_{n-1} op_{n-1} a_n # преобразуется в
a_1 op_1 a_2 and a_2 op_2 a_3 and ... and a_{n-1} op_{n-1} a_n
```

Единственное отличие сцепленного сравнения от полной формы состоит в том, что в сравнении `a` op_1 `b` op_2 `c` выражение `b` вычисляется один раз. При этом в обоих случаях выражение `c` вычисляется, только если выражение `a` op_1 `b` является истинным. Это особенно важно, когда выражения `b` и `c` являются вызовами функции:

```
f() < g() < h()
```

3.4. Истинность объектов

Перед рассмотрением операций `and`/`or`/`not` нужно конкретизировать понятие истинности объекта в Python.

Любой объект можно протестировать на истину внутри `if`/`elif` или `while`. Проверка на истинность выполняется с помощью неявного преобразования объекта в `bool`. Далее описаны объекты, которые преобразуются в `False`.

- Константы `None` и `False`.
- Нулевые значения чисел: `0`, `0.0`, `0j` (комплексное число, см. 5.8), `Decimal(0)` (десятичное число, см. 5.13.1) и `Fraction(0)` (рациональное число, см. 5.13.2).
- Пустые коллекции и последовательности `''` (пустая строка), `[]` (пустой список), `()` (пустой кортеж), `{}` (пустой словарь, см. 9), `set()` (пустое множество, см. 10), `range(0)` (пустой диапазон).

Все остальные объекты, в частности `True`, являются истинными, если в классе не определено обратное, например, `1` и `'a'` являются истинными.

Далее показан пример использования критерия истинности целых чисел: цикл `while` продолжается до тех пор, пока пользователь не введёт `0`:

```
n = -1 # начальное значение
while n:
    # вычисления
    n = int(input('Input number:\n'))
```

Для получения булевского значения объекта его нужно передать в конструктор `bool`: `bool('s')` → `True`.

3.5. Операции `and`, `or` и `not`

Операции `and`, `or` и `not` выполняют логическое И, ИЛИ и НЕ соответственно:

```
1 < 2 and '1' < '2' # Истинное
```

Операции `and`, `or` и `not` используют определение истинности, описанное в предыдущем разделе. Таким образом, операндами этих операций могут быть произвольные объекты. Например, для целых чисел `a` и `b` тело `if` в следующем примере выполнится, если оба эти числа не равны нулю.

```
if a and b:
    print(1/a + 1/b)
```

В Python операции `and` и `or` обладают двумя важными свойствами. Первое свойство состоит в том, что интерпретатор осуществляет *укороченное выполнение* (*short-circuit evaluation*) сравнения: выполнение операции прекращается, когда становится известен результат операции. Операция `and` завершается, если первый операнд является ложным (тогда результат всей операции является ложным). Операция `or` завершается, если первый операнд является истинным (тогда результат всей операции является истинным).

Благодаря этому свойству, программы выполняются быстрее. Далее функция `g()` вызывается, только если результат выполнения функции `f` является истинным:

```
if f() and g():
    # тело if
```

Кроме того, укороченное сравнение позволяет писать более элегантный код и избегать исключений:

```
if a and 1/a > b:
    # тело if
if 0 <= i < len(L) and L[i] > 0: # L - список
    # тело if
```

В предыдущем примере сначала выполняется проверка того, что значение `a` не равняется нулю. Затем, при ненулевом значении `a`, вычисляется значение `1/a` с последующими сравнением `>` и проверкой истинности. Таким образом, в этом примере укороченное выполнение сравнения позволяет избежать деления на ноль. Во втором условии индекс в списке проверяется на выход за границы списка, после чего выполняется доступ к элементу списка по уже проверенному индексу. Если бы в Python отсутствовало укороченное выполнение сравнения, тогда во время проверки обоих условий могло бы быть выброшено исключение.

В случае отсутствия укороченного сравнения в Python, первое условие из предыдущего примера нужно было бы записывать следующим образом:

```
if a:
    if 1/a:
        # тело
```

Как видно из примера, укороченное сравнение в некоторых ситуациях упрощает написание кода.

Второе важное свойство операций `and` и `or` заключается в том, что они возвращают последний проверенный операнд, а не значение `True`/`False`.

Рассмотрим выражение `a and b`. Если объект `a` является ложным, тогда возвращается `a`, иначе возвращается `b`:

```
1 and 3  # → 3
[] and 4 # → []
```

Если в выражении `a or b` объект `a` является истинным, тогда возвращается `a`, в противном случае — `b`:

```
1 or 3 # → 1
0 or 3 # → 3
```

Рассматриваемое свойство особенно полезно при использовании операции `or`. Предположим, пользователь должен ввести данные через консоль. Если введённая пользователем строка не является пустой, тогда сохраняем это значение. В противном случае в переменную присваивается значение по умолчанию:

```
s = input('input text:') or 'default'
```

Также это свойство используется в функциях, принимающих значение по умолчанию:

```
def f(arr=None):
    arr = arr or []
```

Если функция `f` вызывается без аргументов, тогда параметр `arr` инициализируется `None`, а в первой строке переменной `arr` будет присвоена ссылка на пустой список.

В отличие от операций `and` и `or`, операция `not` всегда возвращает булево значение:

```
not 1  # → False
not [] # → True
```

3.6. Выражения (expressions)

Объединение объектов (литералов, переменных) и операций, применённых к этим объектам, образует *выражение* (*expression*):

```
1 + 2     # выражение сложения чисел
'1' + '2' # выражение конкатенации строк
a[1]      # выражение применения операции индексации
f()       # выражение вызова функции
[1, 2, 3] # выражение создания списка
```

Сложные выражения могут состоять из нескольких операций:

```
2 + 2 * 2
```

При вычислении таких выражений становится важным порядок выполнения операций. В таблице 1 операторы перечислены в порядке увеличения приоритета: в первой строке операции с меньшим приоритетом, в последней — с бóльшим, а операции в одной ячейке имеют одинаковый приоритет:

```
2 + 2 * 2 # сначала будет вычислено выражение 2*2
```

Приоритет возведения в степень выше, чем у унарной операции `-`, поэтому `-1**2` вернёт `-1`.

Если операции имеют одинаковый приоритет, тогда эти операции вычисляются слева направо. Для изменения порядка вычислений в сложном выражении можно воспользоваться круглыми скобками для группировки выражения:

```
(2 + 2) * 2 # сначала будет вычислено выражение 2+2
```

Значение выражения *вычисляется* интерпретатором. При этом значение любого выражения можно присвоить переменной:

```
a = 2 * 2
b = 2 + a
```

Выражения всегда выполняются слева направо, не считая выполнения оператора присваивания. Далее представлены взятые из документации [11] примеры сложных выражений, в которых выражения будут вычисляться в порядке `expr1`, `expr2`, `expr3`, `expr4`:

```
expr1, expr2, expr3, expr4           # создание кортежа
(expr1, expr2, expr3, expr4)         # создание кортежа
{expr1: expr2, expr3: expr4}         # создание словаря
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5) # вызов expr1
expr3, expr4 = expr1, expr2          # присваивание
```

3.7. Операторы (statements)

Выражение (даже очень сложное) вычисляется до единственного значения. *Оператор* (*statement*) *выполняется* интерпретатором без вычисления значений. Операторы выполняются друг за другом. Таким образом, выражения вычисляют значения, тогда как операторы выполняют действия (не связанные с вычислением значения).

Операторы можно разделить на две группы: простые и составные. *Простые* операторы записываются в одну строку, например, `break`. *Составные* операторы, в свою очередь, состоят из заголовка и тела, например, `if`. Составные операторы записываются в несколько

строк: заголовок записывается в отдельной строке, а тело оператора записывается с отступом относительно заголовка. Оператор, который должен выполняться после составного оператора, должен иметь вертикальный отступ, равный отступу заголовка, то есть должен быть записан без отступа:

```
if expr: # → заголовок
    ...  # → тело
# следующий за if оператор
```

Операторы конструируются из выражений (выражения являются частью операторов):

```
a = b + c
d = f()
```

В этом примере в первой строке перед выполнением оператора присваивания вычисляется значение суммы, а во второй строке вызывается функция.

Любое выражение может быть использовано в качестве оператора:

```
b + c
f()
```

Вычисленные таким образом значения никуда не присваиваются, однако функция может иметь побочный эффект.

Впрочем выражения (следовательно, и операторы) не могут содержать операторы, потому что выражения требуют значения, а операторы не вычисляют значения:

```
if a = 1: # ошибка: оператор = внутри другого оператора
    # тело
```

В этом примере оператор `if` в заголовке ожидает выражения, но передаётся оператор, из-за чего происходит синтаксическая ошибка.

В таблице 2 представлен полный список операторов Python. В Python отсутствует оператор `switch`/`case`, однако этот оператор может быть реализован через оператор `if`/`elif`/`else`. Оператор `del` удаляет переменные и элементы последовательности:

```
del a    # удаление переменной a
del L[0] # удаление нулевого элемента списка
```

После удаления переменной попытка получения значения этой переменной приведёт к ошибке.

Далее будут подробно рассмотрены большинство операторов из таблицы 2.

3.8. Однострочные и многострочные операторы

Как упоминалось ранее, простые операторы записываются с новой строки:

```
a = 1
b = 2
print(a, b)
```

Однако в Python есть возможность записать несколько простых операторов в одну строчку, поставив между ними точку с запятой (`;`):

```
a = 1; b = 2; print(a, b)
```

Таблица 2 — Список операторов в Python

Оператор	Тип	Описание
= (присваивание)	Простой	Создание ссылок
Произвольное выражение	Простой	Выполнение выражения (см. 3.6), например, вызов функции
`if`/`elif`/`else`	Составной	Ветвление (см. 1.9)
`for`/`else`	Составной	Цикл по последовательности (см. 3.14)
`while`/`else`	Составной	Универсальный цикл с предусловием (см. 3.15)
`pass`	Простой	Пустой заполнитель тела составного оператора (см. 3.16)
`break`	Простой	Выход из цикла
`continue`	Простой	Переход к следующей итерации цикла
`def`	Составной	Определение функции (см. 1.12)
`return`	Простой	Возвращение результата из функции (см. 1.12)
`yield`	Простой	Отправление значения вызывающему коду
`global`	Простой	Объявление глобальной переменной
`nonlocal`	Простой	Объявление нелокальной переменной
`import`	Простой	Импортирование модуля (см. 1.14)
`from`	Простой	Импортирование атрибута модуля (см. 1.14)
`class`	Составной	Определение класса
`try`/`except`/`finally`	Составной	Перехват исключений (см. 1.11)
`raise`	Простой	Генерация исключений (см. 1.11)
`assert`	Простой	Отладочные проверки
`with`/`as`	Составной	Диспетчер контекста
`del`	Простой	Удаление ссылок

Заголовок составных операторов (`if`, `while`) должен быть записан в отдельной строке. Если тело оператора состоит из одной строки, тогда это тело можно указать сразу после двоеточия. В этом случае инструкция, следующая за оператором, не относится к телу составного оператора:

```
if a > 1: b = 2; print(b)
print(a) # не тело if
```

Описанными выше инструментами Python не стоит злоупотреблять, так как слишком большое количество операторов в одной строке может ухудшить читаемость кода.

Далее рассмотрим как однострочные операторы могут быть записаны в несколько строк. Первый способ основан на том, что содержимое скобок `[]`, `{}` и `()` может быть записано в нескольких строчках:

```
a = (1+2 +
    3) # a → 6
b = [1, 2,
    3] # b → [1, 2, 3]
```

Вторая и последующие строки многострочных операторов могут иметь отступ, отличающийся от отступа первой строки многострочного оператора.

Второй способ заключается в том, что в конце строки можно записать обратную косую черту, при этом после неё не должно быть никаких других символов (кроме перевода на новую строку):

```
a = 1 + 2 \
    - 3 + 4
```

Этот способ может использоваться автоматически текстовыми редакторами.

3.9. Оператор присваивания

Оператор присваивания = (*assignment statement*) присваивает ссылку на объект. В правой части должно находиться выражение, вычисленное значение которого присваивается левой части (переменной или элементу объекта):

```
a = 2 + 2 * 2 # a → 6
L = [1, 2, 3] # L → [1, 2, 3]
L[0] = 4      # L → [4, 2, 3]
```

Левая часть присваивания называется *целью* (*target*).

В Python имеется *единая модель присваивания*, которая применяется в различных контекстах, например, в цикле `for` и при вызове функции:

```
for i in range(5):
    # тело цикла
f(1)
```

В этом примере на каждой итерации в `i` копируется ссылка на объект, получаемый от `range`, а ссылка на `1` будет скопирована в параметр функции `f`.

Представленные выше примеры демонстрируют базовую форму оператора присваивания. В таблице 3 продемонстрированы все формы оператора присваивания, которые позволяют решать различные задачи.

Таблица 3 — Формы операторов присваивания

Оператор	Описание
`a = 1`	Базовая форма присваивания
`a = b = 1`	Групповое присваивание
`a += 1`	Дополненное присваивание
`a, b = 1, 2`	Присваивание кортежа
`a, b = [1, 2]`	Присваивание списка
`a, b, c = '123'`	Присваивание последовательности
`a, *b = '123'`	Расширенная распаковка последовательности

Групповое присваивание (*multiple-target assignment*) позволяет присвоить вычисленное значение нескольким целям:

```
a = b = c = '1'  # a → 1; b → 1; c → 1;
d = L[0] = e = 5 # d → 5; L → [5, 2, 3]; e → 5;
```

Оператор присваивания является правоассоциативным, то есть выполняется справа налево: в первой строке предыдущего примера `'1'` присваивается сначала `c`, затем `b` и в конце `a`. Это групповое присваивание эквивалентно:

```
c = '1'
b = c
a = b
```

Однако групповое присваивание более очевидно выражает эту идею.

Групповое присваивание работает благодаря тому, что оператор = сначала присваивает значение, а затем возвращает его. Если используется групповое присваивание, тогда это возвращаемое значение передаётся находящемуся слева оператору.

Дополненное присваивание (*augmented assignment*) `a binop= b` является сокращенной формой `a = a binop b`, где `binop` — бинарная операция. Далее представлен список всех вариантов дополненного присваивания:

`a += b`	`a -= b`	`a *= b`	`a /= b`
`a //= b`	`a %= b`	`a <<= b`	`a >>= b`
`a **= b`	`a &= b`	`a |= b`	`a ^= b`

Рассмотрим пример использования дополненного присваивания:

```
a = b = 1
b += 2      # b → 3
b *= a + b # сначала вычисляется правая часть; b → 12
```

Стоит заметить, что в Python нет операций инкремента `a++` и `a--`, потому что числа в Python являются неизменяемыми. Для инкремента и декремента следует применять `a += 1` и `a -= 1` соответственно.

Если объект поддерживает операцию `binop`, тогда он поддерживает и дополненное присваивание с `binop`. Однако есть несколько отличий формы `a = a binop b` и дополненного присваивания, которые позволяют последней форме выполняться быстрее. Первое различие состоит в том, что выражение `a` выполняется один раз:

```
c.d.e += 1
```

В этом примере атрибут `e` будет извлечён только один раз.

Второе различие заключается в том, что в целях ускорения работы программы разработчик может определить в классе своё действие на дополненное присваивание. Например, списки оптимизированы для выполнения операции +=. Пусть `L` является списком, при выполнении `L = L + [1, 2]` будет создан новый соединённый список, а затем этот новый список будет присвоен в `L`. Дополненное присваивание `L += [1, 2]` просто добавляет элементы 1 и 2 в конец списка без создания нового списка.

Присваивание кортежа (*tuple assignment*) и *присваивание списка* (*list assignment*) являются очень полезными формами присваивания. Эти формы позволяют присвоить кортеж или список целям, количество которых равно длине кортежа или списка:

```
a, b = (1, 2)      # a → 1; b → 2
a, b = (1+2, 3*4) # a → 3; b → 12
a, L[0] = (1, 6)  # a → 1; L → [6, 2, 3]
c, d = [2, 3]      # c → 2; d → 3
```

Отличие между присваиванием кортежи и присваиванием списка состоит лишь в типе объекта справа от =.

Присваивание в левой части выполняется слева направо.

Круглые скобки в правой части можно опустить, потому что кортеж задаётся запятыми, а не круглыми скобками:

```
a, b = 1, 2 # a → 1; b → 2
```

Кроме того, слева тоже можно указать скобки, которые являются необязательными:

```
(a, b) = 1, 2 # a → 1; b → 2
```

Присваивание кортежей часто используется для обмена значениями двух переменных:

```
a, b = b, a # a → 2; b → 1
```

Этот способ обмена значениями возможен благодаря тому, что сначала выполняется правая часть и создаётся кортеж из ссылок на переменные `a` и `b`, а уже затем копии ссылок присваиваются в переменные.

Обобщённой формой присваивания кортежа и присваивания списка является *присваивание последовательности* (*sequence assignment*), которое позволяет в правой части указать произвольную последовательность:

```
a, b, c = '123'     # a → '1'; b → '2'; c → '3'
a, b, c = range(3) # a → 0; b → 1; c → 2
```

Количество элементов в последовательности должно совпадать с количеством целей слева:

```
a, b = '123' # ошибка
```

Присваивание последовательности позволяет присваивать вложенные структуры:

```
(a, b), c = [[1, 2], 3] # a → 1; b → 2; c → 3
(a, b), c = ['12', '3'] # a → '1'; b → '2'; c → '3'
```

В этом примере круглые скобки слева являются обязательными, потому что они указывают, что в переменные `a` и `b` должны быть присвоены элементы из последовательности.

Расширенная распаковка последовательности (*extended sequence unpacking*) делает возможным присваивание последовательности переменным, количество которых меньше количества элементов в последовательности. Для этого необходимо перед целью указать `*`, тогда сначала всем остальным целям присвоятся значения, а лишь затем не присвоенные значения объединятся в список и будут присвоены цели с `*`:

```
a, *b = '123'           # a → '1'; b → ['2', '3']
a, *b = range(0, 8, 2) # a → 0; b → [2, 4, 6]
a, *L[0] = [1, 2, 3]    # a → 0; L → [[2, 3], 2, 3]
```

Цель, перед которой указана `*`, может располагаться в любой позиции:

```
*a, b = '123'         # a → ['1', '2']; b → '3'
a, *b, c = [1, 2, 3] # a → 1; b → [2]; c → 3
```

Если все значения были присвоены, тогда цели с символом `*` будет присвоен пустой список:

```
a, *b = '1' # a → '1'; b → []
```

3.10. Выражение присваивания (walrus)

В языках C/C++ можно использовать присваивание внутри оператора (`while` и `if`):

```
if (a = 1) {...}
```

В этом примере `1` присвается в переменную `a`, а затем уже переменная `a` проверяется на истинность. Результатом проверки будет истина, поэтому тело `if` будет выполнено. Но часто подобный код был следствием довольно распространённой ошибки в C/C++: при написании условия `if (a == 1)` был забыт символ «=». Из-за такого рода ошибок в Python запрещено синтаксически применять присваивание внутри операторов. В Python это реализовано благодаря тому, что «=» является оператором, а операторы нельзя помещать внутрь других операторов. Поэтому для получения в Python такого же результата, как в предыдущем примере, при помощи оператора присваивания, можно воспользоваться кодом:

```
a = 1
if a:
    # тело if
```

Рассмотрим решение задачи ввода данных от пользователя, который после окончания ввода должен ввести пустую строку:

```
s = input()
result = ''
while True:
    if s == '':
        break
    result = result + s
    s = input()
```

Решение получилось не слишком элегантным. Для улучшения подобного кода в Python с версии 3.8 имеется возможность использовать *выражение присваивания* (*assignment expression*), или *walrus* (начиная с Python 3.8):

```
result = ''
while s := input():
    result = result + s
```

Выражение `:=` присваивает значение справа в переменную слева и возвращает это значение, которое проверяется на истинность оператором `while`. Если переменной не существовало, тогда она создаётся.

Выражение присваивания может находиться везде, где допустимо выражение (выражение присваивания, возможно, придётся поместить в скобки):

```
a = (b:=c**2 - d**2) + (e:=2*n - k)
```

Подобное применение выражения присваивания позволяет узнать значение подвыражения во время отладки. Приоритет выражения присваивания является самым низким среди всех операций (см. табл. 1), поэтому в предыдущем примере после выполнения присваивания в `b` будет храниться значение `c**2 - d**2`, а `e` — `2*n - k`.

Кроме этого, walrus можно использовать при создании списка:

```
L = [y := f(x), 2*y, 3*y]
```

3.11. Оператор assert

Оператор assert (утверждение) применяется для проверки истинности условия в режиме отладки, поэтому этот оператор может быть использован для проверки условий во время тестирования программы. После оператора необходимо записать условие, которое будет проверяться. Если условие выполняется, тогда выполнение программы продолжается со следующего оператора. В противном случае возбуждается исключение AssertionError:

```
a = 1
assert a == 1 # ничего не происходит
assert a == 0 # исключение
```

Возникновение исключения позволяет найти вызвавшую его строчку кода.

В операторе assert можно указать сообщение, которое будет выведено, если условие не будет выполнено:

```
assert a == 0, 'a is not zero'
```

Если возбуждается исключение, то на экран будет выведено 'a is zero'.

Стоит помнить, что assert является оператором, а не функцией, поэтому не нужно аргументы помещать в круглые скобки:

```
assert (a != 0, 'a is zero')
```

Здесь проверяется на истинность кортеж (a != 0, 'a is zero'), который всегда будет истинным, потому что переданный кортеж не пустой.

Часто утверждения применяются для проверки корректности данных перед началом вычисления, и, кроме того, для проверки правильности вычисленного результата. Рассмотрим применение assert для проверки аргументов и результатов функции, которая должна находить решение квадратного уравнения:

```
def solve(a, b, c):
    assert a != 0
    # вычисление корней
    assert x1 and a*x1**2 + b*x1 + c == 0
    assert x2 and a*x2**2 + b*x2 + c == 0
    return x1, x2
```

В начале тела функции solve выполняется проверка переменной a != 0. Если a равняется нулю, тогда решаемое уравнение не является квадратным, а значит программист, вызвавший эту функцию допустил ошибку. Перед возвращением корней из функции корни подставляются в уравнение для проверки правильности вычисленных значений.

Нужно чётко разделять проверку утверждений и обработку ошибок. Утверждение, задаваемое assert, по логике программы должно выполняться всегда. Если условие не выполняется, тогда в самой программе присутствует ошибка. При этом, обработка ошибок выполняется для ситуаций, которые всё же возможны: отсутствует открываемый файл, пользователь ввёл некорректные данные и тому подобное.

В программе есть специальная переменная __debug__, которая хранит значение, запущена ли программа в режиме отладки или нет. Для отключения отладки и всех операторов assert нужно запустить программу в режиме оптимизации:

```
python -O program.py
```

3.12. Тернарная операция

В Python присутствует *условное выражение* (*conditional expression*), или *тернарная операция* (*ternary operator*). Однако синтаксис этой операции отличается от синтаксиса тернарных операций в других языках программирования:

```
a = value1 if condition else value2
```

Если `condition` является истиной, тогда переменной `a` будет присвоено значение `value1`, иначе — `value2`. Прошлый пример аналогичен следующей конструкции:

```
if condition:
    a = value1
else:
    a = value2
```

В следующем примере осуществляется проверка числа на чётность, а результат этой проверки сохраняется в переменную в виде строки:

```
s = 'чётное' if n%2 == 0 else 'нечётное'
```

Тернарная операция может содержать произвольные выражения:

```
a = expr1 if cond_expr else expr2
```

При выполнении тернарной операции происходит укороченная оценка: сначала вычисляется `cond_expr`, если оно истинно, тогда вычисляется выражение `expr1`, в противном случае — выражение `expr2`. Благодаря этому свойству следующий пример не выбрасывает исключения:

```
a = 1/b if b else 1
```

Однако для лучшей читаемости кода стоит отказаться от чрезмерно сложных тернарных операторов в пользу оператора `if`.

3.13. Операторы `break` и `continue`

Операторы `break` и `continue` могут применяться только внутри циклов: `for` и `while`. Оператор `break` выходит из ближайшего охватывающего цикла, переходя к инструкции, следующей за заголовком ближайшего цикла. Оператор `continue` переходит к следующей итерации (заголовку) ближайшего охватывающего цикла. Например, в следующем примере `continue` перейдёт к следующей итерации `for` (ближайший охватывающий цикл), а `break` выйдет из цикла `while`:

```
for i in range(5):
    if i == 3:
        continue
    j = 0
    while j < 4:
        if j == 2:
            break
        j += 1
```

Внутри одного цикла могут быть записаны оба оператора, количество которых может быть произвольным.

3.14. Оператор for/else

Цикл for применяется для обхода элементов произвольной последовательности с выполнением тела цикла:

```
for i in range(3):
    print(i, end=' ')
# > 0 1 2
```

Переменной i последовательно присваиваются значения 0, 1, 2. Если переменной не существовало до начала цикла, она создаётся. При этом переменная продолжает существовать и после завершения цикла, а значением переменной цикла будет последнее присвоенное значение, например, в предыдущем примере i после окончания цикла будет равно 2. В следующем примере будет напечатано число 6:

```
for i in range(2, 100, 2):
    if i % 3 == 0:
        break
print(i) # > 6
```

В качестве последовательности для обхода могут выступать списки, строки и диапазоны. С помощью for найдём сумму списка чисел:

```
sum = 0
for a in [1, 2, 3]:
    sum += a
print(sum) # > 6
```

Особенностью цикла for в Python является возможность использования блока else после for (else и for должны иметь одинаковый вертикальный отступ). Тело блока else выполняется, если в теле цикла не был выполнен break. Это позволяет разрабатывать циклы отказавшись от флагов.

Рассмотрим применение блока else на примере проверки существования элемента в списке. Сначала приведём решение этой задачи с помощью флагов.

```
a = 1
L = [0, 2, 4, 6]
found = False
for b in L:
    if a == b:
        found = True
        break
print('Yes' if found else 'No')
```

В этом примере выполняется обход элементов списка. Если искомый элемент встречается в списке, тогда цикл завершается при помощи break с предварительной установкой флага в True. По умолчанию значение flag равняется False, что позволяет корректно обрабатывать пустой список. В последней строке в зависимости от присутствия элемента в списке выводится 'Yes' или 'No'

Теперь рассмотрим реализацию решения той же задачи, но уже с применением блока else:

```
for b in L:
    if a == b:
```

```
        print('Yes')
        break
else:
    print('No')
```

В этом примере оператор `break` исполняется интерпретатором, только если искомый элемент обнаружен в списке. Другими словами, блок `else` будет выполнен при условии отсутствия элемента в списке. Стоит заметить, что `else` должен находиться под словом `for`. Если поместить весь блок `else` внутрь `for`, то блок `else` будет уже относиться к `if`.

В других языках подобного поведения можно добиться путём создания функции. В Python такой способ тоже возможен (может быть, даже удобнее использования тернарной операции):

```
def is_in(a, L):
    for b in L:
        if a == b:
            return True
    return False

print('Yes' if is_in(a, L) else 'No')
```

Функция `is_in` вернёт `False`, только если до этого не был выполнен оператор `return True`, иначе говоря, если не был найден элемент в списке.

Цикл `for` позволяет обходить произвольную последовательность, в частности, можно обойти список кортежей. Во время этого обхода в заголовке `for` можно воспользоваться присваиванием кортежей, включая распаковывание вложенных структур, и расширенным присваиванием последовательностей:

```
for a, b, c in [(1, 2, 3), (4, 5, 6)]:
    print(a, end=' ') # > 1 4

for a, (b, c) in [(1, (2, 3)), (4, (5, 6))]:
    print(b, c, end='; ') # > 2 3; 5 6;

for first, *middle, end in [(1, 2, 3, 4), (5, 6, 7)]:
    print(middle, end='; ') # > [2, 3]; [6];
```

Например, второй `for` эквивалентен:

```
for item in [(1, (2, 3)), (4, (5, 6))]:
    a, (b, c) = item
    print(b, c)
```

Как отмечено в разделе 3.9, подобные присваивания возможны благодаря тому, что присваивания переменным в заголовке `for` осуществляются по обычным правилам присваивания.

В Python можно создавать вложенные циклы с произвольной глубиной.

Такая форма часто используется с функциями `enumerate` и `zip`. Функция `enumerate(a)` принимает произвольную последовательность `a` и возвращает элементы последовательности и индексы, которые соответствуют этим элементам:

```
((0, a0), (1, a1), ..., (n-1, a_{n-1}))
```

Далее продемонстрирован пример использования `enumerate` в цикле `for`:

```
for i, a in enumerate('abc'):
    print(i, a, end='; ') # > 0 a; 1 b; 2 c;
```

Функция `zip` применяется для одновременного обхода нескольких последовательностей. Например, `zip([4, 5, 6], 'abc')` вернёт следующую последовательность:

```
((4, 'a'), (5, 'b'), (6, 'c'))
```

С помощью цикла `for` можно обойти эту последовательность, как в примере с `enumerate`. С помощью функции `zip` и `range` можно смоделировать функцию `enumerate`, применённую к последовательности с заранее известным количество элементов:

```
for i, a in zip(range(len(L)), L):
    # тело
```

В общем случае функция `zip` принимает n последовательностей и возвращает последовательность из кортежей размера n. Функция `zip` обходит последовательности, завершая работу при достижении конца самой короткой последовательности.

3.15. Операторы `while`/`else`

Цикл `while` даёт возможность написания универсальных циклов. Хотя `while` является более универсальной конструкцией, в Python предпочтительным является применение оператора `for` во всех случаях, где это возможно и целесообразно. Одна из причин выбора в пользу `for` состоит в том, что во многих случаях `for` выполняется быстрее `while`.

Пока условие `condition_expr` в заголовке `while` является истинным, выполняется тело цикла:

```
while condition_expr: # ← заголовок
    # тело цикла
```

В качестве `condition_expr` может выступать любое выражение, результат вычисления которого будет приведён к булевскому типу согласно правилам истинности (см. 3.4).

Как и оператор `for`, оператор `while` может иметь блок `else`, который является необязательным. Таким образом, общий формат цикла `while` следующий:

```
while cond_expr:
    # тело цикла
else:
    # тело else выполняется, если не было break
```

На примере проверки числа $n > 2$ на простоту рассмотрим применение `else`:

```
n = 11 # число, простоту которого необходимо проверить
i = 2
while i <= int(n**0.5) + 1: # итерация до ⌈√n⌉
    if n % i == 0:
        print('составное')
        break
    i += 1
else:
    print('простое')
```

Цикл `while` является циклом с предусловием, что означает, что сначала проверяется условие и лишь затем начинается выполнение тела цикла. Если условие не выполняется, тогда тело цикла не будет выполнено ни разу. В Python отсутствует оператор для написания цикла с постусловием, однако такой цикл всегда можно реализовать через цикл с предусловием:

```
while True:
    # тело цикла
    if condition_expr:
        break
```

3.16. Оператор `pass`

Все составные операторы (`if`/`elif`/`else`, `for`/`else`, `while`/`else`, `def`, `class`, `try`/`except`/`finally`) согласно синтаксису Python обязаны иметь тело. Чтобы не было синтаксической ошибки, для обозначения отсутствующего тела применяется оператор `pass`, например, в качестве временной заглушки:

```
if cond_expr:
    # тело, код которого разрабатывается в данный момент
else:
    pass # заглушка во избежание синтаксической ошибки
```

Кроме этого, можно воспользоваться многоточием (`...`). Стоит заметить, что многоточие не является оператором, а является объектом специального класса `ellipsis`:

```
def f(): ...

a = ... # a → Ellipsis
```

Таким образом, символ многоточия не является ключевым символом, а эффект от применения многоточия такой же, как в следующем примере:

```
def f(): 1
```

3.17. Резюме

Выражения состоят из операций и операндов (объектов). Выражение может быть вычислено и присвоено переменной. Для определения порядка вычисления сложных выражений используется приоритет операций. Операции с одинаковым приоритетом выполняются слева направо. Операторы, в свою очередь, составляются из выражений и выполняются по порядку. Операторы подразделяются на простые и составные. Простой оператор записывается в одну строки, а составной, имея заголовок и тело, записывается в несколько строк.

Операции можно разделить на несколько больших групп: арифметические, логические, побитовые, операции сравнения и проверки вхождения, создание последовательностей. Кроме этого, присутствуют следующие операции: вызов функции, индексация, тернарная операция.

В Python существуют различные операторы. Оператор присваивания в Python присваивает ссылку имени. Существуют различные формы присваивания: базовое, групповое, дополненное присваивания, присваивание кортежа/списка/последовательности, расширенная распаковка последовательности.

Оператор `if` позволяет выбрать путь исполнения программы в засимости от выполнения заданного условия. Операторы `for` и `while` применяются для создания циклов. Для выхода из цикла используется оператор `break`, а для перехода к следующей итерации применяется `continue`.

4. Динамическая типизация

В этой главе будет рассмотрена динамическая типизация, которая является важнейшим понятием в Python. Знание основ динамической типизации позволяет понять, каким образом создаются объекты и присваиваются значения. Кроме того, будут рассмотрены разделяемые ссылки и связанные с ними проблемы, о которых должен знать каждый программист на Python.

Динамическая типизация позволяет не объявлять переменные и присваивать переменным объекты разных типов:

```
a = 1
a = 'a'
```

Далее подробно рассматривается модель динамической типизации в Python. На первый взгляд, она может показаться сложной, однако на практике всё гораздо проще.

4.1. Имена, ссылки и объекты

В Python *имена* (*names*) отличаются от объектов. В следующем примере `a` является именем, а `1` является объектом:

```
a = 1
```

Имена и объекты связываются с помощью ссылок. На рисунке 2 показан результат выполненного кода в виде созданных имени, объекта и связывающей их ссылки. На этом рисунке имя находится слева, объект — справа, а ссылка представлена соединяющей их стрелкой.

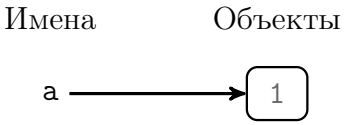

Рис. 2. Имена и объекты после выполнения `a = 1`

В Python вместо термина «переменная» был специально выбран термин «имя», чтобы подчеркнуть, что оно не хранит никакого значения.

Рассмотрим выполнение предыдущего примера по шагам. Оператор присваивания сначала выполняет правую часть и лишь затем левую. Поэтому сначала создаётся объект (см. рис. 3, а), затем создаётся имя (см. рис. 3, б), и только после этого создаётся ссылка (см. рис. 3, в).

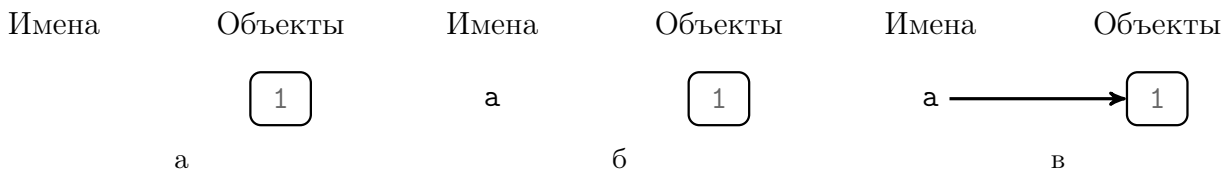

Рис. 3. Присваивание `a = 1` (по шагам): а — создание объекта, б — создание имени, в — связывание имени и объекта (создание ссылки)

Создание имени происходит в момент первого присваивания, а последующие присваивания удаляют ссылку у уже существующего имени и создают новую ссылку (см. рис. 4):

```
a = 1
a = 2
```

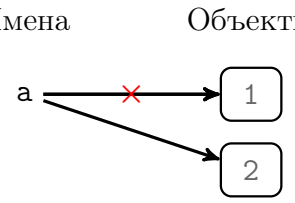

Рис. 4. Присваивание существующему имени `a = 2` с удалением старой ссылки

Для объектов и имён можно привести аналогию, взятую из [2]. Объект можно представить в виде коробки с содержимым, а имя в виде подписанной наклейки, которая наклеивается на коробку. Эту наклейку можно переклеивать на другие коробки, а на одну коробку можно приклеить несколько наклеек.

Хотя имена можно не объявлять заранее, перед применением они должны быть инициализированы, например, в следующем коде выбрасывается исключение при попытке вывести на экран значение несуществующего имени:

```
print(b) # ошибка
```

В выражении уже существующее имя заменяется объектом, на который ссылается. Таким образом, в следующем примере `b` будет ссылаться на тот же объект, на который ссылается и имя `a` (см. 5, а), а не на само имя `a` (см. 5, б):

```
a = 1
b = a
```

а б

Рис. 5. Присваивание имени уже существующего имени `b = a`: создаётся ссылка на объект (а), а не на имя (б)

Таким образом, через имя `b` нельзя изменить ссылку имени `a`.

Групповое присваивание, в свою очередь, также копирует ссылки (результат на рисунке 5, а):

```
a = b = 1
```

Во время присваивания уже существующему имени создаётся новый объект и ссылка, а не меняется объект, а старая ссылка удаляется (см. рис. 6):

```
a = b = 1
a = a + 1
```

Присваивание имени влияет только на это имя, но не на другие.

Применённый к имени оператор `del` (см. 3.7) удаляет имя и ссылку (см. рис. 7):

```
del a
```

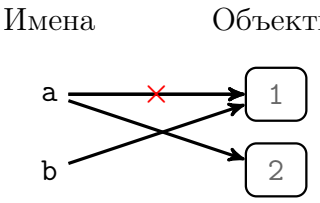

Рис. 6. Присваивание нового объекта создаёт ссылку, а не меняет объект

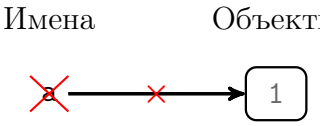

Рис. 7. Удаление имени приводит и к удалению ссылки

На рисунке 8 изображено логическое представление следующего списка:

```
L = [1, 2, 3, 4]
```

Рис. 8. Логическое представление списка

При работе со списками представления, продемонстрированного на рисунке 8, в большинстве случаев достаточно. Однако во внутренней реализации списки не хранят значения внутри, а хранят ссылки на объекты (см. рис. 9). Таким образом, ссылка может соединять не только имя с объектом, но и составной объект с его компонентом.

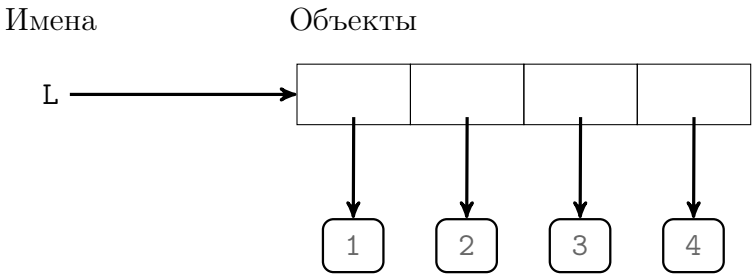

Рис. 9. Реализация списков

В выражении справа от = индексация списка приводит к замене объектом, на который указывает ссылка, поэтому результатом следующего кода будет картина, представленная на рисунке 10, а, а не на рисунке 10, б:

```
L = [1, 2, 3, 4]
b = L[0]
```

Поэтому через имя `b` нельзя изменить ссылку элемента списка.

Для изменения ссылки элемента списка необходимо слева от оператора = указать название списка и индекса (см. рис. 11):

```
L = [1, 2, 3, 4]
b = 5
L[0] = b
```

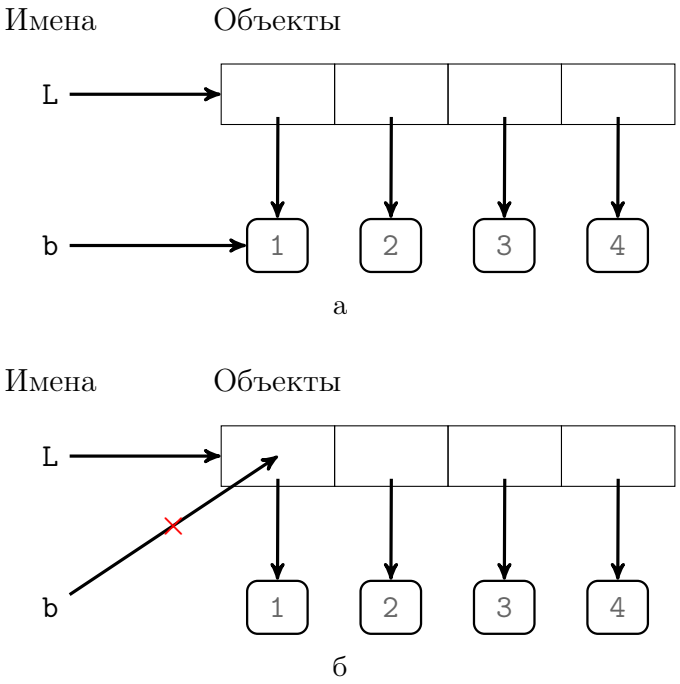

Рис. 10. Присваивание элемента списка `b = L[0]`: копируется ссылка на объект (а), а не на элемент списка (б)

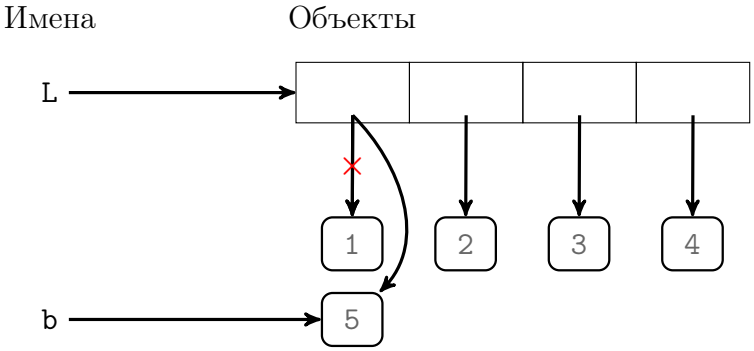

Рис. 11. Результат выполнения кода `L[0] = b`

Реализация объектов с переменными экземпляра реализованы схожим образом. Предположим, в классе `Point` присутствуют две переменные `x` и `y`, в таком случае экзепляр `p` этого класса с `x = 1` и `y = 2` будет храниться в памяти как показано на рисунке 12.

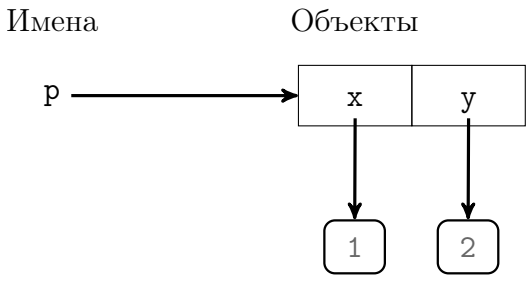

Рис. 12. Представление в памяти объекта `p` с `p.x=1` и `p.y=2`

Объекты классов и списки являются *составными объектами*, которые содержат ссылки на другие объекты, поэтому все примеры, приведённые для списков, будут работать аналогичным образом и для произвольных объектов.

Таким образом, присваивание не копирует объект, а всегда создаёт ссылку на объект, сохраняя её в имени или компоненте объекта. Подобное поведения для больших объектов повышает производительность программы.

Как было сказано в разделе 3.9, модель присваивания действует не только во время присваивания, но и в операторе `for` и во время передачи аргументов при вызове функций. Далее код выполняет обход элементов в цикле (см. 13), где цифры в кружочках обозначают номер итерации, на которой имя ссылалось на объект (в конце итерации имя `a` будет ссылаться на последний элемент итерации 3):

```
L = [1, 2, 3]
for a in L:
    pass
```

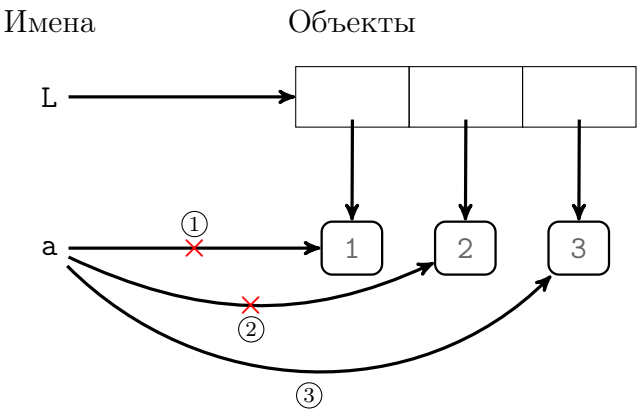

Рис. 13. Обход списка в `for`

На рисунке 14 продемонстрировано состояние имени и объектов во время вызова `func(b)` в следующем коде:

```
def func(a):
    pass
b = 1
func(b)
```

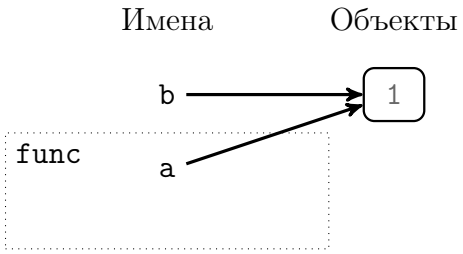

Рис. 14. Во время вызова функции копируется функции

4.2. Объекты

Объект — область памяти для хранения значений. Объекты либо задаются в виде литералов, либо создаются в результате выполнения выражений.

У каждого объекта есть свой идентификатор, значение которого можно получить при помощи функции `id`:

```
id(1) # вернёт идентификатор
```

Возвращаемое значение функцией `id` меняется от запуска к запуску.

Гарантируется, что каждый объект имеет уникальный идентификатор, который не меняется во время выполнения программы.

Данные могут храниться в виде объектов встроенных типов или пользовательских классов. В Python объект может быть либо изменяемым, либо неизменяемым.

Сборке мусора подвергаются именно объекты, а не переменные. Объект может быть удален лишь после того, как все ссылки на него были удалены. Но точный момент удаления объекта без ссылок зависит от реализации сборщика мусора в интерпретаторе Python.

4.3. Тождественность и равенство объектов

Для проверки тождестенности (identity) объектов (ссылаются ли имена на один и тот же объект или нет) применяются операторы `is` и `is not`:

```
a = 1
b = a
a is b # → True
```

Для проверки объектов на равенство (объекты содержат одинаковые значения) применяются операторы == и !=:

```
L = [1, 2, 3]
L2 = [1, 2, 3]
L is L2 # False
L == L2 # True
```

Операторы `is`/`is not` для проверки используют функцию `id`, а оператор `is` может быть реализован с помощью следующего кода:

```
id(a) == id(b)
```

Оператор ==/!= для объектов по умолчанию реализован через проверку `is`/`is not`. В классе можно изменить поведение операторов == и !=, но не операторов `is` и `is not`.

4.4. Тип данных

Динамическая типизация подразумевает отсутствие объявлений типов данных объектов:

```
a = 1
b = 'a'
```

Методы, которые можно вызвать у объекта, однозначно определяются типом этого объекта.

Тип данных в Python объявлять не нужно, интерпретатор определяет тип данных объекта автоматически. Тип данных «хранится» не в переменной, а в самом объекте, что показано на рисунке 15. Таким образом, интерпретатор всегда знает тип данных объекта, на который указывает имя. При виде литерала интерпретатор создаёт объект с сохранением соответствующего типа в самом объекте, а возвращаемый в результате выполнения выражения объект уже хранит в себе свой тип данных, например, в выражении `1 + 2` интерпретатор создаст два объекта `1` и `2` из литералов, записав в них тип данных `int`, а затем операция +, применённая к целым числам, «знает», что нужно вернуть результат типа `int`.

Для получения типа данных объекта можно воспользоваться функцией `type`:

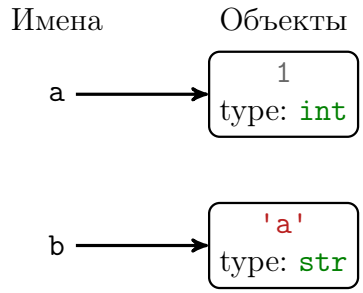

Рис. 15. Тип данных хранится в объекте, а не в имени

```
a = 1
print(type(a)) # → <class 'int'>
```

Чтобы проверить, является ли объект экземпляром определённого класса, используется встроенная функция `isinstance`:

```
isinstance(a, int)   # → True
isinstance(a, float) # → False
```

Эту функцию можно реализовать через:

```
type(a) == int # → True
```

Таким образом, каждый объект имеет уникальный идентификатор, тип и значение.

Как отмечено в книге [1], ссылки в Python являются аналогами указателей в C/C++, которые автоматически разыменовываются.

4.5. Сборка мусора

Сборка мусора осуществляется для удаления из памяти объектов, которые больше не потребуются, а освобождённая память может быть использована при создании новых объектов. Программисту не нужно заботиться об освобождении ненужной памяти, это происходит автоматически. Реализация сборки мусора зависит от используемого интерпретатора. В текущем разделе рассмотрим подробно сборку мусора в CPython.

Для каждого объекта интерпретатор выполняет подсчёт ссылок, храня количество ссылок на него в самом объекте. Когда количество ссылок становится равным нулю, *в тот же момент* объект удаляется из памяти. В следующем примере объект `1` удаляется во время выполнения второй строки (см. рис. 16):

```
a = 1
a = 2
```

С помощью функции `getref` из модуля `sys` можно получить количество ссылок на объект:

```
import sys
s = 'hello'
sys.getrefcount(s) # → 2
```

В предыдущем примере количество ссылок на объект `'hello'` равняется 2: первая ссылка идёт от имени `s`, вторая ссылка указывает из самой функции `sys.getrefcount`.

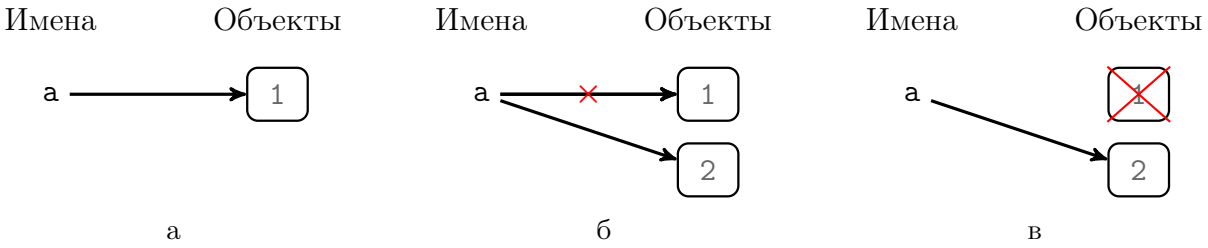

Рис. 16. Переприсваивание удаляет ссылку и объект: а — до переприсваивания, б — удаление ссылки, в — удаление объекта

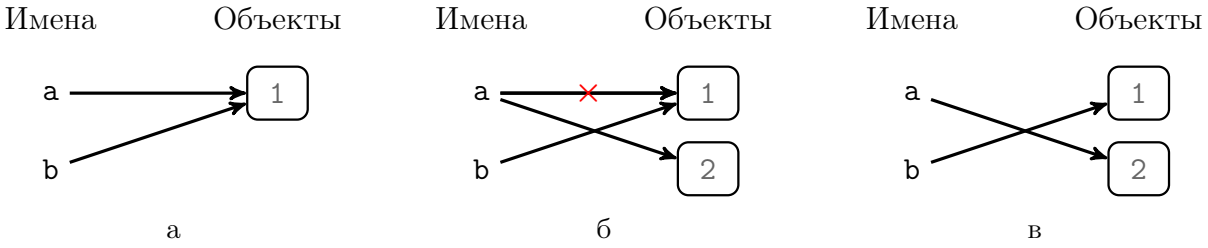

Рис. 17. При наличии другой ссылки переприсваивание не удаляет объект: а — до переприсваивания, б — переприсваивание с удалением ссылки, в — конечное состояние

Если после удаления ссылки на объект имеются другие ссылки, тогда объект не удаляется (см. рис. 17):

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

Удаление имени при помощи оператора `del` тоже может привести (при отсутствии другой ссылки) к удалению объекта:

```
a = 1
del a
```

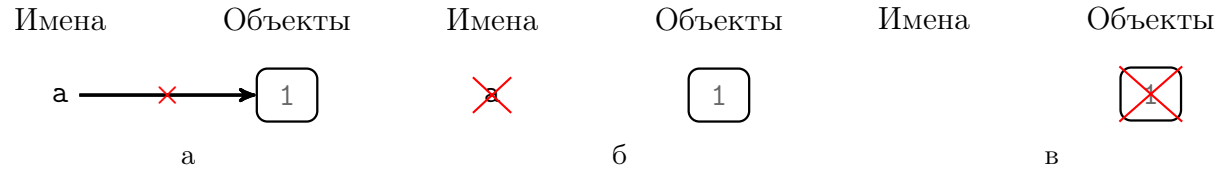

Рис. 18. Удаление имени (по шагам): а — удаление ссылки, б — удаление имени, в — удаление объекта

Список при удалении рассматривается как единый объект, что показано на рисунке 19 (кружочки показывают последовательность удаления имени, ссылок и объектов):

```
L = [1, 2, 3]
del L
```

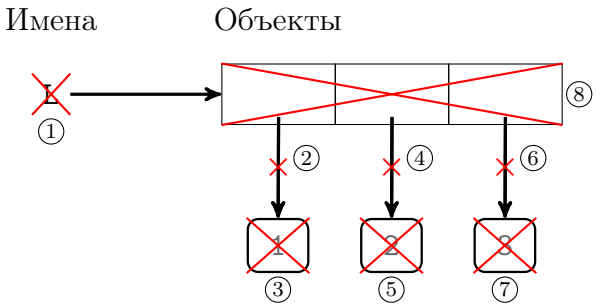

Рис. 19. Удаление списка

Циклические ссылки — ссылки, образующие цикл, если представить объекты в виде вершин направленного графа, а ссылки — в виде рёбер. Объекты, участвующие в цикле, никогда удалены не будут, потому что объект не удаляется, пока все ссылки на объект не будут удалены (см. рис. 20). Количество объектов в цикле может быть произвольным, в частности, одним:

```
L = [1]
L.append(L)
print(L) # > [1, [...]]
```

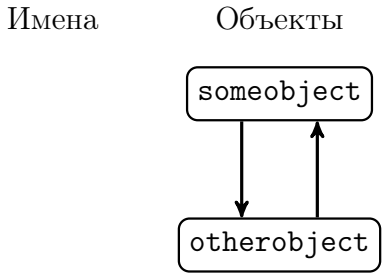

Рис. 20. Пример циклической ссылки

Программисту не нужно заботиться об удалении циклических ссылок, для этой цели применяется модуль `gc`, который в фоновом режиме отслеживает такие ссылки и удаляет объекты в этом цикле. По умолчанию, модуль `gc` постоянно работает, однако его можно отключить вручную, будучи уверенным в невозможности появления циклических ссылок.

4.6. Разделяемые ссылки

Разделяемые ссылки (*shared references*)— ситуация, когда есть несколько ссылок на один и тот же объект, например, как на рисунке 5, а. Возможно, более хорошим термином был бы «разделяемый объект», потому что разделяется именно объект, а не ссылки (имена и ссылки не могут быть разделяемыми). Разделяемые ссылки являются привычной ситуацией в Python, но, когда разделяемые ссылки *указывают на изменяемый объект*, это может привести к нежелательным последствиям: изменение объекта через имя может повлиять на другое имя. Рассмотрим следующий пример (см. рис. 21):

```
L = [1, 2, 3]
L2 = L
```

В этом примере изменение элемента списка (компонента списка) через имя `L` отразится и на имени `L2` (см. рис. 22):

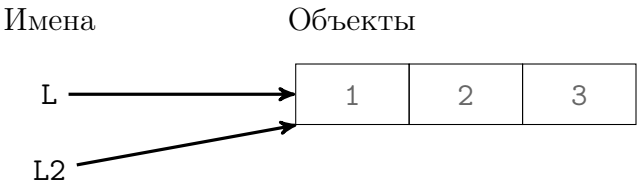

Рис. 21. Разделяемая ссылка на список

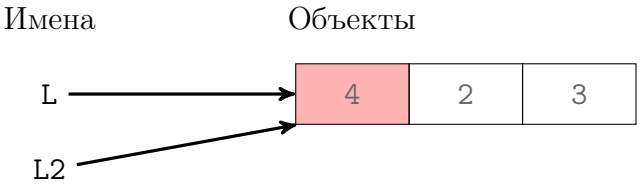

Рис. 22. Изменение списка через разделяемую ссылку

```
L[0] = 4 # L → [4, 2, 3]; L2 → [4, 2, 3]
```

Для обнаружения разделяемых ссылок можно воспользоваться рассмотренной ранее операцией `is`:

```
L is L2 # → True
```

Разделяемые ссылки возникают из-за копирования ссылок в контексте присваивания, например, при групповом присваивании копируется ссылка на один и тот же объект (результат как на рисунке 22):

```
L2 = L = [1, 2, 3]
L[0] = 4 # L → [4, 2, 3]; L2 → [4, 2, 3]
```

Кроме того, дополненное присваивание также можеть модифицировать изменяемый объект. Например, оператор += не будет создавать новый список, а добавит элементы в существующий:

```
L2 = L = [1, 2, 3]
L += [4] # L → [1, 2, 3, 4]; L2 → [1, 2, 3, 4]
```

Модель присваивания к тому же применяется и при передаче параметров в функцию (см. рис. 23):

```
def func(L2):
     L2[0] = 4 # L2 → [4, 2, 3]
L = [1, 2, 3]
func(L) # L → [4, 2, 3]
```

К тому же подобное случается при указании в заголовке функции значения по умолчанию. При объявлении функции создаётся один единственный объект. Во время вызова функции без указания значения происходит копирование ссылки на этот объект:

```
def func(L2=[]):
    L2.append(1)
    print(L2)
func() # > [1]
func() # > [1, 1]
```

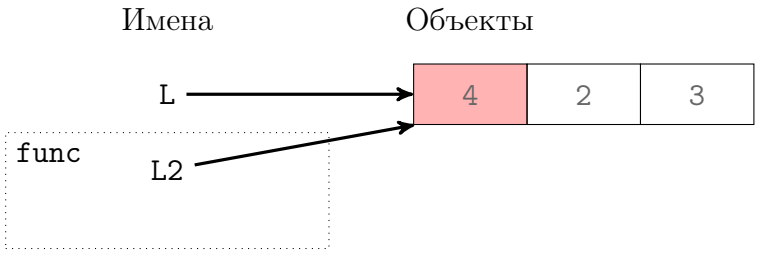

Рис. 23. Изменение списка через разделяемую ссылку

В этом примере имя L2 будет ссылаться на один и тот же объект, созданный ещё в момент объявления функции. Если же передать значение в эту функцию, тогда переданный список изменится:

```
L = [1, 2, 3]
func(L) # > [1, 2, 3, 1]; L → [1, 2, 3, 1]
```

Для предотвращения подобной ситуации можно воспользоваться следующим кодом, который использует особенность оператора or (см. 3.5):

```
def func(L2=None):
    L2 = L2 or []
    L2.append(1)
    print(L2)
func() # > [1]
func() # > [1]
L = [1, 2, 3]
func(L) # > [1, 2, 3, 1]; L → [1, 2, 3, 1]
```

В этом примере пустой список будет создаваться во время вызова функции, поэтому имя L2 будет ссылаться на разные списки.

Однако при изменении ссылки изменения объекта не происходит (см. рис. 24):

```
L = [1, 2, 3]
L2 = L
L = [4, 5, 6] # L → [4, 5, 6]; L → [1, 2, 3]
```

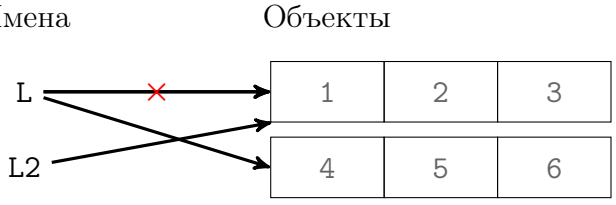

Рис. 24. Присваивание имени с разделяемой ссылкой ведёт к созданию новой ссылки без изменения объекта

Чтобы предотвратить изменение списка с разделяемой ссылкой, вместо присваивания списка его можно скопировать (см. рис. 25):

```
L = [1, 2, 3]
L2 = L.copy()
L[0] = 4 # L → [4, 2, 3]; L → [1, 2, 3]
```

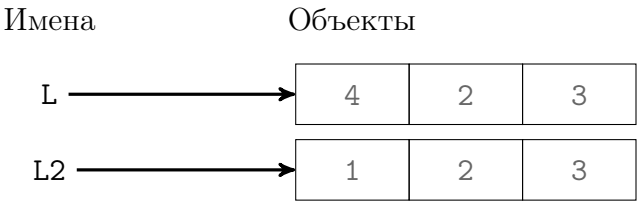

Рис. 25. Копирование списка и изменение скопированного списка

Метод `copy` присутствует в списке, однако может отсутствовать в объекте произвольного класса, поэтому для копирования такого объекта нужно воспользоваться функцией `copy` из модуля `copy`:

```
from copy import copy
L2 = copy(L)
```

Для решения проблемы, связанной с изменением списка внутри функции, можно воспользоваться двумя способами. В первом способе список копируется до вызова функции:

```
func(some_list.copy())
```

Этот способ используется при отсутствии уверенности в том, что вызываемая функция не изменит список. Во втором способе меняется определение самой функции, чтобы не навредить вызывающему коду:

```
def func(L):
    L = L.copy()
```

Но такое копирование не помогает, когда список содержит вложенный список (см. рис. 27, на рисунке список `[2]` продемонстрирован упрощённо):

```
L = [[1]]
L2 = L.copy()
L[0][0] = 2 # L → [[2]]; L2 → [[2]]
```

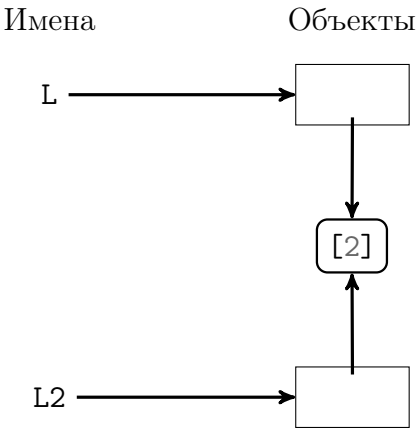

Рис. 26. Изменение списка списков после поверхностного копирования

Функция `copy` создаёт новый список и копирует ссылки, но не значения, поэтому список `[2]` является разделённым между двумя списками.

Далее представлен ещё один пример:

```
L = [1]
L2 = [L] # L2 → [[1]]
L.append(2) # L → [1, 2]; L2 → [[2]]
```

Рассмотрим часто встречающуюся ошибку при использовании операции повторения * на списке. После применения этой операции в полученном списке будут храниться ссылки на повторяемый(е) элемент(ы) (см. рис. 27):

```
L = [0]
L2 = L * 3 # → [0, 0, 0]
```

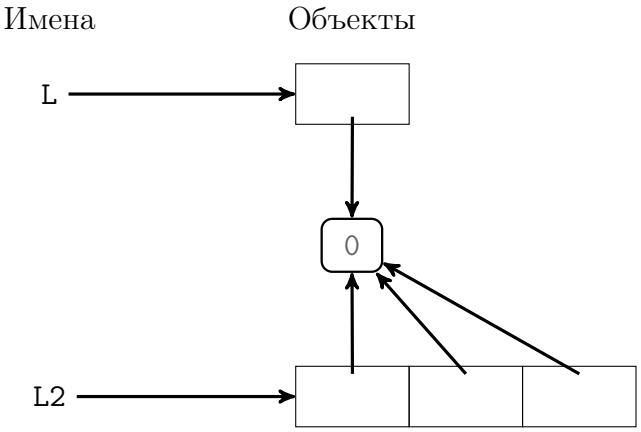

Рис. 27. Результат повторения списка `L2 = L * 3`

При повторении списка может возникнуть проблема с разделяемыми ссылками на изменяемый объект (см. рис. 28, где список `[1]` для упрощения представлен без ссылки и объекта):

```
L = [[0]]
L2 = L * 3  # L2 → [[0], [0], [0]]
L[0][0] = 1 # L2 → [[1], [1], [1]]
```

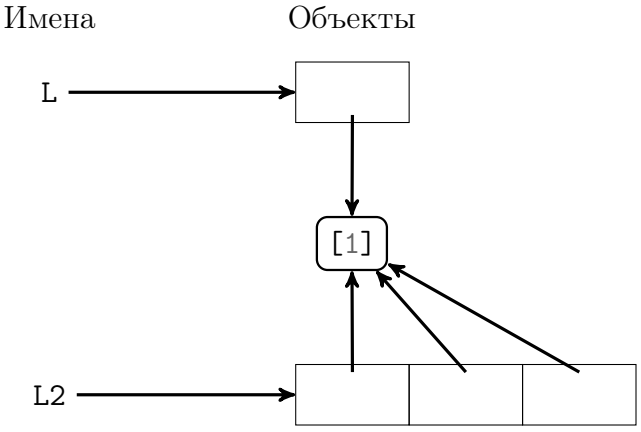

Рис. 28. Результат изменения повторённого списка со ссылкой на изменяемый объект

Проблемы с разделяемыми ссылками для объектов идентичны проблемам со ссылками на списки.

Для решения проблем с изменением вложенных объектов можно воспользоваться функцией `deepcopy` из модуля `copy`:

```
from copy import deepcopy
obj2 = deepcopy(complicated_obj)
```

Обобщая вышесказанное, для решения проблемы с разделяемыми ссылками необходимо выполнить копирование изменяемых данных с требуемым уровнем глубины копирования.

В Python всегда нужно помнить о разделяемых ссылках. Разделяемые ссылки не являются недостатком, они часто являются желательным поведением программы. Если стандартное поведение разделяемых ссылок не подходит, тогда есть возможность выполнить копирование объекта. Кроме того, копирование объектов по умолчанию (в частности, списков) во время присваивания не является действием по умолчанию из-за снижения производительности программы.

4.7. Модуль `copy`

Модуль `copy` применяется для копирования объектов и содержит две функции: `copy` и `deepcopy`.

Функция `copy` создаёт *поверхностную копию* (*shalow copy*) объекта. При выполнении поверхностной копии составного объекта сначала конструируется новый составной объект, а затем *копируются ссылки* на компоненты исходного объекта.

Функция `deepcopy`, в свою очередь, создаёт *глубокую копию* (*deep copy*). Во время копированиия составного объекта в самом начале создаётся составной объект, а уже потом *копируются вложенные объекты* (за исключением неизменяемых объектов).

Главная проблема глубокого копирования заключается в наличии циклических ссылок, которые могут привести к созданию бесконечного цикла. Эта проблема решается путём использования словаря внутри функции `deepcopy`, куда сохраняются уже сохранённые объекты.

4.8. Кэширование объектов

В CPython числа в диапазон $[-5, 256]$ и некоторые строки могут кэшироваться, то есть вместо создания будет возвращаться уже закэшированный объект:

```
1 is 1              # → True
'a' is 'a'          # → True
import sys
sys.getrefcount{1} # → 407
```

Из последней строки можно получить информацию о том, сколько ссылок на объект `1` существует во внутренних модулях Python.

Из-за кэширования таких объектов на них всегда будет указывать ссылка, поэтому эти объекты не будут подвергаться сборке мусора.

При этом ввиду неизменяемости чисел и строк не возникает проблем из-за разделяемых ссылок.

Однако нельзя полагаться на такое поведение от интерпретатора, так как оно может быть изменено в будущем.

4.9. Резюме

В этой главе была подробно рассмотрена динамическая типизация в Python, которая позволяет не объявлять типы объектов с помощью отслеживания типов этих объектов во время выполнения.

В Python объект и имя являются различными понятиями: объект представляет собой хранимые в памяти данные, а задаваемое программистом в виде строки имя используется для доступа к объектам. Имя и объект связываются между собой ссылкой, а составные объекты хранят ссылки на другие объекты.

Имя создаётся в момент первого присваивания и должно быть инициализировано до первого обращения к этому имени. Во время присваивания создаётся ссылка на объект.

В выражении имя/компонент объекта заменяется объектом, на который ссылается имя.

После создания новой ссылки старая (при существовании) удаляется. Когда количество ссылок на объект становится равным нулю, объект подвергается сборке мусора и удаляется. В CPython объект удаляется сразу, как только удаляется последняя ссылка на него. Для решения проблемы циклических ссылок применяется модуль `gc`.

Особую аккуратность следует соблюдать при работе с разделяемой ссылкой на изменяемый объект, потому что работающий с ней код может изменить этот объект. Модификация разделяемых объектов на изменяемые ссылки может воздействовать на другие части программы. Разделяемые ссылки могут быть созданы не только во время присваивания, но и в других контекстах присваивания. Во избежание проблем с разделяемыми ссылками можно выполнить поверхностное копирование или глубокое копирование.

Подобное детальное рассмотрение динамической типизации помогает разобраться в том, как работает Python. Тем не менее на практике, как правило, всё проще.

В целях упрощения изложения материала в этом пособии везде, кроме этой главы, под термином «переменная» подразумевается имя+ссылка+объект, например, «создаётся переменная типа `int`», «значение переменной равно `5`» и тому подобное. Мы будем возвращаться к терминам «имя» и «ссылка» для описания того, что происходит внутри интерпретатора.

5. Числа

5.1. Числовые типы

В данной главе рассматриваются типы данных для представления чисел, которые включают в себя целые числа (класс `int`), числа с плавающей точкой (класс `float`), комплексные числа (класс `complex`), десятичные числа с фиксированной точностью (класс `Decimal`), рациональные числа с числителем и знаменателем (класс `Fraction`). Последние два типа являются частью стандартной библиотеки, поэтому требуют подключения модуля.

Числа создаются либо с помощью задания литералов `1`, `1.2`, `1+2j` (комплексное число), либо в результате вызова встроенных функций (`int('1')` → `1`) и операторов (`1 + 2` → `3`).

В Python числовые типы могут хранить и положительные, и отрицательные числа. Числа в Python поддерживают обычные математические операции, например, сложение (`+`), умножение (`*`), и дополненные присваивания (например, `+=`). Отдельные числовые типы реализуют методы, специфичные для этих типов.

Очень важной особенностью чисел в Python, в отличие от других языков программирования, является *неизменяемость* чисел. Это обусловлено проблемой разделяемых ссылок. Рассмотрим рисунок 29, на котором два имени `a` и `b` ссылаются на один числовой объект.

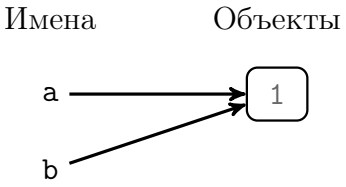

Рис. 29. Присваивание имени уже существующего имени `b = a`

В случае изменяемости целых чисел код `a += 2` привёл бы к изменению общего объекта-числа (см. рис. 30), что создавало бы множество проблем при решении различных задач с применением чисел:

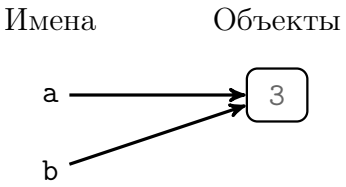

Рис. 30. Присваивание имени уже существующего имени `b = a`

Далее будут рассмотрены операции и методы, общие для всех числовых типов.

5.2. Операции над числами

Все числовые типы, кроме комплексных чисел, поддерживают операции, описанные в таблице 4.

Комплексные числа поддерживают все операторы из таблицы 4, исключая `//`, `%`, `divmod`. Кроме этого, комплексные числа нельзя привести к `int` и `float`.

Обычное деление `a/b` всегда возвращает вещественное число, даже если `a` и `b` являются целыми числами, и `a` делится нацело на `b`: `6/2` → `3.0`.

Таблица 4 — Операции, поддерживаемые числами

Операция	Описание
`a + b` и `a - b`	Сумма и разность `a` и `b`
`a * b` и `a / b`	Перемножение и деление `a` на `b`
`a // b`	Целочисленное деление `a` на `b`
`a % b`	Остаток от деления `a` на `b`
`-a` и `+a`	Отрицание `a` и возвращение неизменённого `a`
`abs(a)`	Абсолютное значение, или модуль числа `a`
`int(a)`	`a` конвертируется в `int`
`float(a)`	`a` конвертируется во `float`
`complex(re, im)`	Комплексное число с вещественной частью `re` и мнимой частью `im`
`c.conjugate()`	Сопряжённое число
`divmod(a, b)`	Пара значений `(a // b, a % b)`
`pow(a, b)` и `a**b`	Возведение `a` в степень `b`

Целочисленное деление `a//b` возвращает целое число. Если `a` не делится на `b` нацело, тогда возвращается целая часть, округлённая в сторону минус бесконечности: `3//2` → `1`, `-3//2` → `-2`, `3//2` → `-2`, `-3//-2` → `1`.

Деление на `0` возбуждает исключение `ZeroDivisionError`.

В Python поддерживается смешанная арифметика: можно складывать и делить принадлежащие разным типам данных числа (`1 + 2.0` → `3.0`, `3/2.0` → `1.5`). Во время смешанных операций более узкий тип данных расширяется до типа данных второго. Комплексное число является более широким, чем тип `float`, а `float` — чем тип `int`. Целые и вещественные числа поддерживают обычное и смешанное сравнение (`1 < 2` → `True`, `1.0 < 2.0` → `True`, `1.0 < 2` → `True`). Комплексные числа, в свою очередь, не поддерживают сравнение вовсе.

Метод `conjugate` доступен во всех классах числовых типов.

Для возведения в степень используется встроенная функция `pow(a, b)` или оператор `a**b`. Вызов `pow(0, 0)` возвращает `1`. В качестве `a` и `b` могут выступать как целые, так и вещественные числа: `pow(9, 0.5)` → `3.0` и `pow(2, 1/3)` → `1.2599210498948732`.

Типы `int`, `float` и `complex` могут принимать целые и вещественные числа, приводя к созданию соответствующего типа, например, `int(5.3)` → `5`, `float(1)` → `1.0`. Кроме этого, конструкторы могут сконвертировать строку, переданную в качестве аргумента: (`int('1')` → `1`, `int('1.2')` → `1.2`. Эта строка должна состоять из цифр от `0` до `9` и символов, для которых метод `isdecimal` (см. 6.8) возвращает `True`.

5.3. Сравнения чисел

Целые и вещественные могут сравниваться на равенство (`==`, `!=`) и для определения порядка (`<`, `<=`, `>`, `>=`). Все операции имеют одинаковый приоритет. Стоит напомнить, что в Python можно использовать сцепленные сравнения, например, `a > b > c` эквивалентно `a > b and b > c`.

5.4. Встроенные функции min, max, sum

Функции min и max, как следует из названия, возвращают минимум и максимум из чисел, переданных в качестве параметров. Количество аргументов может быть произвольное:

```
min(1, 2)       # → 1
max(1, 4, 3, 2) # → 4
```

Функция sum(numbers_list, start=0) возвращает сумму чисел из списка numbers_list с первоначальным значением суммы start. Если список пустой, возвращается значение start:

```
sum([]), sum([], 1) # → (0, 1)
sum([1, 2, 3])      # → 6
sum([1, 2, 3], 4)   # → 10
```

5.5. Булевский тип

Булевский тип представлен в Python классом bool, который является подклассом целого числа, поэтому bool поддерживает те же методы, что и int. В Python имеются булевские значения-константы True и False, которые представляют целые числа 1 и 0.

Любой объект может быть приведён к булевскому типу путём явного вызова конструктора bool (см. 3.4).

5.6. Целые числа

Особенностью целых чисел (и только их) в Python является неограниченная точность. Например, можно вычислить a = 2**1024, что во многих других языках программирования вызвало бы переполнение (overflow).

Целое число можно создать с помощью литералов. Десятичное число задаётся при помощи необязательного знака и десятичных цифр (подробнее в разделе 6.8), например, 1, -23, 456. Кроме этого, целое число можно задать в двоичной, восьмеричной и шестнадцатеричной системах счисления, указав префикс 0b, 0o и 0x соответственно: 0b11 → 3, 0o11 → 9 и 0x11 → 17.

Для создания целого числа из строки можно воспользоваться вызовом конструктора int(s, base=10), где s является строковым представлением числа в системе счисления base (по умолчанию равняется 10): int('11') → 11, int('11', 2) → 3, int('11', 8) → 9, int('11', 16) → 17.

Функция bin(a), oct(a), hex(a) возвращает a в двоичной, восьмеричной и шестнадцатеричной системах счисления в виде строки с идентифицирующим систему счисления префиксом: bin(3) → '0b11', oct(9) → '0o11', hex(17) → '0x11'.

5.7. Числа с плавающей точкой

Числа с плавающей точкой представляются классом float, которые реализованы через тип double в C. Информацию (в частности, точность) о представлении числа с плавающей точкой на машине, на которой запущена программа, можно получить с помощью sys.float_info. В результате будет получено

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
↪  min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15,
↪  mant_dig=53, epsilon=2.220446049250313e-16, radix=2,
↪  rounds=1).
```

В классе `float` присутствует метод `is_integer`, проверяющий, является ли число целым или нет:

```
(6/2).is_integer() # → True
(3.000001).is_integer() # → False
```

Задать вещественное число можно с помощью литералов, содержащих необязательный знак, цифры и точку, например, `1.0`, `-2.33`. Кроме этого, можно создать вещественное число, указав экспоненту со степенью, например, `0.3e2` → `30.0` и `1.33e-2` → `0.0133`.

5.8. Комплексные числа

Комплексные числа имеют вещественную и мнимую часть.

Комплексное число можно задать с помощью литерала, имеющего следующий шаблон `n+mj`, где `n` и `m` являются целыми или вещественными числами, задающими вещественную и мнимую часть. Например, `1+1j` (`1` перед `j` обязательна), `-1j`. Другим способом создания комплексного числа является конструктор `complex(real, imag=0)`.

Доступ к вещественной и мнимой части можно выполнить через свойства `real` и `imag` соответственно. Вещественной и мнимой частью могут быть только объекты типа `float`, поэтому при создании комплексного числа числовые литералы или переданные в качестве параметров числа конвертируются во `float`:

```
c = complex(1, 5)
c.real, c.imag         # → (1.0, 5.0)
type(c.real) is float # → True
type(c.imag) is float # → True
```

У целых и вещественных чисел также имеются свойства `real` и `imag`:

```
a, b = 1, 2.0
a.real, a.imag # → (1, 0)
b.real, b.imag # → (2.0, 0.0)
```

5.9. Побитовые операции над целыми числами

Побитовые операции могут выполняться только над целыми числами. Результат этих операций вычисляется в *дополнительном коде* (*two's complement*) с бесконечным количеством битов знака.

Приоритет побитовых операций меньше, чем приоритет числовых операций (подробнее см. раздел 3.2).

Операция `a | b` вычисляет побитовое ИЛИ, `a & b` — побитовое И, `a ^ b` — XOR, `~a` — инвертирование битов.

Для побитового сдвига используются операции `a << n` и `a >> n`, сдвигающие число на `n` бит влево и вправо соответственно. Эти операции эквиваленты `a*2**n` и `a/2**n`. Если передается отрицательное `n`, тогда возникает исключение `ValueError`.

5.10. Округление вещественных чисел

В Python присутствует большое количество функций для округления вещественного числа. Функция `round(a, n=0)` округляет `a` к ближайшему числу с `n` цифрами после запятой, из модуля `math` функция `floor` округляет до меньшего числа, `trunc` — к меньшему по модулю, `ceil` — к большему числу:

```
round(0.51), round(-0.51) # → (1, -1)
round(0.5), round(-0.5)   # → (0, 0)
round(0.49), round(-0.49) # → (0, 0)
round(0.557, 2)           # → 0.56
from math import floor, trunc, ceil
floor(0.51), floor(-0.51) # → (0, -1)
floor(0.5), floor(-0.5)   # → (0, -1)
floor(0.49), floor(-0.49) # → (0, -1)
trunc(0.51), trunc(-0.51) # → (0, 0)
trunc(0.5), trunc(-0.5)   # → (0, 0)
trunc(0.49), trunc(-0.49) # → (0, 0)
ceil(0.51), ceil(-0.51)   # → (1, 0)
ceil(0.5), ceil(-0.5)     # → (1, 0)
ceil(0.49), ceil(-0.49)   # → (1, 0)
```

5.11. Дополнительные методы целых чисел

Метод `bit_length()` возвращает количество бит, которое необходимо для хранения числа без знака:

```
print(bin(42), (42).bit_length())   # > 0b101010 6
print(bin(-42), (-42).bit_length()) # > -0b101010 6
```

Метод `to_bytes(length, byteorder, *, signed=False)` возвращает массив байтов, представляющий целое число. Первые два параметра являются обязательными. Параметр `length` указывает размер возвращаемого массива байтов. Если указанный размер меньше необходимого размера для представления числа, тогда выбрасывается исключение `OverflowError`. Если размер больше, тогда дополнительные байты заполняются нулями. Параметр `byteorder` задаёт порядок байтов: `'big'` или `'little'`. Если указано значение `'big'`, тогда самый значимый байт числа находится в начале массива байтов. Если указано значение `'little'`, тогда самый значимый байт числа находится в конце массива байтов. Чтобы узнать порядок байтов, который используется в системе, необходимо обратиться к `sys.byteorder`:

```
print((129).to_bytes(1, 'big'))  # > b'\81'
print((1029).to_bytes(1, 'big')) # OverflowError
print((0x0102).to_bytes(4, 'big'))
# > b'\x00\x00\x01\x02'
n = 0x01020304
n.to_bytes(4, 'big')    # → b'\x01\x02\x03\x04'
n.to_bytes(4, 'little') # → b'\x04\x03\x02\x01'
```

Ключевой параметр `signed` сигнализирует, используем ли мы отрицательные числа. Если параметр будет `False`, а число будет отрицательным, тогда возникает исключение `OverflowError`. Этот параметр по умолчанию равен `False`:

```
print((-127).to_bytes(2, 'big')) # OverflowError
print((-127).to_bytes(1, 'big', signed=True)) # > b'\x81'
```

Указывать каждый раз длину не удобно, поэтому можно реализовать свою функцию, заодно зафиксировав порядок байтов:

```
def to_bytes(n):
    bytes_length = (n.bit_length() + 7) // 8
    return n.to_bytes(bytes_length, 'big')
```

Результатом операции `(a + b - 1)//b` для целых `a` и `b` является округлённое вверх значение `a/b`. Таким образом, возвращается массив с минимально возможной длиной.

Обратной к предыдущей операции является статический метод `int.from_bytes(_bytes, byteorder, *, signed=False)`, который возвращает целое число, представимое переданным массивом байтов. Значение порядка `byteorder` должно совпадать с тем значением, которое было передано в метод `to_bytes`:

```
b = (1029).to_bytes(2, 'big')    # b → b'\x04\x05'
a = int.from_bytes(b, 'big')     # a → 1029
a = int.from_bytes(b, 'little') # a → 1284
```

Методы `to_bytes` и `from_bytes` могут использоваться для работы с числами как с массивами байтов: для записи в двоичный файл, для передачи по сети и других подобных задач.

5.12. Модули для работы числами

В Python существует модуль `math` для выполнения математических вычислений над числами модуль `random` для генерации случайных значений.

5.12.1. Модуль `math`

Модуль `math` содержит описанные в стандарте `C` математические функции, необходимые для математических вычислений. В модуле `cmath` находятся аналогичные функции для работы с комплексными числами.

Для начала перед любым использованием функций из этого модуля необходимо подключить модуль `math`:

```
import math
```

В модуле представлены математические константы `math.pi` → `3.141592653589793` и `math.e` → `2.718281828459045`, а также содержится функция `math.sqrt(n)`, вычисляющая квадратный корень числа `n`.

Для возведения числа в степень можно воспользоваться функцией `math.pow(a, b)`:

```
math.pow(3, 2) # → 9.0
math.pow(9, 1/2) # → 3.0
```

Функция `math.log(n[, base=math.e])` вычисляет логарифм `n` по базе `base`. Для нахождения факториала числа `n` следует применять функцию `math.factorial(n)`. Также в модуле `math` имеются тригонометрические функции `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `atan2()`, назначение которых явно вытекает из названия. Кроме того, в рассматриваемом модуле имеются функции `radians(deg)` и `degrees(rad)`, которые преобразуют координаты в радианы и в угловые соответственно.

5.12.2. Модуль `random`

Модуль `random` реализует генерацию псевдослучайных чисел, то есть похожих на случайные числа, но не являющихся таковыми. Этот модуль нельзя использовать для задач, связанных с безопасностью (для криптографии). Для криптографии можно воспользоваться модулем `secrets` или функцией `os.urandom(size)`, которая возвращает случайный набор `size` байтов и реализована через вызов функции операционной системы.

Для работы с функциями из модуля `random` его необходимо подключить:

```
import random
```

Основной функцией, на основе которой реализованы остальные функции, является функция `random.random()`, генерирующая случайное вещественное число из диапазона `[0.0, 1.0)`:

```
random.random() # -R-> 0.11085258654745067
```

В этом примере символ $\xrightarrow{R}$ означает, что возвращаемое функцией `random` число является псевдослучайным и при следующем запуске это число будет совершенно другим.

Для генерации `n` случайных байтов можно воспользоваться функцией `random.randbytes(n)`, которая впервые появилась в Python версии 3.9, например, `random.randbytes(10)` → `b'\x0f\xda\x8b;\xd1N\xd2cA\xd2'`.

Для получения целого числа из диапазона `[a, b]` применяется функция `random.randint(a, b)`. Функция `random.randrange(stop)` применяется для генерации случайного числа из диапазона `[0, stop)`, а функция `random.randrange(start, stop, step=1)` — из диапазона `[start, stop)` с шагом `step`, что аналогично случайному выбору числа из диапазона `range(start, stop, step)` без построения диапазона. Таким образом, `randint(a, b)` эквивалентно `randrange(a, b+1)`. Дополнительно в Python 3.9 появилась возможность генерации целого числа, состоящего из `k` битов, при помощи функции `random.getrandbits(k)`:

```
# возвращает случайное число в интервале [5, 10]
random.randint(5, 10)       # -R-> 8
# возвращает случайное число в интервале [0, 10)
random.randrange(10)        # -R-> 9
# возвращает случайное число в интервале [5, 10)
random.randrange(5, 10)     # -R-> 8
# возвращает случайное чётное число в интервале [0, 10)
random.randrange(0, 10, 2) # -R-> 4
random.getrandbits(50)      # -R-> 1026828266097004
```

В модуле `random` существуют отдельные функции для работы с последовательностями. Функция `random.shuffle(a)` случайным образом перемешивает последовательность на месте (последовательность изменяется):

```
L = [1, 2, 3, 4]
random.shuffle(L) # L -R-> [3, 2, 4, 1]
```

Символ $\xrightarrow{R}$ используется в примере выше для того, чтобы подчеркнуть, что объект, на который ссылается имя `L`, получен псевдослучайной функцией.

Функция `random.choice(seq)` даёт возможность получения случайного значения из последовательности `seq`. Если последовательность пустая выбрасывается исключение `IndexError`:

```
random.choice(L)         # -R-> 2
random.choice('abcdef') # -R-> 'b'
```

Кроме этого, функция `random.choices(seq, weights=None, *, cum_weights=None, k=1)` позволяет выбрать `k` элементов из последовательности `seq` *с повторениями*, возвращая список выбранных элементов. Для указания количества нужно задать ключевой параметр `k`:

```
L = [1, 2, 3, 4]
random.choices(L)       # -R-> [3]
random.choices(L, k=3) # -R-> [1, 1, 2]
```

Как видно, здесь элемент `1` выбирается с повторениями (два раза). Параметр `weights` (веса) должен быть задан в виде списка с длиной, равной `len(seq)`, а `weights[i]/sum(weights)` задаёт вероятность выбора `i`-го элемента последовательности:

```
random.choices(L, [1, 4, 2, 3], k=2) # -R-> [4, 2]
```

В этом примере общая сумма весов составляет `10`, поэтому шанс выбрать `1` — 10%, `2` — 40%, `1` — 20%, `1` — 30%. Параметр `cum_weights` задаёт кумулятивные веса. Кумулятивные веса для предыдущего примера задаются `[1, 4+1, 2+4+1, 3+2+4+1]`:

```
random.choices(L, cum_weights=[1, 5, 5, 10]) # -R-> [4]
```

В этом примере вероятность выбора значения `3` равняется нулю.

Функция `random.sample(seq, k)` возвращает список размера `k`, элементы которого выбраны случайным образом из `seq` *без повторения* (если элемент списка уже был выбран, он больше не рассматривается). Параметр `k` должен быть в диапазоне `[0, len(seq)]`:

```
random.sample(L, 4) # -R-> [3, 1, 4, 2]
```

В Python 3.9 эта функция может принимать ключевой параметр `counts`, позволяющий задать вероятность выбора элемента следующим образом. Применение `random.sample([1, 2], counts=[4, 1])` аналогично вызову `random.sample([1, 1, 1, 1, 2])`:

```
random.sample([1, 2], 1, counts=[4, 1]) # -R-> [1]
random.sample([1, 2], 4, counts=[4, 1]) # -R-> [1, 1, 2, 1]
```

Генерация псевдослучайных чисел является детерминированным процессом. Это означает, что если будет известно начальное состояние генератора, тогда будет известно число, которое будет сгенерировано. Для задания начального состояния используется функция `seed(a=None)`, принимающая в качестве параметра начальное состояние генератора, или сид:

```
random.seed(1)
random.random() # → 0.13436424411240122
random.random() # → 0.8474337369372327
random.seed(1)
random.random() # → 0.13436424411240122
random.random() # → 0.8474337369372327
```

В этом примере мы устанавливаем начальное значение заново, после чего генератор выдаёт предыдущее значение. При этом получаемые значения будут всегда одни и те же, что подчеркнуто обозначением $\rightarrow$ вместо $\xrightarrow{R}$. Если не указывать сид, тогда по умолчанию значением сида будет задано текущее системное время, что даёт небольшой эффект случайности. Установка собственного значения сида может применяться для тестирования приложений, использующих генерацию случайных чисел. В качестве значения сида могут выступать объекты классов `int`, `float`, `str` и `bytes` (массива байтов).

Кроме задания первоначального состояния, можно запомнить и задать текущее состояние генератора при помощи функций `random.getstate()` и `random.setstate()` соответственно:

```
random.seed(1)
random.random() # → 0.13436424411240122
state = random.getstate()
random.random() # → 0.8474337369372327
random.setstate(state)
random.random() # → 0.8474337369372327
```

Состояние генератора является общим для всего приложения. В некоторых ситуациях это является нежелательным, поэтому существует класс `random.Random`, который инкапсулирует в себе состояние генератора. Конструктор класса `random.Random` принимает параметр `seed=None`, указывающий начальное значение объекта генератора. Создаваемый таким способом объект имеет методы, аналогичные описанным выше функциям, с тем же названием:

```
r1, r2 = random.Random(1), random.Random()
r1.random()              # → 0.13436424411240122
r2.randint(5, 10)        # -R-> 5
r1.choice([1, 2, 3, 4]) # → 1
```

В этом разделе были рассмотрены функции, возвращающие числа с равномерным распределением, в модуле `random` также имеются функции для получения чисел, обладающих другим распределением.

5.13. Классы `Decimal` и `Fraction`

При работе с вещественными числами возникают две проблемы. Первая проблема (ошибка представления) заключается в невозможности точно представить десятичное вещественное число с помощью двоичного представления `float`, что отражено в следующем примере:

```
0.1 + 0.1 + 0.1 == 0.3 # → False
0.1 + 0.1 + 0.1 - 0.3  # → 5.551115123125783e-17
```

Как видно из этого примера, результаты не соответствуют математическим. В дополнение к этому, ввиду конечного размера компьютерной памяти числа `float` не могут представить бесконечные числа, например, 0.(3). Описанная ошибка затрудняет тестирование функций, возвращающих вещественное число. При сравнении вещественных чисел `a` и `b` на равенство желательно использовать сравнение `abs(a - b) < delta` с заданием значения `delta`, допустимого для запущенной задачи.

Вторая проблема является следствием первой. При выполнении операций деления вещественного числа постепенно накапливается ошибка точности, что может серьёзно сказаться на конечном результате.

Для решения первой проблемы используется класс `Decimal`, а для второй — класс `Fraction`.

5.13.1. Класс `Decimal`

Для хранения десятичных чисел используется класс `Decimal`, представляющий собой числа с фиксированной точностью. Результат выполнения операций над объектами класса `Decimal` полностью подчиняются правилам математики (проблема конечности памяти остаётся). Данный класс необходимо импортировать из модуля `decimal`:

```
from decimal import Decimal
```

Для создания десятичных чисел используется конструктор, принимающий строку:

```
a = Decimal('0.1') + Decimal('0.1') + Decimal('0.1')
a == Decimal('0.3') # → True
```

В дополнение к этому способу десятичное число можно создать из целого и из кортежа:

```
a = Decimal(1)
print(a)                          # > Decimal('1')
b = Decimal((0, (3, 1, 4), -2)) # b → Decimal('3.14')
```

При инициализации десятичного числа кортежем первый элемент должен указывать знак числа (`0` для положительных, а `1` для отрицательных), второй элемент — кортеж из цифр и последний — значение экспоненты. Поэтому пример выше даёт число $+314 \cdot 10^{-2}$. Десятичные числа реализованы путём хранения знака числа, кортежа цифр и экспоненты.

Кроме этого, объект `Decimal` можно создать из вещественного, но при этом происходит потеря точности:

```
Decimal('0.1') == Decimal(0.1) # → False
```

Здесь интерпретатор сначала, обнаружив литерал вещественного числа, создаёт вещественное число с потерей точности, а затем создаёт десятичное число из вещественного.

Ввиду того, что десятичные числа имеют ограниченную (фиксированную) точность, объекты `Decimal` имеют три дополнительных значения: NaN (Not a Number) — `Decimal('NaN')`, плюс бесконечность — `Decimal('Infinity')` и минус бесконечность `Decimal('-Infinity')`.

Над десятичными числами можно выполнять те же операции, что и над рассмотренными ранее типами данных:

```
Decimal('0.1') + Decimal('0.1')   # → Decimal('0.2')
Decimal('0.2') - Decimal('0.1')   # → Decimal('0.1')
Decimal('0.1') * Decimal('0.1')   # → Decimal('0.01')
Decimal('3') / Decimal('10')      # → Decimal('0.3')
Decimal('0.45') % Decimal('0.12') # → Decimal('0.09')
```

Как видно из примера, десятичные числа автоматически увеличивают точность чисел, полученных в результате умножения и деления. Во время сложения и вычитания десятичных чисел точность не меняется.

Десятичные числа могут взаимодействовать с целыми числами, но не с вещественными:

```
Decimal('0.1') * 2   # → Decimal('0.2')
Decimal('0.1') * 0.1 # TypeError
```

Нули в конце чисел не отбрасываются

```
Decimal('0.10') + Decimal('0.20') # → Decimal('0.30')
Decimal('0.10') * Decimal('0.20') # → Decimal('0.0200')
```

Как было упомянуто ранее, десятичные числа имеют фиксированную точность, иначе следующий пример вызвал бы переполнение памяти:

```
Decimal(1) / 3
# → Decimal('0.33333333333333333333333333')
```

В следующем примере показывается получение глобального значения точности с последующей наглядной демонстрацией:

```
from decimal import getcontext
print(getcontext().prec) # > 28
a = Decimal(1) / 3 # a → 0.33333333333333333333333333
```

$\underbrace{}_{28}$

Точность относится не к дробной части числа, а *ко всем его цифрам*, то есть *точность* — максимальное количество хранимых цифр числа:

```
Decimal(100) / 3 # a → 33.33333333333333333333333333
```

$\underbrace{}_{28}$

Эту глобальную точность можно не только узнать, но и задать вручную:

```
getcontext().prec = 6
a = Decimal(1) / Decimal(3) # a → 0.333333
```

$\underbrace{}_{6}$

При этом можно создать числа, имеющую бóльшую точность нежели глобальная:

```
getcontext().prec = 3
a = Decimal('0.1234') # a → Decimal('0.1234')
```

Однако в результате выполнения операций возвращается число, имеющее глобальное значение точности. Это особенно важно ввиду возможной потери точности путём округления результата:

```
Decimal('0.1234') + Decimal(0)       # → Decimal('0.123')
Decimal('0.123') * Decimal('0.456') # → 0.0561
```

$\underbrace{}_{3}$

```
getcontext().prec = 28
Decimal('0.123') * Decimal('0.456') # → 0.056088
```

Важным методом класса `Decimal` является метод `quantize`, позволяющий отсечь лишние разряды, например, для вывода на экран. В этот метод нужно передать десятичное число, результатом метода будет десятичное число с количеством разрядов, равным количеству разрядов переданного числа. Рассмотрим действие этого метода на примере:

```
a = Decimal(1) / 3
print(a.quantize(Decimal('1.00000'))) # > 0.33333
print(a.quantize(Decimal('1.00')))    # > 0.33
```

Кроме описанных выше операций десятичные числа поддерживают сравнение между собой, вследствие чего можно найти минимум/максимум или отсортировать десятичные числа.

Дополнительно десятичные числа содержат собственную реализацию операций вычисления квадратного корня и ей подобных с помощью методов `sqrt`, `exp` и так далее:

```
a.sqrt() # → Decimal('0.57735026918962576450914878 05')
```

5.13.2. Класс `Fraction`

Класс `Fraction` из модуля `fractions` представляет собой рациональное число, представленное в виде пары числителя и знаменателя. Этот класс используется, когда необходимо выполнять большое количество операций деления над целыми числами, при этом избегая потери точности.

Для использования класса его сначала необходимо импортировать:

```
from fractions import Fraction
```

Для создания объекта `Fraction` можно воспользоваться одним из нескольких конструкторов. Во избежание потери точности лучше всего применить конструктор `Fraction(numerator=0, denominator=1)`, который принимает числитель (`numerator`) и знаменатель (`denominator`).

```
a = Fraction(-1, 3)
print(a) # > -1/3
```

Если знаменатель (параметр `denominator`) равен 0, выбрасывается исключение `ZeroDivisionError`.

Другим конструктором, предоставляющим возможность создания дроби без потери точности является конструктор, принимающий строку формата `'[sign] numerator [/ denominator]'`:

```
a = Fraction('-1/3') # a → Fraction(-1, 3)
```

Кроме этого, можно создать дробь из вещественного или десятичного числа, однако уже с потерей точности:

```
a = Fraction(1/3)
# a → Fraction(6004799503160661, 18014398509481984)
a == Fraction(1, 3) # → False
```

Для получения числителя и знаменателя можно получить обратившись по свойствам (переменная в объекте) `numerator` и `denominator`. Целые числа тоже имеют эти свойства:

```
f = Fraction(5, 10)
print(f.numerator, f.denominator)     # 1 2
print((5).numerator, (5).denominator) # 5 1
```

После создания дроби над объектом можно выполнять арифметические операции:

```
f = Fraction(1)
f / 3                  # → Fraction(1, 3)
1 / Fraction(3)        # → Fraction(1, 3)
Fraction(1, 2) + f / 3 # → Fraction(5, 6)
```

Конструкторы, принимающие `float`, выполняют так называемую *аппроксимацию*, или *приближение*, то есть нахождение максимально близкой к вещественному числу дроби. Для решения проблем с неточной аппроксимацией можно воспользоваться методом `limit_denominator(max_denominator=1000000)`, который выполняет аппроксимацию дробью, чей знаменатель не превосходит `max_denominator`:

```
a = Fraction(1/3).limit_denominator()
# a → Fraction(1, 3)
```

У классов `float` и `int` есть метод `as_integer_ratio()` (начиная с Python 3.8) для вычисления приближения дробью, в котором результатом будет кортеж из двух чисел:

```
(1/3).as_integer_ratio()
# → (6004799503160661, 18014398509481984)
(2).as_integer_ratio() # → (2, 1)
```

При использовании дробей в математических вычислениях необходимо следить за тем, чтобы во всех операциях деления применялись дроби, а также все промежуточные вычисленные значения были представлены дробями. А уже после нахождения решения поставленной задачи преобразовать результат при необходимости в вещественное число:

```
result = Fraction(1, 3)
float(result) # → 0.3333333333333333
```

5.14. Резюме

В Python работу с числами можно организовать с помощью целых чисел (`int`), чисел с плавающей точкой (`float`) и при помощи комплексных чисел (`complex`). Во избежание проблем с потерей точности при работе с числами `float` стоит применять десятичные числа (`Decimal`) и рациональные числа (`Fraction`). Числовые типы являются неизменяемыми. Целые числа имеют неограниченную точность.

Над числами можно выполнять различные математические операции, их можно сравнивать, использовать встроенный модуль `math` для математических вычислений. Модуль `random` позволяет получить псевдослучайные числа и выполнять операции над последовательностями.

Булевский тип представлен двумя константами `True` и `False`.

6. Строки

6.1. Основы строк

Строка — последовательность символов, представляющая текстовую информацию. Строки в Python реализуются классом `str` и являются последовательностью односимвольных строк, а типа `char` не существует. В отличие от C++ в Python строки не заканчиваются нулём (`\0`), вместо этого в строковом объекте хранится список символов и длина строки.

Строки в Python хранятся в Unicode (кодировка UTF-8), что позволяет хранить в строках символы большого количества алфавитов.

Важным свойством строк в Python является то, что они не изменяемы (как и в C#). Поэтому методы строк не изменяют их, а создают новые.

6.2. Создание строк

Литералы строк задаются при помощи одинарных кавычек или двойных кавычек, при этом оба способа создания строки являются полностью эквивалентными. Первый способ является предпочтительнее, потому что позволяет меньше набирать на клавиатуре:

```
'abc' == "abc" # → True
```

В этом пример был использован оператор ==, который будучи применённым к двум строкам проверяет их содержимое на равенство, то есть если два разных объекта представляют одинаковую строку, тогда оператор == вернёт `True`.

Пустая строка задаётся путём двух подряд идущих кавычек `''`.

Для задания длинной строки можно воспользоваться следующей особенностью Python: два строковых литерала, записанных рядом и разделённых только пробелами, автоматически объединяются, что позволяет удобно записывать длинные строки:

```
a = 'abc''123' # a → 'abc123'
b = (
    'abcd'
    '1234'
)              # b → 'abcd1234'
```

Управляющие символы (*escape characters*) задаются при помощи экранирования: `'\n'`, `'\t'`, `'\r'`. Кроме этого, необходимо экранировать кавычки при использовании в литералах, задаваемых соответствующими кавычками: `'\''`, `"\""`. Для вставки в строку обратного слэша его необходимо экранировать: `'\\'`.

В некоторых ситуациях полезно отключить экранирование символов, например, при указании пути к файлам. В этом случае помогают *неформатированные строки*, для задания которых перед открывающей кавычкой необходимо указать префикс `r`, поэтому эти строки также называют *r-строки*:

```
r'c:\newdir\text.txt' == 'c:\\newdir\\text.txt' # → True
```

Дополнительно с помощью тройных кавычек можно удобно задать *многострочные строки* (*multiline strings*), в которых перевод на новую строку заменяется на `'\n'`:

```
a = '''abc
123'''
a == 'abc\n123' # → True
```

Часто подобные строки используются в качестве псевдо многострочного комментария:

```
'''
some unnecessary code
or comment
'''
```

Этот код содержит многострочный строковый литерал, при выполнении которого создаётся строковый объект. Созданный объект никуда не присваивается. Таким образом, подобный код исполняется во время выполнения, а не удаляется интерпретатором, поэтому при использовании подобных псевдокомментариев необходимо соблюдать строгие правила Python в плане отступов. Однако подобный способ не рекомендуется использовать, вместо этого следует применять комментарии с помощью `#`.

Для создания строки из произвольного объекта необходимо передать его в конструктор, например, `str(1)` создаёт строку `'1'`. Вызов конструктора без параметров `str()` создаёт пустую строку `''`.

Функция `repr` для строк возвращает строку в кавычках с экранированными символами:

```
print(repr('123\n')) # > '123\n'
```

6.3. Базовые операции над строками

Конкатенация, или склейка, двух строк выполняется при помощи оператора `+`:

```
s1 = 'Hello '
s2 = 'World'
print(s1 + s2) # > Hello World
print(s2 + s1) # > WorldHello
```

Для повторения строки несколько раз используется оператор `*`:

```
s = 'Hello'
print(s*3) # > HelloHelloHello
```

Для понимания принципов работы оператора `*` нужно помнить, что умножение на n является сокращением $(n-1)$-кратного сложения.

В памяти компьютера символ представляются в виде числа — *кода символа*. Для получения кода символа необходимо воспользоваться функцией `ord`, для обратной операции (получение символа по коду) используется функция `chr`:

```
b = ord('a')  # b → 97
print(chr(b)) # > a
```

Аргумент функции `chr` должен находиться в пределах от `0` до `1_114_111` (`0x10FFFF`) включительно, иначе возникает исключение `ValueError`.

6.4. Операции над последовательностями

Для определения длины строки используется функция `len`:

```
len('Hello') # → 5
```

Как было упомянуто ранее, длина строки хранится в виде переменной экземпляра в объекте строки, поэтому извлечение длины строки выполняется быстро: без обхода всей строки.

Для проверки вхождения одной строки в другую (поиск подстроки) используется оператор `in`. Этот оператор возвращает `True`, когда первая строка полностью содержится во второй, и возвращает `False` в противном случае. Причём проверка осуществляется с учётом регистра:

```
'el' in 'Hello'  # → True
'hel' in 'Hello' # → False
```

Каждый символ в строке имеет свой индекс, что показано на рисунке 31.

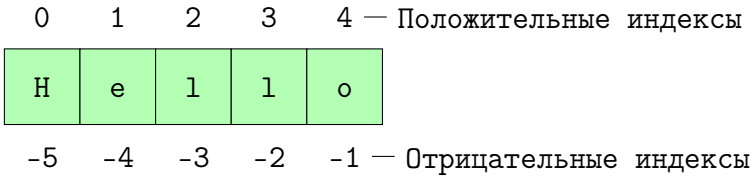

Рис. 31. Индексы в строке

Нумерация с начала строки (слева направо) начинается с `0`, а индексом последнего символа строки `s` является `len(s) - 1`. Нумерация с конца строки (справа налево) начинается с `-1`, уменьшаясь при движении к началу строки и достигая значения `-len(s)` на самом первом символе. Таким образом, символ с индексом `-i` будет иметь индекс `len(s)-i`, где $0 < i \leq n$.

Обращение к символу по индексу называется *индексацией*, которая осуществляется посредством квадратных скобок с указанием индекса. Однако при попытке обратиться к индексу, превышающему длину строки или меньшему, чем `-len(s)`, выбрасывается исключение:

```
a = 'Hello'
a[0]  # → 'H'
a[-1] # → 'o'
a[5]  # Исключение
a[-5] # → 'H'
a[-6] # Исключение
```

Ввиду того, что строки неизменяемы, нельзя поменять значение символа по индексу:

```
a = 'Hello'
a[0] = 'h' # Исключение
```

Следующий код демонстрирует, что в Python типа данных «символ» не существует:

```
s = 'a'
print(type(s[0])) # > <class 'str'>
s == s[0]         # → True
```

6.5. Срезы

В отдельную категорию операций над последовательностями можно отнести *срезы* (*slices*), которые являются особенностью языка Python. С помощью срезов можно получить последовательность символов. Для строки `s` срез `s[i:j]` возвращает подстроку

(копию строки) с индекса `i` до `j`, не включая его. При этом исходная строка не изменяется. Далее представлен пример среза, продемонстрированный на рисунке 32. Мы выбираем все символы со второго индекса (отмечены синим) до восьмого индекса (красным цветом), не включая его:

```
s = 'Hello World'
s2 = s[2:8] # s2 → 'llo Wo'
```

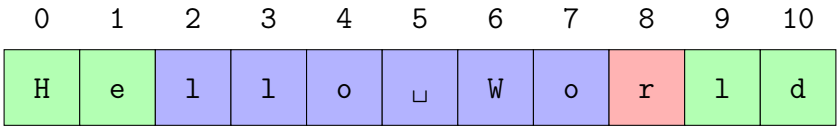

Рис. 32. Пример среза последовательных символов `s[2:8]`

По умолчанию `i` (начало среза) устанавливается в `0`, а `j` (конец среза) — в `len(s)`, поэтому можно не указывать начало и/или конец среза. Далее показаны некоторые примеры срезов с пояснениями:

```
s[:4]  # срез от начала до 4го символа не включительно
s[3:]  # срез с 3го символа до конца строки
s[:]   # срез всей строки, то есть создаётся копия
s[1:]  # копия строки кроме первого символа
s[:-1] # копия строки кроме последнего символа
```

Дополнительно в срезе можно указать шаг `s[i:j:step]`, то есть возвращается строка, состоящая из символов `s[i]`, `s[i+step]`, `s[i+2*step]` пока значение `i+n*step` не станет равным или не превзойдёт `j`. По умолчанию `step = 1`. Далее представлен пример, графическое представление которого показано на рисунке 33:

```
s[2:8:2] # → 'loW'
```

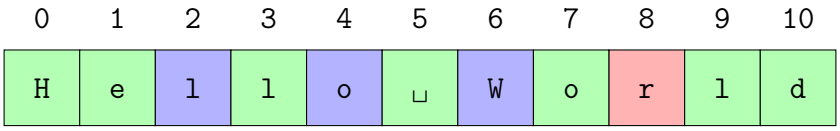

Рис. 33. Пример среза `s[2:8:2]` с шагом 2

Также можно использовать следующий синтаксис:

```
s[::3] # → 'HlWl'
```

Кроме этого, можно указать отрицательный шаг, тем самым задав обратный порядок обхода строки при срезе. В этом случае первым параметром указывается самый правый символ среза, а вторым — самый левый символ (не включительно):

```
s = 'World'
s[::-1]    # → 'dlroW'; разворот строки
s[-2:1:-1] # → 'lr'
```

В последней строке примера срез начинается со второго символа с конца `'l'`, проходится в обратном порядке и заканчивается, не доходя до символа `'d'`.

В отличие от индексации, срезы можно использовать, не беспокоясь о выходе за границу, потому что при выходе за границы значение `i` устанавливается в `0`, а `j` — в `len(s) - 1`. При этом в Python выход за границы нижней границы наступает при обращении к индексу, не превосходящему значение `-len(s)`, выход за границы верхней границы — к индексу, имеющему значение не менее `len(s)`. Например, для строки `'hello'` срез `s[-6:6]` будет аналогичен `s[0:5]` (или `s[-5:5]`), то есть будет возвращена копия строки `'hello'`.

6.6. Обход строки

Существует три способа обхода строк. Первый способ обхода заключается в переборе всех индексов строки от `0` до `len(s) - 1` с шагом `1` с последующим обращением по соответствующему индексу. Для этого необходимо воспользоваться функцией `range`:

```
s = 'Hello'
for i in range(len(s)):
    print(s[i], end=' ') # > H e l l o
```

В этом примере `i` проходит от `0` до `len(s)`, не включительно его, с шагом `1`. В теле цикла выполняется обращение к текущему символу и осуществляется вывод его на экран.

Меняя значения параметров, передаваемых в `range`, можно настраивать способ обхода строки, например, можно обойти только символы с чётным индексом или в обратном порядке (с помощью положительных или отрицательных индексов):

```
for i in range(0, len(s), 2):
    print(s[i], end=' ') # > H l o
for i in range(len(s) - 1, -1, -1):
    print(s[i], end=' ') # > o l l e H
for i in range(-1, -len(s) - 1, -1):
    print(s[i], end=' ') # > o l l e H
```

Этот способ является стандартным в C++, но в Python более подходящим способом является посимвольный обход строки. Для этого используется нижеприведённый код:

```
for ch in s:
    print(ch, end=' ') # > H e l l o
```

В этом способе в переменную `ch` последовательно помещается каждый символ, а в теле цикла выводится на экран. Этот способ обхода является наиболее предпочтительным, что отражено разработчиками Python меньшим количеством необходимого для набора текста.

Для обхода строки посимвольно в обратном порядке можно воспользоваться срезами:

```
for ch in s[::-1]:
    print(ch, end=' ') # > o l l e H
```

В отличие от аналогичного примера с функцией `range`, в этом примере создаётся перевёрнутая копия строки, что может стать проблемой, если первоначальная строка слишком большая.

Иногда кроме символа требуется получить ещё и индекс символа, в этом случае удобнее всего воспользоваться функцией `enumerate`, пример использования которой представлен ниже:

```
for i, ch in enumerate(s):
    print(i, ch, end=', ') # > 0 H, 1 e, 2 l, 3 l, 4 o,
```

Если нужно обойти посимвольно несколько строк одновременно, можно воспользоваться функцией `zip`:

```
for a, b in zip('hello', 'world'):
    print(a, b, end=', ') # → h w, e o, l r, l l, o d,
```

6.7. Сравнение строк

При сравнении строк на равенство с помощью `==` и `!=` проверяется содержимое двух строк с предварительным сравнением ссылок.

Операторы `<`, `<=`, `>` и `>=` сравнивают строки лексикографически (посимвольно), где при сравнении каждого символа используется его код:

```
'abc' <= 'abc' # → True
'abc' < 'abd'  # → True
```

Если строки разной длины, но одна строка является началом второй, тогда первая строка считается меньше второй:

```
'a' < 'ab' # → True
'ab' < 'a' # → False
'' < 'a'   # → True
```

Аналогом оператора `<` является следующая функция:

```
def less_than(s1, s2):
    for c1, c2 in zip(s1, s2): # обход строк одновременно
        if ord(c1) >= ord(c2):
            return False
    return len(s1) < len(s2)
```

Стоит напомнить, что `zip` проходит по двум последовательностям, останавливаясь при достижении конца самой короткой последовательности. Последняя строка нужна для случая, когда одна из строк короче другой.

Операторы сравнений могут использоваться для лексикографической сортировки строк. Кроме этого, сравнения можно использовать, в частности, чтобы проверить, что символ `c` является английской буквой в нижнем регистре: `'a' < c < 'z'`.

6.8. Проверка содержимого

Существует много методов, выполняющих проверку содержимого строки. Метод `isalpha()` возвращает `True`, если строка состоит исключительно из символов алфавита и при этом содержит хотя бы один символ, а `False` в противном случае. Метод `isdecimal()`, в отличие от предыдущего, возвращает `True`, только если строка содержит только арабские цифры и при этом содержит хотя бы один символ. Метод `isdecimal()` похож на предыдущий, если строка содержит любые цифры (в Unicode цифры могут обозначаться не только арабскими цифрами) и при этом содержит хотя бы один символ, возвращает `True`, иначе — `False`.

При передаче в конструктор `int` строки она может содержать только необязательный начальный символ знака и символы, для которых метод `isdecimal` возвращает `True`. Поэтому с применением этого метода можно реализовать проверку того, хранится ли в строке целое число или нет:

```
def is_int(s):
    if s[0] == '-' or s[0] == '+':
        s = s[1:]
    return s.isdecimal()
```

Обозначим через ALPHA множество символов, для которых `isalpha()` возвращает `True`. Для других методов будем использовать аналогичные обозначения. Для описанных выше множеств выполняется условие

$$\text{NUMERIC} \subset \text{DIGIT} \subset \text{DECIMAL},$$

где $\subset$ обозначает, что одно множество включено в другое.

Метод `isalnum` проверяет, что для каждого символа `c` строки выполняется `c.isalpha() or c.isnumeric()`, или в терминах множеств

$$\text{ALNUM} = \text{ALPHA} \cup \text{NUMERIC}\,.$$

где $\cup$ обозначает объединение множеств.

Метод `isascii()` возвращает `True`, если строка состоит исключительно из ASCII символов и при этом содержит хотя бы один символ, а `False` в обратном случае.

Метод `isidentifier()` возвращает `True`, если строка может быть использована в качестве имени переменной в языке Python, а `False` в противном случае.

Метод `isspace()`, если строка содержит только пробельные символы и при этом содержит хотя бы один символ, возвращает `True`, иначе — `False`. Метод `isprintable()` возвращает `True`, если строка состоит исключительно из печатных символов алфавита или является пустой, а `False` в противном случае.

Метод `islower()`/`isupper()` проверяет, находятся ли все символы строки в верхнем/нижнем регистре. Метод `istitle()` проверяет, что только первая буква каждого слова является заглавной.

6.9. Методы поиска

Оператор `in` позволяет проверить наличие подстроки в строке, но часто необходимо найти индекс в строке, где начинается искомая подстрока. Для этой задачи используется метод `find()`, который выполняет поиск подстроки с учётом регистра. Если строка не содержит подстроку, этот метод возвращает `-1`:

```
'Hello World'.find('Wor') # → 6
'Hello World'.find('wor') # → -1
```

Дополнительно метод `find` может принимать два необязательных аргумента `start` и `end`: `s.find(sub[, start[, end]])`, что, по сути, запускает поиск подстроки в срезе `s[start:end]`. Таким образом, параметр `end` задаёт индекс последнего элемента, не включая его:

```
'Hello World'.find('l', 5)    # → 9; start = 5
'Hello World'.find('o', 1, 3) # → -1; start = 1; end = 3
```

Метод `s.index(sub[, start[, end]])` также возвращает индекс вхождения подстроки, который, в отличие от `find`, выбрасывает исключение `ValueError` при отсутствии подстроки.

В дополнение к описанным в этом разделе методам существуют два строковых метода `s.rfind(sub[, start[, end]])` и `s.rindex(sub[, start[, end]])`, которые находят самый высокий индекс вхождения подстроки в срезе `s[start:end]`, то есть находят самый правый индекс:

```
'Hello World'.rfind('ld')  # → 9
'Hello World'.rindex('ld') # → 9
'Hello World'.rfind('he')  # → -1
'Hello World'.rindex('he') # Исключение ValueError
```

Для выполнения проверки на то, начинается/заканчивается ли строка на заданную подстроку, применяется метод `startswith(s2)`/`endswith(s2)`:

```
'hello'.startswith('h') # → True
'hello'.endswith('o')   # → True
```

6.10. Форматирование строк

Форматирование строк позволяет конструировать строки, используя значения переменных. Для форматирования строк в Python применяется метод `format`.

Самый простой способ форматирования — передача аргументов для форматирования по порядку. Рассмотрим на примере:

```
x = 42
'{} = {}'.format('x', x) # → 'x = 42'
```

В этом примере вместо первых фигурных скобок вставляется первый аргумент (`'x'`), вместо вторых — значение переменной `x`. Стоит заметить, что значение переменной автоматически преобразуется в строку.

В фигурных скобках можно указать позицию аргумента, который будет вставлен вместо фигурных скобок:

```
'{1} = {0}'.format(x, 'x') # → 'x = 42'
```

Таким образом, `'{} {}'.format(1, 2)` эквивалентно `'{0} {1}'.format(1, 2)`. Указание позиции позволяет использовать аргумент несколько раз:

```
'{0} or not {0}'.format('to be') # → 'to be or not to be'
```

Кроме этого, при форматировании можно извлечь значение по индексу или свойство:

```
a = [4, 5, 8]
'a[1] = {0[1]}'.format(a) # → 'a[1] = 5'
from fractions import Fraction
a = Fraction(2, 3)
'numerator = {0.numerator}'.format(a) # → 'numerator = 2'
```

Однако нельзя вызвать метод внутри фигурных скобок: код `'{0.bit_length()}'.format(42)` выбросит исключение.

Помимо позиционной передачи аргументов имеется возможность передать аргументы по ключевым словам:

```
y = 23
'{var} = {value}'.format(var='y', value=y) # → 'y = 23'
```

Чтобы экранировать фигурные скобки, их необходимо продублировать `{{` и `}}`.

В Python 3.6 появилась более простая и удобная операция для форматирования, называемая *интерполяция* строк (*literal string interpolation*). Строки задаются префиксом `f`, поэтому они часто называются `f`-строками. Эти строки хранят в себе выражения языка Python, которые выполняются во время исполнения программы:

```
print(f'x = {x}') # > x = 42
```

Как видно из примера, интерполяция позволяет получить результат, аналогичный форматированию:

```
print('x = {}'.format(x)) # > x = 42
# или
print('x = ' + str(x))    # > x = 42
```

Синтаксис с префиксом `f` можно использовать совместно с префиксом `r`:

```
filename = input('Input filename:\n')
a = fr'c:\new dir\{filename}'
```

Новый синтаксис позволяет задать более сложные выражения:

```
print(f'x = {x}, bit length = {x.bit_length()}')
# > x = 42, bit length = 6
print(f'{x+1}') # > 43
```

Внутри фигурных скобок можно поместить любое выражение, допустимое в том участке кода, где использована строка.

Выражения в интерполируемых строках выполняются всегда слева направо, что может быть важно при наличии побочных эффектов.

Были рассмотрены разные способы задания форматирования у строк, далее будут описаны различные характеристики форматирования (*format specifications*), которые, например, позволяют задать количество знаков после запятой для вещественных чисел:

```
'{:.3f}'.format(1/3) # → '0.333'
```

Все передаваемые в качестве параметров объекты преобразуются в строку при выводе с помощью вызова `str`, однако иногда бывает нужно вывести объект с помощью вызова `repr`. Для этого необходимо указать `'!r'`:

```
s = '{!r}'.format('123\n') # s → "'123\\n'"
print(s)                   # > '123\n'
```

Для настройки характеристик форматирования используется следующая форма, которую нужно указать после двоеточия и которая должна следовать после параметра `'!r'` при его наличии.

```
[[fill]align][sign][#][0][width][grouping][.prec][type]
```

Характеристики форматирования можно использовать как для метода `format`, так и для интерполяции строк.

Начнём объяснение с параметра `type`. Для целых чисел можно указать следующие значения: `'b'` (*b*inary) — бинарный формат, число выводится в двоичной системе счисления; `'c'` (*c*haracter) — конвертирует представленный кодом символ в соответствующий символ; `'d'` (*d*ecimal), `'o'` (*o*ctal), `'x'` или `'X'` (he*x*) — выводят числа в десятичной, восьмеричной, шестнадцатеричной (с использованием малых или заглавных английских букв от a до f) системе счисления; `'n'` — вывод числа в формате `'d'` с использованием разделителя символов согласно настройкам региона. По умолчанию при отсутствии параметра применяется `'d'`:

```
'{0:d} {0:b} {0:o} {0:X}'.format(15) # → '15 1111 17 F'
```

При использовании подобного форматирования не всегда можно отличить отформатированное число в десятичной системе счисления от остальных. Во избежание подобной проблемы можно воспользоваться параметром `'#'`, добавляющим префикс системы счисления:

```
'{0:#b} {0:#o} {0:#X}'.format(15) # → '0b1111 0o17 0XF'
```

Для вещественных и десятичных чисел в качестве параметра можно указать следующие значения. При использовании `'e'` или `'E'` числа выводятся с экспонентой (`e` или `E`) (по умолчанию точность равна 6), `'f'` или `'F'` — вывод числа как числа фиксированной точности (по умолчанию точность равна 6), `'g'` или `'G'` — в зависимости от числа вывод его в виде числа фиксированной точности или числа с экспонентой (по умолчанию точность равна 6), `'%'` — вывод числа как процента (число умножается на 100 и выводится в виде числа с фиксированной точностью и знаком %). По умолчанию (при отсутствии параметра) используется `'g'` с достаточной для представления вещественного числа точностью. При использовании этих настроек целые числа преобразуются в вещественные:

```
'{:f}'.format(1) # → '1.000000'
'{}'.format(1/3) # → '0.3333333333333333'
a = 1000/3
'{:e}'.format(a) # → '3.333333e+02'
'{:f}'.format(a) # → '333.333333'
'{:g}'.format(a) # → '333.333'
f'{0.123:.%}'    # → '12.3%'
```

Еще раз стоит отметить, что при выводе числа с помощью форматов `'e'` и `'f'` используется 6 разрядов после запятой, а для формата `'g'` — 6 разрядов для общего количества цифр.

Параметр `width` позволяет задать *минимальное* значение длины строки, полученной после форматирования. Если длина отформатированной строки меньше указанной, строка дополняется пробелами (числа дополняются пробелами слева, все остальные типы — справа), если больше — то длина строки становится такой, чтобы вместить всё содержимое строки:

```
'{:5}'.format(123)    # → '␣␣123'
'{:5}'.format(123456) # → '123456'
'{:5}'.format(1/3)    # → '0.3333333333333333'
```

Параметр `grouping` может принимать два значения `'_'` или `','`, обозначающее символ, с помощью которого в числах будут группироваться последовательности из трёх цифр:

```
'{:_}'.format(10000)  # → '10_000'
'{:8,}'.format(10000) # → '␣␣10,000'
```

Параметр `prec` (precision) для вещественных чисел в формате `'f'`/`'F'` или `'g'`/`'G'` позволяет указать максимальное количество цифр после запятой. Параметр `prec` для вещественных чисел в формате `'g'`/`'G'` позволяет указать максимальное количество цифр в целом: до запятой и после. По умолчанию применяется формат `'g'`. Для целых чисел этот параметр использовать нельзя:

```
'{:.3}'.format(1/3)    # → '0.333'
'{:.3}'.format(100/3)  # → '33.3'; в формате 'g'
'{:.3f}'.format(100/3) # → '33.333'
```

Для типов данных, отличных от чисел, параметр `prec` задаёт максимальное количество символов, используемых из строкового представления аргумента:

```
'{:.3}'.format('Hello World')  # → 'Hel'
'{:5.3}'.format('Hello World') # → 'Hel␣␣'
'{!s:.5}'.format([1, 2, 3])    # → '[1, 2'
```

Последняя строка является одним из немногих примеров необходимости явного указания параметра `'!s'` (перевод в строку при помощи вызова `str`). Таким образом, `'{:.n}'.format(s)` для строки `s` аналогичен `'{}'.format(s[:n])`.

Перед шириной можно указать параметр `'0'`, при использовании которого числа после знака дополняются нулями:

```
'{:07.2f}'.format(100/3)  # → '0033.33'
'{0:5} {0:05}'.format(1)  # → '    1 00001'
'{0:5} {0:05}'.format(-1) # → '   -1 -0001'
```

Параметр `'sign'` задаётся для числовых типов. Значение `'+'` сигнализирует о том, что символ знака нужно выводить как для положительных, так и для отрицательных. Значение `'-'` используется для вывода знака только у отрицательных чисел, что является значением по умолчанию. Кроме этого, можно передать пробел, тогда перед положительными числами будет выводиться пробел, а перед отрицательными — знак минуса:

```
'{:+} {:+}'.format(1, -2) # → '+1 -2'
'{:-} {:-}'.format(1, -2) # → '1 -2'
'{: } {: }'.format(1, -2) # → '␣1 -2'
```

Первые два параметра `'align'` и `'fill'` задают способ выравнивания и символ для заполнения лишнего пространства. При задании `'align'` как `'<'` значение выравнивается по левому краю с добавлением по необходимости пробелов (значение по умолчанию для большинства объектов), `'>'` — по правому краю (значение по умолчанию для чисел), `'='` — знак числа помещается слева, цифры числа справа, а между ними помещается заполнитель, `'^'` — выравнивание по середине с добавлением по необходимости слева и справа пробелов. По умолчанию используется пробел для выравнивания и заполнения строки до минимальной длины, но параметр `'fill'` позволяет задать символ, который будет использоваться в качестве заполнителя:

```
a, b = 1, -2
'{:5} {:5}'.format(a, b) # → '    1    -2'
f'{a:<5} {b:<5}'         # → '1     -2   '
f'{a:0<5} {b:0<5}'       # → '10000 -2000'
f'{a:=5} {b:=5}'         # → '    1 -   2'
f'{a:0=5} {b:0=5}'       # → '00001 -0002'
f'{a:^5} {b:^5}'         # → '  1    -2  '
f'{a:0^5} {b:0^5}'       # → '00100 0-200'
```

Очевидно, что формат `'{:05}'` аналогичен формату `'{:0=5}'`.

Характеристики форматирования, ко всему прочему, могут содержать значения в фигурных скобках, которые будут заменены переданным параметром:

```
s, width = 'hello', 7
'{:{}}'.format(s, width)     # → 'hello␣␣'
f'{s:{width}}'               # → 'hello␣␣'
a, width, prec = 100/3, 7, 3
f'{a:{width}.{prec}f}'       # → '␣33.333'
```

6.11. Выравнивание строк

В разделе о форматировании строк были описаны характеристики форматирования, позволяющие выравнивать строки. В Python присутствуют отдельные методы строк `ljust`, `rjust` и `center` для выравнивания строк. Эти методы принимают два аргумента: обязательный `width` и необязательный `fillchar=' '`. Параметр `width` задаёт длину, до которой строка дополняется символами `fillchar`. Если строка длиннее `width`, то строка не обрезается, а дополнения не происходит:

```
'hello'.ljust(7)      # → 'hello␣␣'
'hello'.rjust(7)      # → '␣␣hello'
'hello'.center(7)     # → '␣hello␣'
'hello'.ljust(7, '-') # → 'hello--'
'hello'.ljust(3)      # → 'hello'
```

Метод `s.zfill(width)` дополняет строку `s` слева нулями до ширины `width`. Этот метод аналогичен методу `s.rjust(width, '0')` за исключением того случая, когда в качестве `s` выступает заданное в виде строки отрицательное число:

```
'12345'.zfill(7)  # → '0012345'
'12345'.zfill(3)  # → '12345'
'-12345'.zfill(7) # → '-012345'
'hello'.zfill(7)  # → '00hello'
```

Для целого `n` и строки `s=str(n)` метод `s.zfill(width)` аналогичен форматированию `f'{s:0={width}}'`.

6.12. Преобразование регистров

Существуют различные методы для преобразования регистра. Для преобразования первого символа в верхний регистр, а всех остальных в нижний используется метод `capitalize()`, для преобразования первой буквы каждого слова в верхний регистр, а всех остальных в нижний — `title()`, для преобразования всех символов в нижний — `lower()`, всех символов в верхний — `upper()`, изменение нижних регистров в верхний и наоборот — `swapcase()`. Преобразуются только символы, имеющие соответствующий регистр, например, знаки пунктуации не изменяются. Далее представлен пример выполнения этих методов

```
s = 'hEllo, wOrLd'
s.capitalize()     # → 'Hello, world'
s.title()          # → 'Hello, World'
s.lower()          # → 'hello, world'
s.upper()          # → 'HELLO, WORLD'
s2 = s.swapcase()  # → 'HeLLO, WoRlD'
s == s2.swapcase() # → True
```

Эти методы могут применяться, в частности, для сравнения двух строк без учёта регистра:

```
'hello, world' == 'hEllo, wOrLd'.lower() # → True
```

6.13. Замена символов

Для замены в строке одной подстроки `s1` на другую `s2` используется метод `replace(s1, s2)`. При этом этот метод не изменяет исходную строку (строки неизменяемы), а возвращает новую строку:

```
s = 'hello'
s.replace('e', 'a') # → 'hallo'; s → 'hello'
```

В этот метод можно дополнительно передать число, обозначающее максимальное количество вхождений подстроки, которые будут заменены:

```
'hello'.replace('l', 'L', 1) # → 'heLlo'
```

В Python есть специальный метод `expandtabs(tabsize=8)` для замены всех символов табуляции на пробелы, при этом заменяя один символ табуляции на `tabsize` (по умолчанию 8) пробелов:

```
'\t\tabc'.expandtabs(2) # → '␣␣␣␣abc'
```

Этот метод можно реализовать вручную с помощью `s.replace('\t', ' '*tabsize)`.

6.14. Трансляция строк

Рассмотрим задачу изменения некоторых символов в строке на другие. Например, в строке `'abc'` нужно заменить `'a'` на `'1'`, `'c'` на `'3'`, `'e'` на `'5'`. Решением этой задачи будет `'1b3'`. Для подобных задач в Python используется метод `translate(table)`. Переданная таблица транслирования `table` должна быть словарём (подробнее про словари в главе 9), ключи которого представляют собой код заменяемого символа, а значения — код символа или строка (произвольной длины), на которую происходит замена. Если задать трансляцию символа в себя произойдёт исключение `LookupError`. Если в строке есть символы, коды которых отсутствуют в виде ключа в словаре, эти символы остаются в результирующей строке без изменений:

```
d = {ord('a'): ord('1'), ord('c'): ord('3'), ord('e'): ord('5')}
'abc'.translate(d) # → '1b3'
d = {ord('a'): '1', ord('c'): '3', ord('e'): '5'}
'abc'.translate(d) # → '1b3'
```

Если в качестве значения в словаре передаётся `None`, тогда соответствующий ключу символ будет удалён из строки:

```
d = {ord('a'): None, ord('c'): None, ord('e'): None}
'abc'.translate(d) # → 'b'
d = {ord('a'): '', ord('c'): '', ord('e'): ''}
'abc'.translate(d) # → 'b'
```

Использование метода `translate` описанным выше образом не очень удобно, поэтому существует статический метод `str.maketrans`, создающий словарь для замены. В этот метод можно передать словарь, в котором ключом может быть символ или код символа, значением — символ, код символа или `None`. Символы, переданные в качестве ключей, конвертируются в код соответствующего символа:

```
d = str.maketrans({'a': '1', 'c': '3', 'e': '5'})
# d → {97: '1', 99: '3', 101: '5'}
'abc'.translate(d) # → '1b3'
```

Кроме этого, можно передать две строки `s1` и `s2` одинакового размера. Каждый символ из `s1` будет заменён на находящийся на той же позиции символ из строки `s2`:

```
d = str.maketrans('ace', '123')
# d → {97: '1', 99: '3', 101: '5'}
'abc'.translate(d) # → '1b3'
```

В дополнение к этому можно передать третьим параметром строку `s3`. Каждый символ из этой строки будет заменяться на `None`, то есть будут удалены:

```
d = str.maketrans('a', '1', 'ce')
# d → {97: 49, 99: None, 101: None}
'abc'.translate(d) # → '1b'
```

6.15. Удаление символов

Иногда возникает задача удаления с начала или конца строки пробельных символов (`'\n'`, `'\t'`, `' '`), например, пользователь при вводе в конце строки ввёл несколько пробелов. Для решения этих задач служат методы `lstrip()`, `rstrip()`, `strip()`, которые удаляют пробельные символы слева, справа и с обеих сторон соответственно:

```
' \thello \n'.strip() # → 'hello'
' \thello \n'.lstrip() # → 'hello \n'
' \thello \n'.rstrip() # → ' \thello'
```

Описанные методы могут принимать строку, символы которой будут удаляться из соответствующей части строки. Из строки будут удаляться символы до тех пор, пока они встречаются в соответствующей части строки:

```
' \thello \n'.strip('\n') # → ' \thello '
# удаляются символы 'a', 'b', 'c' справа
'abc123bacacb'.rstrip('abc') # → 'abc123'
# удаляются символы 'a', 'b', 'c' справа и слева
'abc123bacacb'.strip('abc') # → '123'
```

С версии Python 3.9 появились методы `removeprefix(prefix)` и `removesuffix(suffix)`, которые удаляют строку `prefix` и `suffix` из начала и конца строки соответственно:

```
'abccab1cababc'.removesuffix('abc')      # → 'abccab1cab'
s = 'abccab1cababc'.removeprefix('abc') # → 'cab1cababc'
s.removesuffix('abc')                    # → 'cab1cab'
```

Таким образом, методы `strip` воспринимают аргументы как множество символов, а методы `removeprefix(prefix)` и `removesuffix(suffix)` — как подстроки.

С помощью `lstrip` можно, например, реализовать собственную функцию `bit_length` (подробнее в 5.11) и проверки, представляет ли строка число или нет:

```
def bit_length(n): # n - целое число
    s = bin(n)
    return len(s.lstrip('-0b'))

def is_int(s):
    return s.strip().lstrip('-+').isdecimal()
```

6.16. Методы разбиения и `join`

Метод `split(delim)` позволяет разбить строку по разделителю `delim`, получая список строк. Рассмотрим пример, представленный на рисунке 34 и в следующем коде:

```
lines = 'Hello World\n123'.split('\n')
# lines → ['Hello World', '123']
```

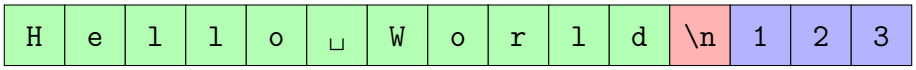

Рис. 34. Пример использования `split` по символу `'\n'`: символы до симола `'\n'` и после объединяются в строки и возвращаются в виде списка

В этом примере в строке `'Hello World\n123'` находятся все символы `'\n'` (на рисунке отмечен красным). Далее все символы между символами `'\n'` объединяются в одну строку (строки из зеленых и синих символов), создавая список строк.

Если два разделительных символа идут подряд, тогда в результирующий список попадут пустые строки:

```
'Hello\n\nWorld'.split('\n') # → ['Hello', '', 'World']
```

Если в метод `split` не передавать параметр, тогда строка разбивается по пробельным символам (`'\t'`, `'\n'`, `' '`, ...), при этом удаляются пустые строки:

```
words = 'Hello World\n\n123'.split()
# words → ['Hello', 'World', '123']
```

Метод `split` принимает необязательный параметр `maxsplit`, который задаёт количество разбиений:

```
s = 'Hello World\n\n123'.split(maxsplit=1)
# s → ['Hello', 'World\n\n123']
s = 'Hello World\n\n123'.split(maxsplit=2)
# s → ['Hello', 'World', '123']
s = 'Hello World\n\n123'.split('\n', maxsplit=1)
# s → ['Hello World', '\n123']
s = 'Hello World    123'.split(' ', maxsplit=2)
# s → ['Hello', 'World', '   123']
```

Метод `rsplit` аналогичен `split` за исключением того, что `rsplit` выполняет разбиение, проходя по строке от конца к началу:

```
s = 'Hello World\n\n123'.rsplit('\n', maxsplit=1)
# s → ['Hello World\n', '123']
```

Для разбиения строки на отдельные строки (lines) используется метод `splitlines(keepends=False)`, который аналогичен `split('\n')`. Если параметр `keepends` задаётся как `True`, тогда символ конца строки добавляется в конец строк:

```
s = 'Hello\nWorld\n123'.splitlines()
# s → ['Hello', 'World', '123']
s = 'Hello\nWorld\n123'.splitlines(True)
# s → ['Hello\n', 'World\n', '123']
```

Метод `join` выполняет обратную операцию. Для списка строк необходимо указать символ, который будет вставлен между строк:

```
'\n'.join(lines) # → 'Hello World\n123'
' '.join(words)  # → 'Hello World 123'
', '.join(words) # → 'Hello, World, 123'
```

Кроме описанных выше методов существуют два метода `partition(separator)` и `rpartition(separator)`, которые находят первое появление строки `separator` слева и справа соответственно, далее строка разбивается и возвращается кортеж, который состоит из строки до разделителя, самого разделителя и строки после разделителя. Если разделителя в строке нет, возвращается кортеж из исходной строки и двух пустых строк:

```
s = 'Hello World 123'.partition(' ')
# s → ('Hello', ' ', 'World 123')
s = 'Hello World 123'.rpartition(' ')
# s → ('Hello World', ' ', '123')
s = 'Hello World 123'.partition('\n')
# s → ('Hello World 123', '', '')
s = 'Hello World 123'.rpartition('\n')
# s → ('', '', 'Hello World 123')
```

6.17. Изменение строк

Сами строки являются неизменяемыми, но рассмотрим способ, позволяющий получить строку, в котором изменён один символ. Первый способ — использование списков: сначала строка преобразуется в список, изменяется один элемент списка с последующим получением строки:

```
s = 'hello'
list2 = list(s)     # list2 → ['h', 'e', 'l', 'l', 'o']
list2[1] = 'E'      # list2 → ['h', 'E', 'l', 'l', 'o']
s = ''.join(list2) # s → 'hEllo'
```

Вторым способом является использование срезов:

```
s = 'hello'
s = s[:1] + 'E' + s[2:] # s → ['h', 'E', 'l', 'l', 'o']
```

С помощью описанных выше способов можно изменить произвольное количество символов в строке.

6.18. Резюме

Строки (класс `str`) являются последовательностью символов и используются для хранения и обработки текстовой информации. Строки в Python являются неизменяемыми и хранятся в кодировке UTF-8.

Создание строк осуществляется либо заданием литералов, либо использованием конструктора `str`. Строки задаются в одинарных либо двойных кавычках. Многострочные строки задаются тройными кавычками.

Для определения длины строки применяется функция `len`.

Конкатенация строк при помощи оператора + позволяет получить их сцепление.

Оператор `in` позволяет осуществить проверку на вхождения подстроки, а индексация — получение символа по индексу. В Python присутствует возможность индексации по отрицательному индексу, осуществляющему отсчёт с конца строки. Срезы позволяют быстро получить подстроку, указывая начало, конец и необязательный шаг среза. С помощью цикла `for` можно обойти строку посимвольно.

В Python имеется мощная система форматирования строк, в частности, интерполяция, которая позволяет настроить способ отображения объектов в виде строки.

Преобразование регистров строк даёт возможность обрабатывать строки без учёта регистра.

«Изменения» строки можно добиться путём применения методов `replace`, `translate`, срезов или преобразования к списку.

Важным строковым методом является `split`, разбивающий строку по разделителю. С помощью этого метода можно, в частности, получить список слов. Метод `join` выполняет обратную операцию, выполняя конкатенацию элементов списка и позволяя, например, из списка слов получить предложение (без знаков препинания).

7. Списки и кортежи

7.1. Основы списков

Список (*list*), представляемый классом `list`, является *последовательностью* (*sequence*), которая позволяет хранить объекты произвольных типов. В представленном ниже коде создаётся список, хранящий число, строку и список (см. рис. 35):

```
L = [1, '2', [3]]
```

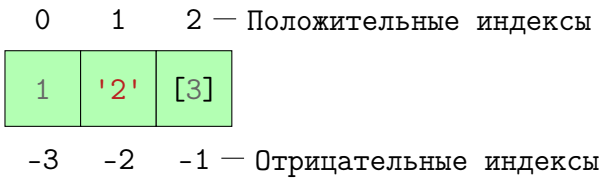

Рис. 35. Представление и индексы списка `[[1, '2', [3]]]`

Элементы списка пронумерованы слева направо, начиная с нуля, вследствие чего можно выполнять индексацию и другие операции над последовательностями.

Списки являются *изменяемыми* (*mutable*): можно как изменять элементы списка, так и изменять размер самого списка, добавляя или удаляя элементы списка:

```
L[0] = 4     # L → [4, '2', [3]]; изменение элемента
L.append(5) # L → [4, '2', [3], 5]
```

В этом примере использовался метод списка `append`, который добавляет в конец списка объект, передаваемый в качестве аргумента.

В отличие от многих языков программирования в Python очень легко вывести содержимое всего списка целиком. Для этого необходимо просто передать список в виде параметра в функцию `print`:

```
print(L) # > [4, '2', [3], 5]
```

При выводе интерпретатор обходит весь список, включая все вложенные объекты, и выводит содержимое в консоль.

7.2. Создание списков

Для создания списков существует несколько способов. Самый простой — использовать литералы, задавая собственноручно содержимое списка через запятую в квадратных скобках:

```
L = [] # пустой список
L2 = [123, 'a', 1.23, L]
```

В дополнение к этому способу существует возможность создания списка путём вызова конструктора класса `list`, передав в конструктор произвольную последовательность. Созданный список будет иметь тот же порядок элементов, что и переданная последовательность. Этот способ позволяет создавать списки во время выполнения программы:

```
L = list('hello')  # L → ['h', 'e', 'l', 'l', 'o']
L = list(range(5)) # L → [0, 1, 2, 3, 4]
```

При передаче строки (последовательности символов) в конструктор создаётся список, состоящий из отдельных символов переданной строки. Похожим образом в предыдущем примере выполняется вторая строка, потому что функция `range` возвращает последовательность целых чисел.

Кроме этого, можно скопировать существующий список с помощью метода `copy()`:

```
L = [1, 2, 3].copy() # L → [1, 2, 3]
```

7.3. Сравнение списков

Списки поддерживают различные операции сравнения. Наиболее употребительными операциями являются проверка равенств двух списков с помощью операторов == и !=. Эти операторы в первую очередь проверяют ссылки списков, если они равны, тогда оператор == возвращает `True`, а оператор != возвращает `False`. Оператор == проверяет, что списки имеют одинаковую длину и в одних и тех же индексах списка хранятся одинаковые значения (осуществляется проход по спискам с поэлементным сравнением), а второй оператор проверяет, что в списках не хранятся одинаковые значения:

```
L = [1, 2]
L2 = [1, 2]
# Сравниваются ссылки, содержимое не просматривается
L == L           # → True
# Содержимое просматривается
L == L2          # → True
L != ['1', '2'] # → True
```

Кроме этого, списки можно сравнивать с помощью операторов <, <=, >, >=. Эти операторы выполняют лексикографическое (поэлементное) сравнение, принцип работы которого похож на сравнение строк. Рассмотрим принцип работы оператора < (остальные операторы действуют аналогичным образом). Этот оператор поэлементно сравнивает два списка, применяя к сравниваемым элементам оператор <. Если элементы равны, осуществляется переход к сравнению следующих элементов. Если же элемент первого списка меньше, возвращается `True`, иначе — `False`. Ниже представлен пример сравнения списков:

```
[1, 'a'] < [1, 'b'] # → True
[1] < [1, 'b']      # → True
```

В этом примере в первой строке сначала сравниваются нулевые элементы, они являются одинаковыми, поэтому происходит сравнение следующих элементов. Ввиду `'a' < 'b'` результатом операции сравнения списков является `True`. Вторая строка возвращает `True` из-за того, что первый список короче второго.

7.4. Операции над последовательностями

Списки как и строки поддерживают операции над последовательностями. Для получения длины списка применяется функция `len`:

```
L = [1, 2, 3, 4]
len(L) # → 4
```

Значение длины списка хранится в переменной экземпляра списка, а функция `len` возвращает это значение.

Индексация выполняется с помощью квадратных скобок. Как и строки, списки поддерживают отрицательную индексацию:

```
L[0]      # → 1
a = L[-1] # a → 4
```

В Python при обращении по индексу, выходящему за границы списка, то есть при обращении к несуществующему элементу списка, выбрасывается исключение IndexError:

```
a = L[4]  # выбрасывается исключение IndexError
L[-5] = 0 # выбрасывается исключение IndexError
```

Срезы работают для списков тем же образом, что и для строк, однако возвращают список, при этом исходный список не меняется:

```
L[1:-1] # → [2, 3];       L → [1, 2, 3, 4]
L[::2]  # → [1, 3];       L → [1, 2, 3, 4]
L[::-1] # → [4, 3, 2, 1]; L → [1, 2, 3, 4]
```

В отличие от индексации срезы более лояльны к выходу за границы списка: во время создания срезов при выходе за границы списка исключение не выбрасывается. В этом случае граница начала среза устанавливается в начало списка, а граница конца среза — в конец списка:

```
L[2:10]  # → [3, 4]
L[-10:2] # → [1, 2]
```

В этом примере `L[2:10]` фактически заменяется на `L[2:len(L)]`, а `L[-10:2]` — на `L[0:2]` (или `L[-len(s):2]`).

В дополнение к этим методам списки поддерживают конкатенацию и повторение. Конкатенация дублирует первый список, затем в конец этого списка добавляется второй список, а результирующий список возвращается в качестве результата. Повторение позволяет выполнить многократную конкатенацию списка, создавая новый список:

```
L2 = L + [5, 6] # L2 → [1, 2, 3, 4, 5, 6] - конкатенация
L2 = L * 2      # L2 → [1, 2, 3, 4, 1, 2, 3, 4]
# L → [1, 2, 3, 4] - исходный список не изменился
```

Повторение часто используется для создания списка размера `n` для последующего использования:

```
L = [0]*5    # L → [0, 0, 0, 0, 0]
L = [None]*3 # L → [None, None, None]
L = []*5     # L → []
```

Как видно из последней строки, при повторении пустого списка возвращается пустой список.

Стоит отметить, что с обеих сторон от операции + должны находиться объекты одного типа, иначе произойдёт ошибка:

```
[1, 2, 3] + "4, 5, 6" # Ошибка
```

Для обхода списка используются те же способы, как и при обходе строк:

```
L = list('abc')
for i in range(len(L)):
    print(i, L[i], end=', ') # > 0 a, 1 b, 2 c,
for item in L:
    print(item, end=', ') # > a, b, c,
for i, item in enumerate(L):
    print(i, item, end=', ') # > 0 a, 1 b, 2 c,
```

7.5. Поиск элементов в списке

Для проверки, содержится ли заданный элемент в списке, используется оператор `in`, который является идиомой в Python для проверки наличия элемента в последовательности:

```
L = [1, 2, 3, 4]
1 in L   # → True
0 in L   # → False
'1' in L # → False
```

Для вычисления индекса элемента используется метод `index`, который возвращает индекс элемента в списке. Если такого элемента не существует, выбрасывается исключение `ValueError`:

```
L2 = ['a', 'b', 'c']
L2.index('b') # → 1
L2.index('f') # Исключение ValueError
```

Поэтому для корректной работы с методом `index` нужно либо вылавливать исключение (работа с исключениями будет рассмотрена позже), либо выполнить первоначальную проверку:

```
if 'f' in L2:
    print(L2.index('f')) # Исключение не выбрасывается
```

В последнем приимере в случае присутствия элемента в списке будут выполняться два прохода по списку до искомого элемента, что может плохо сказаться на производительности. Поэтому в качестве третьего способа можно реализовать свою функцию, в которой нужно вручную пройти по списку.

Также метод `index` принимает необязательные аргументы `start` и `end`, которые ограничивают границу поиска элемента:

```
[1, 2, 3, 4, 2].index(2)        # → 1
[1, 2, 3, 4, 2].index(2, 2)     # → 4
[1, 2, 3, 4, 2].index(2, 2, -1) # Исключение ValueError
```

С использованием этих аргументов можно пройтись по всем вхождениям искомого элемента в списке.

Для подсчёта количества вхождений элемента в список используется метод `count`:

```
L = [1, 2, 3, 1]
L.count(1) # → 2
L.count(5) # → 0
```

7.6. Добавление элементов в список

Списки, в отличие от строк, можно модифицировать *на месте* (*in place*), то есть изменить сам список без создания копии, улучшая производительность программы. Эта функциональность характерна только для списков, поэтому эти операции реализованы в виде методов, а не функций.

В нескольких следующих разделах описаны методы, позволяющие изменять список. Начнём рассмотрение с методов добавления элементов в список. Метод `append` добавляет элемент в конец списка, метод `extend` похож на предыдущий, однако добавляет не один

элемент, а элементы произвольной последовательности. Метод `insert` добавляет элемент в указанную позицию: первый аргумент — индекс, в который вставляется элемент, второй аргумент — сам элемент:

```
L = [1, 2]
L.append(3)       # L → [1, 2, 3]
L.append('45')    # L → [1, 2, 3, '45']
L.extend([6, 7]) # L → [1, 2, 3, '45', 6, 7]
# передаём строку - добавляется каждый символ
L.extend('89')    # L → [1, 2, 3, '45', 6, 7, '8', '9']
L.insert(2, 0)    # L → [1, 2, 0, 3, '45', 6, 7, '8', '9']
```

Эти методы меняют сам список и возвращают `None`, поэтому будет ошибкой присваивать результат выполнения этих методов переменной:

```
L = L.append(8) # L → None
```

7.7. Удаление элементов из списка

Далее рассмотрим методы удаления элементов из списка. Применение этих методов уменьшает размер списка.

Метод `pop(index=-1)` удаляет элемент из списка по индексу и возвращает удалённое значение. Если индекс не задан, удаляется последний элемент списка:

```
L = ['a', 'b', 'c', 'd']
a = L.pop()  # L → ['a', 'b', 'c']; a →'d'
a = L.pop(1) # L → ['a', 'c']; a →'b'
```

Кроме этого, можно применить оператор `del` для удаления одного элемента или целого среза. Этот оператор может применяться к различным изменяемым объектам, поэтому `del` был реализован в виде оператора, а не метода списка:

```
L = ['a', 'b', 'c', 'd', 'e', 'f']
del L[0]   # L → ['b', 'c', 'd', 'e', 'f']
del L[::2] # L → ['c', 'e']
```

Оператор `del` даёт возможность перечислить через запятую несколько удаляемых значений:

```
L = ['a', 'b', 'c', 'd', 'e', 'f']
del L[0], L[2] # L → ['b', 'c', 'e', 'f']
```

В этом примере сначала удаляется нулевой элемент списка, а затем у изменённого списка `['b', 'c', 'd', 'e', 'f']` удаляется второй символ.

Предыдущие способы удаляли элементы по индексу, а для удаления элемента из списка по значению используется метод `remove`. Этот метод удаляет *первое* вхождение элемента, при отсутствии элемента выбрасывается исключение `ValueError`, при этом `remove` ничего не возвращает:

```
L = ['a', 'b', 'c', 'd']
a = L.remove('b') # L → ['a', 'c', 'd']; a → None
L.remove('b')     # Исключение
```

Для очистки списка (удаления всех элементов) используется метод `clear`.

7.8. Изменение списка с помощью срезов

Помимо описанных ранее способов изменения списков на месте существует специфичный для Python способ — срезы. В этом способе используется конструкция, шаблон которой показан ниже:

```
L[i:j] = L2
```

Данная операция удаляет срез `L[i:j]` и вставляет на это место список `L2`. Пример использования изменения с применением срезов представлен ниже:

```
L = [1, 2, 3, 4, 5]
L[2:4] = [6, 7] # L → [1, 2, 6, 7, 5]
```

В этом примере заменяются два элемента `3` и `4` (со второго по четвёртый, не включая его) на `6` и `7`. Однако размеры удаляемой области и вставляемого списка не обязаны совпадать:

```
L = [1, 2, 3, 4, 5]
L[2:4] = [6, 7, 8, 9] # L → [1, 2, 6, 7, 8, 9, 5]
L[2:-1] = [3, 4]      # L → [1, 2, 3, 4, 5]
```

При использовании этого способа изменения списков есть два вырожденных случая: удаление среза без вставки элементов и вставка списка без удаления среза:

```
L = [1, 2, 3, 4, 5]
L[2:4] = []           # L → [1, 2, 5]
# вставка перед 0-м элементом - аналогично insert
L[:0] = [6]           # L → [6, 1, 2, 5]
# вставка в конец списка - аналогично append
L[len(L):] = [8]      # L → [6, 1, 2, 5, 8]
# вставка в конец списка - аналогично extend
L[len(L):] = [9, 10] # L → [6, 1, 2, 5, 8, 9, 10]
```

Как отмечено в комментариях кода, с помощью срезов можно реализовать все методы, изменяющие списки по индексам, но применение методов делает код более понятным и более простым в использовании.

При этом область удаления и область добавления могут перекрываться благодаря тому, что выражение справа от оператора = вычисляется до присваивания:

```
L = [1, 2, 3, 4, 5]
L[2:4] = L[3:] # L → [1, 2, 4, 5, 5]
```

Можно также при изменении значений передать шаг среза, однако в этом случае количество элементов слева и справа должно совпадать, иначе выбрасывается `ValueError`:

```
L = [1, 2, 3, 4, 5]
L[1::2] = [-2, -4] # L → [1, -2, 3, -4, 5]
L[::2] = [-1, -3]  # ValueError
```

В последней строке количество элементов среза слева равняется трём, а количество элементов списка справа — двум, поэтому происходит ошибка при попытке изменения списка.

7.9. Вложение списков и матрицы

Как было показано ранее, в списке может храниться объект любого типа, в том числе и другой список. Исходя из того, что в Python нет типа данных для представления матриц, вложение списков является единственным способом для хранения матриц без привлечения сторонних библиотек:

```
M = [[1, 2, 3],
     [4, 5, 6]]
```

При попытке получить элемент матрицы `M` через индексацию будет возвращён список — строка матрицы:

```
M[1] # → [4, 5, 6]
```

Таким образом, для получения элемента матрицы `M` необходимо произвести дополнительную индексацию по строке:

```
M[1][1] # → 5
```

Для изменения элементов матрицы необходимо присвоить значение по индексу:

```
M[1][1] = 7       # M → [[1, 2, 3], [4, 7, 6]]
M[0] = [8, 9, 0] # M → [[8, 9, 0], [4, 7, 6]]
```

Глубина вложения списков друг в друга практически неограничена, что позволяет легко хранить многомерные матрицы.

Стандартные инструменты для работы с матрицами удобны в использовании, но не являются эффективными, поэтому для ускорения операций с матрицами используется специальная библиотека NumPy, которая является фундаментом в других математических библиотеках.

7.10. Списковые включения

В Python существует специальная операция для простого создания списка путём применения выражения к произвольной последовательности. Данная идиома Python называется *списковым включением* (*list comprehension*). Далее представлен пример использования спискового включения для получения квадратов чисел из списка `L`:

```
L = [1, 2, 3, 4]
L2 = [a**2 for a in L] # L2 → [1, 4, 9, 16]
```

Списковое включение удобнее читать с ключевого слова `for`: «после создания пустого списка для каждого элемента `a` из списка `L` вычисляется значение `a**2` и добавляется в конец созданного списка». Таким образом, рассмотренный выше пример аналогичен следующему коду:

```
L2 = []
for a in L:
    L2.append(a**2)
# L2 → [1, 4, 9, 16]
```

Хотя оба примера функционально эквивалентны, использование спискового включения является более предпочтительным. Наличие в программе кода, похожего на последний пример, сигнализирует о том, что необходимо переписать код, используя списковое включение. При возникновении сложности с написанием спискового включения можно записать алгоритм в виде последнего примера, а затем преобразовать к синтаксису спискового включения.

Как было отмечено выше, списковое включение создаёт список, проходясь по произвольной последовательности, поэтому список можно создать из `range` и строки. Для этого эти последовательности нужно указать после `in`:

```
[x**2 for x in range(5)] # → [0, 1, 4, 9, 16]
s = 'abc'
[ord(c) for c in s]      # → [97, 98, 99]
[c*2 for c in s]         # → ['aa', 'bb', 'cc']
```

Списковые включения позволяют легко получить столбец матрицы:

```
M = [[1, 2, 3],
     [4, 5, 6]]
[row[2] for row in M] # → [3, 6]
```

В этом примере переменная `row` проходит по строкам матрицы, а в результирующий список мы сохраняем второй элемент этой строки. В итоге будет получен второй столбец матрицы `M`.

Выражение в списковом включении может быть произвольным, например, можно создать список списков или матрицу:

```
L = list(range(1, 4))
[[a, a**2] for a in L] # → [[1, 1], [2, 4], [3, 9]]
N = 4
# создаётся матрица M×N и заполняется нулями
L2 = [[0]*N for i in range(M)]
```

Списковые включения позволяют также производить фильтрацию. В следующем примере список фильтруется, сохраняя только положительные элементы:

```
L = [1, -2, 4, -5, 0]
L2 = [a for in L if a > 0] # L2 → [1, 4]
```

Условием фильтрации может быть произвольное выражение, допустимое в `if`.

Приведённый выше код фильтрации элементов списка аналогичен следующему коду:

```
L2 = []
for a in L:
    if a > 0:
        L2.append(a)
# L2 → [1, 4]
```

Фильтрацию и вычисление произвольных выражений можно совместить.

```
L = [1, -2, 4, -5, 0]
L2 = [a*2 for in L if a > 0] # L → [2, 8]
```

При этом сначала выполняется фильтрация, а лишь затем вычисляется выражение. В некоторых случаях это может быть очень важно:

```
L = [1/a for a in [0, 1, 2] if a] # L → [1.0, 0.5]
```

В этом примере `1/a` вычисляется, когда `a` не равняется нулю.

Так же как `for` может быть произвольной вложенности, списковые включения могут содержать произвольное количество `for` с необязательным `if`:

```
L = [-2, -1, 0, 1, 2]
L2 = [[x, y] for x in L if x % 2 == 0 for y in L if y > 0]
# L2 → [[-2, 1], [-2, 2], [0, 1], [0, 2], [2, 1], [2, 2]]
```

Для лучшего понимания последней конструкции преобразуем её в аналогичную конструкцию с применением `append`:

```
L2 = []
for x in L:
    if x % 2 == 0:
        for y in L:
            if y > 0:
                L2.append([x, y])
```

Рассмотрим пример реализации линеаризации (преобразования в список) двумерной матрицы:

```
M = [[1, 2], [3, 4]]
L = [a for row in M for a in row] # L → [1, 2, 3, 4]
```

Однако не стоит увлекаться сложными списковыми включениями, потому что они могут усложнить чтение кода.

7.11. Сортировка, max, min и разворот списка

Кроме перечисленных выше функций в Python имеются встроенные реализации сортировки, поиска минимума и максимума.

Метод `sort` позволяет отсортировать список на месте, то есть изменяется исходный список, что позволяет экономить память. Во время сортировки элементы списка сравниваются с помощью оператора <:

```
L = [5, 6, 1, 3, 4, 2]
L.sort() # L → [1, 2, 3, 4, 5, 6]
```

Если во время сортировки при сравнении возникает исключение, то операция сортировки прерывается с выбрасыванием исключения, что может оставить список в неполностью отсортированном виде:

```
[1, '2'].sort() # исключение
```

Метод `sort` ничего не возвращает для акцентирования внимания на том, что при вызове `sort` возникают побочные эффекты. Поэтому не нужно присваивать результат выполнения метода `sort` переменной списка. Это является частой ошибкой новичков в Python:

```
L = [5, 6, 1, 3, 4, 2]
L = L.sort() # L → None
```

Python во время сортировки сравнивает элементы списка с помощью оператора <, что позволяет сортировать список строк лексикографически:

```
L = ['Carol', 'Alice', 'Eve', 'Bob']
L.sort() # L → ['Alice', 'Bob', 'Carol', 'Eve']
```

В Python сортировка по умолчанию выполняется по возрастанию, для сортировки по убыванию необходимо установить ключевой аргумент `reverse` в `True`:

```
L = [5, 6, 1, 3, 4, 2]
L.sort(reverse=True) # L → [6, 5, 4, 3, 2, 1]
```

Благодаря поддержке оператора < в списках можно сортировать контейнеры, элементами которого являются сами списки. Рассмотрим сортировку списка сотрудников:

```
L = [['John', 42], ['Bob', 30], ['John', 50]]
L.sort() # L → [['Bob', 30], ['John', 42], ['John', 50]]
```

В этом примере сортируется список списков. При сортировке два списка сравниваются при помощи оператора <, поэтому сначала происходит сортировка по имени, а затем по возрасту.

В методе `sort` можно указать способ сортировки. Для этого необходимо передать указатель на функцию с помощью ключевого параметра `key`. Этот метод работает следующим образом. Для каждого элемента списка вызывается функция `key` (функция `key` будет вызываться для каждого элемента только один раз), в результате получая список ключей. Затем сортируется этот список, а когда меняются местами элементы в списке ключей, меняются и элементы исходного списка:

```
def mylower(s):
    return s.lower()
L = ['a', 'c', 'B']
L.sort(key=mylower) # L → ['a', 'B', 'c']
```

В этом примере мы сортируем список строк вне зависимости от регистра. Отметим, что название функции `mylower` указывается после «`key=`» без круглых скобок, потому что мы передаём указатель на функцию, а не вызываем её. В этом примере для `L` создаётся список ключей `['a', 'c', 'b']`. При обмене `'c'` и `'b'` в списке ключей местами, обмениваются `'c'` и `'B'` в исходном списке. Далее представлен пример сортировки строк по их длине:

```
L = ['abcd', 'z', 'mno']
L.sort(key=len) # L → ['z', 'mno', 'abcd']
```

Таким образом, в Python не нужно реализовывать сортировку вручную, а при необходимости сортировки нужно вызвать встроенный метод `sort`, в которую можно передать ключ сортировки.

Также в Python реализованы функции `min` и `max`, которые либо принимают несколько параметров, перечисленных через запятую, либо принимают один параметр, являющийся произвольной последовательностью:

```
L = [5, 6, 1, 3, 4, 2]
min(L) # → 1
max(L) # → 6
```

Так же как метод списка `sort`, функция `min`/`max` принимает ключевой параметр `key`, который вычисляет значение ключей для каждого элемента, находит минимальное/максимальное значение ключа и сохраняет соответствующее ему значение, в результате возвращается искомое значение.

Функции `min` и `max` находят минимум и максимум в любой последовательности:

```
min('BaC') # → 'B'
max(range(1, 10, 2)) # → 9
```

Метод `sort`, описанный ранее, работает только со списками, но часто требуется отсортировать произвольную последовательность. Это можно выполнить с помощью встроенной функции `sorted`. Эта функция в качестве первого параметра принимает произвольную последовательность и *возвращает отсортированный список*. Исходная последовательность не изменяется. Кроме этого, эта функция может настраиваться с помощью ключевых параметров `reverse` и `key`:

```
L = [5, 1, 3, 4, 2]
sorted(L) # → [1, 2, 3, 4, 5]; L → [5, 1, 3, 4, 2]
sorted('BaC', key=mylower) # → ['a', 'B', 'C']
sorted(range(1, 10, 3), reverse=True) # → [7, 4, 1]
```

Функция, передаваемая в качестве ключа, не обязана зависеть только от элементов в сортируемом списке, но также может использовать другие объекты. Например, отсортируем список сотрудников по департаментам, порядок которых задан в списке.

```
posts = ['programmer', 'manager', 'security']
def by_post(employee_info):
    return posts.index(employee_info[1])
employees = [['Bob', 'manager'], ['John', 'programmer'], ['Alice', 'manager']]
L = sorted(employees, key=by_post)
# L → [['John', 'programmer'], ['Bob', 'manager'], ['Alice', 'manager']]
```

Метод `reverse` разворачивает список на месте и ничего не возвращает в качестве результата. Функция `reversed` создаёт список из последовательности и разворачивает его. Результат этой функции требуется передать в конструктор списка:

```
L = [1, 2]
L.reverse()                 # L → [2, 1]
L2 = list(reversed(L))      # L2 → [1, 2]; L → [2, 1]
L3 = list(reversed('abc')) # L3 → ['c', 'b', 'a']
```

Сортировка в Python является *стабильной* (*stable*). Это означает, что у объектов с одинаковым значением ключа сохраняется исходный порядок, даже если указан параметр `reverse`:

```
data = [['a', 1], ['b', 2], ['a', 2], ['b', 1]]
def zero_item(a):
    return a[0]
a = sorted(data, key=zero_item)
# a → [['a', 1], ['a', 2], ['b', 2], ['b', 1]]
a = sorted(data, key=zero_item, reverse=True)
# a → [['b', 2], ['b', 1], ['a', 1], ['a', 2]]
```

В этом примере первая сортировка списка выполняется по нулевому элементу, сохраняя порядок элементов с одинаковым нулевым элементом, то есть `('a', 1)` будет идти перед `('a', 2)`, а `('b', 2)` — перед `('b', 1)`. В последней строке сортировка опять выполняется по нулевому элементу, но уже в обратном порядке, оставляя при равенстве ключей порядок элементов согласно первоначальному порядку. Рассмотренное свойство стабильности позволяет сортировать контейнер по нескольким критериям. Рассмотрим пример сортировки сотрудников сначала по должности, затем по имени. Для получения этого результата нужно сначала отсортировать по имени, затем по должности.

```
L = [['John', 'manager'], ['Alice', 'programmer'], ['Bob', 'manager']]
def name(info):
    return info[0]
L.sort(key=name)
def post(info):
    return info[1]
L.sort(key=post) # L → [['Bob', 'manager'], ['John', 'manager'],
↪  ['Alice', 'programmer']]
```

Предыдущий пример можно было реализовать также следующем образом.

```
L = [['John', 'manager'], ['Alice', 'programmer'], ['Bob', 'manager']]
def post_name(info):
    return info[1], info[0]
L.sort(key=post_name) # L → [['Bob', 'manager'], ['John', 'manager'],
↪  ['Alice', 'programmer']]
```

Для большего понимания настраиваемых сортировок рассмотрим способ сортировки по ключу без использования параметра `key`. Этот способ называется декорирование-сортировка-удаление декорирования (decorate-sort-undecorate), состоящий из трёх шагов, указанных в названии. Декорирование — добавление к каждому элементу списка информации для контроля сортировки, сортировка — сортировка декорированного списка, раздекорирование — удаление добавленной информации с получением исходных значений элементов. Рассмотрим на примере сортировки строк по длине:

```
L = ['abc', 'z', 'mno']
# декорирование
dec = [[len(s), i, s] for i, s in enumerate(L)]
# dec → [[3, 0, 'abc'], [1, 1, 'z'], [3, 2, 'mno']]
dec.sort() # сортировка декорированного списка
# dec → [[1, 1, 'z'], [3, 0, 'abc'], [3, 2, 'mno']]
# удаление декорирования
L = [s for s_len, i, s in dec] # L → ['z', 'abc', 'mno']
```

Разберём этот пример, помня о том, что списки сортируются лексикографически. Для каждого элемента создаётся список из трёх элементов: длины строки, индекса строки в списке и самой строки. Получившийся декорированный список будет сортироваться по длине строки. Индексы же в списке были добавлены для того, чтобы при равенстве ключей элементы с одинаковым индексом сохраняли своё местоположение, то есть для стабильности сортировки. После сортировки извлекаем строки из декорированного списка (удаляем декорирование) с применением списковых включений.

7.12. Кортежи

Кортеж (*tuple*) является неизменяемой последовательностью, то есть является неизменяемым аналогом списка. Кортеж представляется классом `tuple`. Кортежи используются тогда, когда необходима или предпочтительна неизменяемость: в качестве ключей словаря или элементов множества, для передачи последовательности в функцию, чтобы она не могла изменить кортеж.

Создание кортежей похоже на создание списков. Кортежи могут создаваться с помощью литералов, содержащие в круглых скобках разделённые запятыми элементы. Нужно заметить, что для создания кортежа из одного элемента требуется указать запятую `(1,)`,

иначе круглые скобки воспринимаются как группировка объектов и будут отброшены. Конструктор `tuple` создаёт кортеж, принимая произвольную последовательность. При отсутствии параметра создаётся пустой кортеж. Как и у списка, кортежи при создании сохраняют порядок элементов последовательности:

```
t = (1, 2, 3)     # t → (1, 2, 3)
t = tuple('123') # t → ('1', '2', '3')
```

Стоит заметить, что создание кортежа происходит с помощью запятых, а не круглых скобок. Круглые скобки являются необязательными, кроме случаев задания пустого кортежа или возникновения двусмысленности, например, `f(1, 2, 3)` вызывает функцию, передавая три аргумента, `f((1, 2, 3))` вызывает функцию, передавая аргумент-кортеж. Также кортеж создаётся при обмене переменных, возвращении нескольких значений из функции и во время других похожих операций:

```
a, b = b, a # в правой части создаётся кортеж (b, a)
def f():
    return 1, 2 # создаётся и возвращается кортеж (1, 2)
```

Однако нельзя создавать кортежи с помощью включений, потому что списковые включения используют добавление вычисленного элемента, а кортежи — неизменны.

Кортежи поддерживают все обычные для последовательностей операции, то есть все те операции над списками, которые их не меняют:

```
t = (1, 2, 3, 4)
t2 = tuple(range(1, 5))
t == t2     # → True
t[0]        # → 1
t.index(1) # → 0
t.count(4) # → 1
t[1:-1]     # t → (2, 3)
```

Кортежи хранят ссылки на объекты. Неизменяемость кортежей означает, что нельзя изменить ссылку, но при этом можно изменять объекты, на которые кортеж ссылается:

```
t = ([], 1)
t[0].append(2) # t → ([2], 1)
t[0] = 3        # ошибка: попытка изменения ссылки
```

Нужно помнить эту особенность при использовании кортежей. Например, использование кортежей не защищает полностью аргументы функции от изменений в вызывающем коде.

7.13. Резюме

Списки хранят последовательность элементов, предоставляя доступ к ним по индексам. Создание списков осуществляется с помощью квадратных скобок с указанием элементов последовательности через запятую и путём вызова конструктора `list`, передавая произвольную последовательность. Списковые включения позволяют создавать списки посредством обхода произвольной последовательности. Списки могут иметь произвольную глубину.

Списки предоставляют операцию получения и изменения элемента по индексу, выполняющуюся очень быстро. Списки являются изменяемыми: можно менять и удалять элементы списка, можно добавить новый элемент.

Срезы списков позволяют получать подсписок, указав начало, конец и шаг среза.

Стандартные функции `max`, `min` и `sorted` дают возможность найти максимум, минимум и сортировки списка с возможностью указания функции-ключа для сравнения.

Кортеж является неизменяемым аналогом списка.

8. Операции над последовательностями

В предыдущих главах были рассмотрены следующие типы данных, являющиеся последовательностями: строки, списки и кортежи. Как можно было заметить, эти типы имеют некоторые общие операции (operations). В этой главе описываются операции над последовательностями, собранные воедино.

8.1. Операции над произвольными последовательностями

В таблице 5 продемонстрированы операции над произвольными (изменяемыми и неизменяемыми) последовательностями. Последовательности `s` и `t` должны принадлежать одному типу данных.

Таблица 5 — Список операций над произвольными последовательностями

Операция	Описание
`a in s`, `a not in s`	Содержится/не содержится ли элемент `a` в последовательности `s`
`s + t`	Конкатенация последовательностей `s` и `t`
`s * n` или `n * s`	Повторение `n` раз последовательности `s`
`s[i]`	Значение `i`-го элемента `s` (начало с `0`)
`s[i:j]`	Срез `s` с `i` до `j`, не включая его, с шагом `1`
`s[i:j:k]`	Срез `s` с `i` до `j`, не включая его, с шагом `k`
`len(s)`	Длина (размер) последовательности `s`
`min(s)`/`max(s)`	Минимальный/максимальный элемент последовательности `s`
`s.index(a[, i[, j]])`	Индекс первого появления элемента `a` в `s` (начиная с индекса `i`, заканчивая индексом `j`, не включая его)
`s.count(a)`	Количество элементов `a` в последовательности `s`

Операции в таблице 5 представлены в порядке возрастания приоритета (см. табл. 1), при этом ограниченные горизонтальными линиями операции имеют одинаковый приоритет.

Из рассмотренных ранее классов эти операции поддерживают строки, списки, кортежи и диапазоны. Объект диапазона, возвращённый функцией `range`, поддерживает все операции из таблицы 5, исключая операции `+` и `*`.

Последовательности одинаковых типов можно сравнивать между собой с помощью операций `==`, `!=`, `<`, `<=`, `>`, `>=`.

Операции из представленной таблицы, возвращающие последовательность, возвращают последовательность того типа, к которому применена операция.

Операция `s * n` при значении `n` $\leq$ `0` возвращает пустую последовательность.

Последовательности могут реализовывать операции из таблицы собственным образом, например, строки реализуют оператор `in` через проверку вхождения подстроки.

Стоит помнить, что операция `index` выбрасывает исключение, если элемент в последовательности отсутствует.

Класс `collections.abc.Sequence` используется для реализации собственного класса последовательности.

8.2. Операции над изменяемыми последовательностями

В таблице 6 показаны операции, которые можно выполнить над изменяемыми последовательностями. В этой таблице последовательность `s` является изменяемой последовательностью, тогда как в качестве `t` может выступать произвольная последовательность. Из изученных классов только список поддерживает эти операции.

Таблица 6 — Список операций над изменяемыми последовательностями

Операция	Описание
`s[i] = a`	Присваивание значения `a` в `i`-й элемент
`s[i:j] = t`, `s[i:j:k] = t`	Присваивание срезу произвольной последовательности `t` длины, равной длине среза
`s.reverse()`	Разворачивает последовательность `s` на месте
`s.append(a)`	Добавление элемента `a` в конец последовательности `s`
`s.extend(t)`, `s += t`	Добавление элементов произвольной последовательности `t` в конец последовательности `s`
`s *= n`	Последовательность изменяется повторением себя `n` раз
`s.insert(i, a)`	Добавление элемента `a` в позицию `i`
`del s[i:j]`, `del s[i:j:k]`	Удаление элементов среза из последовательности
`s.pop([i])`	Удаление и возвращение элемента из `i`-й позиции (по умолчанию удаляется последний элемент)
`s.remove(a)`	Удаление первого вхождения элемента `a` из последовательности `s`
`s.clear()`	Очищение последовательности
`s.copy()`	Копирование последовательности

Необходимо помнить, что метод `remove` возбуждает исключение `ValueError`, если элемент `a` отсутствует в `s`.

Если число `n` равняется нулю или является отрицательным, тогда операция `s *= n` очищает последовательность.

Для реализации собственного класса изменяемой последовательности примеяется `collections.abc.MutableSequence`.

8.3. Резюме

В данной главе были описаны операции, общие как для произвольных последовательностей, так и для изменяемых последовательностей. Это позволяет проще запомнить имеющиеся методы в различных классах последовательностей, например: строках, списках, кортежах, диапазонах. Операции `+` требуется передавать последовательности одного типа. Операциям `+=` и `s[i:j] = t` над изменяемыми последовательностями можно передавать произвольную последовательность.

9. Словари

Словарь (*dictionary*) позволяет хранить значения по ключам, иными словами, элементы являются поименованными. Синонимами словарей в других языках программирования являются отображения (map, hashmap). В разделе 1.10 были рассмотрены основы словарей.

Ключи в списке являются уникальными (как индексы в списке), то есть в словаре не может быть одинаковых ключей. Ключами могут быть только неизменяемые объекты, например: числа, строки, кортежи и объект `frozenset` (тип данных, который будет рассмотрен позже). В качестве значений могут выступать произвольные объекты: числа, строки, списки, словари и так далее.

В словари в качестве значений можно поместить списки и другие основные типы, что позволяет легко строить сложные структуры, например, список словарей, значениями которых являются списки строк. Далее рассмотрен пример, в котором в список в словаре добавляется новый элемент:

```
d = {'a': [1, 2, 3]}
d['a'].append(4) # d → {'a': [1, 2, 3, 4]}
```

Функция `print` выводит словарь на экран. Если словарь содержит контейнеры, тогда их содержимое также будет выведено на экран:

```
print(d) # > {'a': [1, 2, 3, 4]}
```

С помощью функции `len` можно получить размер словаря (количество пар ключ/значение):

```
print(len(d)) # > 1
```

Словари лишь хранят ссылки на объекты ключей и значений, поэтому может возникнуть проблема разделяемых ссылок на значения.

Далее рассмотрим работу со словарями более подробно.

9.1. Создание словаря

Для задания словарей вручную можно воспользоваться синтаксисом литералов: в фигурных скобках через запятую указывается разделённая двоеточием пара ключ/значение `{ключ1: значение1, ключ2: значение2}`. В качестве ключа могут выступать любые неизменяемые объекты. Для создания пустого словаря используются пустые фигурные скобки:

```
d = {} # пустой словарь
languages = {'John': ['c++', 'c#'], 'Bob': ['Python']}
ratings = {5: ['John', 'Bob'], 4: ['Alice']}
```

Кроме того, словарь можно создать при помощи конструктора `dict`. Пустой словарь создаётся вызовом `dict` без указания параметров: `d = dict()`. Конструктор можно вызвать с ключевыми параметрами:

```
dict(banana=70, apple=50) # → {'banana': 70, 'apple': 50}
```

Названия параметров преобразуются в строки. Этот способ позволяет создать словарь, в котором ключи могут быть только в виде строк, ограничений на значения нет.

Словари можно создать также из списка ключей с помощью метода класса `dict.fromkeys`. Этот метод принимает последовательность ключей, создаёт словарь с этими ключами, значение которых устанавливается в `None`:

```
dict.fromkeys(['a', 'b']) # → {'a': None, 'b': None}
dict.fromkeys('ab')       # → {'a': None, 'b': None}
```

Вторым параметром можно указать значение, которое устанавливается переданным ключам:

```
d = dict.fromkeys(['a', 'b'], 0) # d → {'a': 0, 'b': 0}
```

При этом при создании значения копируется ссылка на переданный объект, а не сам объект. Поэтому в качестве значения следует с осторожностью использовать изменяемые объекты, например, `fromkeys` копирует ссылку на объект, переданный в качестве второго аргумента:

```
d = dict.fromkeys([1, 2], []) # d → {1: [], 2: []}
# ключи 1 и 2 ссылаются на один объект списка
d[1].append('a')              # d → {1: ['a'], 2: ['a']}
```

Последним способом создания словаря является способ передачи списка пар ключ/значение:

```
prices = dict([('banana', 100), ('orange', 50)])
```

Используя этот способ и функцию `zip`, можно создать словарь путём передачи списка ключей и значений:

```
prices = dict(zip(['banana', 'orange'], [100, 50]))
```

Кроме создания можно скопировать уже существующий словарь, получив копию, хранящую ссылки на ключи и значения, из-за чего возникает проблема разделяемых ссылок на значения (ключи неизменяемы):

```
d = {'a': [1], 'b': [2]}
dcopy = d.copy()  # dcopy → {'a': [1], 'b': [2]}
dcopy['a'][0] = 3 # d → {'a': [3], 'b': [2]}
```

9.2. Сравнение словарей

Словари можно сравнивать только на то, равны они или нет. Словари равняются, только если они содержат одинаковые пары (ключ, значение) без учёта порядка:

```
d = {'a': 1, 'b': 2}
d == {'b': 2, 'a': 1}           # → True
d == dict([('a', 1), ('b', 2)]) # → True
```

При проверке на равенство словарей всегда сравниваются все пары ключ/значение без сравнения ссылок, поэтому в следующем коде выполняется полная проверка всех пар, что при большом размере словаря может привести к снижению производительности программы:

```
d == d # → True; сравниваются все пары
```

Попытка сравнения порядка двух словарей с помощью операторов <, <=, >, >= приводит к исключению `TypeError`.

9.3. Доступ к элементам словаря

Для получения значения по ключу используются квадратные скобки с указанием ключа:

```
prices = {'banana': 100, 'orange': 50}
a = prices['orange'] # a → 50
```

Доступ по несуществующему ключу ведёт к выбрасыванию исключению KeyError:

```
b = prices['apple'] # Исключение KeyError
```

Поэтому во избежание исключения требуется проверка присутствия ключа в словаре, что можно выполнить с помощью оператора in, а проверку отсутствия — с помощью not in:

```
prices = {'banana': 100, 'orange': 50}
'apple' in prices          # → False
'banana' in prices         # → True
if 'apple' not in prices:
    print('no apple')      # > no apple
```

Конструкция проверки наличия ключа и получения с помощью него значения является очень употребимой, поэтому в Python существует метод get, который возвращает значение по ключу. При отсутствии ключа исключение не выбрасывается, а возвращается либо None, либо значение, переданное в качестве второго параметра:

```
p.get('banana')    # → 100
p.get('apple')     # → None
p.get('apple', 0) # → 0
```

Проверка существования ключа в словаре и получение значения по ключу выполняются очень быстро.

9.4. Обход словаря

В этом разделе будут рассмотрены различные способы обхода словаря `p = {'banana': 70, 'apple': 50}`.

Обойти все ключи можно с помощью оператора for, указав после ключевого слова in словарь:

```
for fruit in p:
    print(fruit, end=' ') # > banana apple
```

Кроме того, можно воспользоваться методом keys(), возвращающим последовательность ключей:

```
for fruit in p.keys():
    print(fruit, end=' ') # > banana apple
```

Другими словами, в контексте обхода словарей p является сокращением для p.keys().

Оператор for принимает произвольную последовательность, то есть из предыдущего способа можно сделать вывод, что p в контексте for является последовательностью ключей. Тогда для получения списка ключей можно воспользоваться конструктором list:

```
list(p)        # → ['banana', 'apple']
list(p.keys()) # → ['banana', 'apple']
```

Для обхода словаря по значениям нужно воспользоваться методом `values()`:

```
for price in p.values():
    print(price, end=' ') # > 70 50
list(p.values())          # → [70, 50]
```

Метод `items()` возвращает последовательность пар-кортежей ключ/значение:

```
list(p.items()) # → [('banana', 70), ('apple', 50)]
```

Для обхода словаря одновременно по ключам и значениям с помощью метода `items()` и с помощью присваивания кортежа именам `fruit` и `price` в контексте `for`:

```
for fruit, price in p.items():
    print(fruit, price, end=' ') # > banana 70 apple 50
```

Для обхода ключей словаря в обратном порядке применяется функция `reversed(d)`, `reversed(d.values())`, `reversed(d.items())`, возможность использования которой на словарях появилась в Python 3.8:

```
for fruit in reversed(p):
    print(fruit, end=' ') # > apple banana
```

9.5. Порядок ключей

В предыдущем разделе были рассмотрены способы обхода словаря, однако не был описан порядок, в котором этот обход выполняется. До Python 3.7 и CPython 3.6 нет четкого порядка ключей, в котором осуществляется обход:

```
for k in {'c': 1, 'b': 3, 'a': 5}:
    print(k, end=' ') # > a b c - возможный вывод
```

Результат этого вывода может быть совершенно другим при запуске на другом компьютере или другой версии Python.

Начиная с Python 3.7 и CPython 3.6, появился чёткий порядок ключей, а именно в порядке вставки ключей в словарь:

```
for k in {'c': 1, 'b': 3, 'a': 5}:
    print(k, end=' ') # > c b a
```

Методы `keys`, `values` и `items` возвращают значения в порядке вставки ключей. При выводе словаря на печать функция `print` также выводит элементы словаря в порядке вставки ключей.

Частой задачей является обход отсортированных ключей. Для выполнения эти задачи применяется следующий код:

```
prices = {'banana': 70, 'apple': 50}
for fruit in sorted(prices): # или sorted(prices.keys())
    print(fruit, end=' ')        # > apple banana
for fruit, price in sorted(prices.items()):
    print(fruit, price, end=' ') # > apple 50 banana 70
```

Результатом `prices.items()` будет последовательность кортежей ключ/значение, поэтому при сортировке кортежи сравниваются сначала по нулевому элементу (ключу) и лишь затем по первому (значению). Но из-за того, что все ключи уникальны, сравнение значений выполняться не будет, поэтому `sorted(prices.items())` вернёт список пар ключ/значение, этот список будет отсортирован по ключам.

Словари сохраняют порядок вставки. При этом обновление ключей не влияет на порядок вставки, однако после удаления вставка ключа произойдёт в конец словаря:

```
prices = {'banana': 50, 'apple': 100}
# порядок ключей: 'banana', 'apple'
prices['banana'] = 70 # порядок ключей: 'banana', 'apple'
del prices['banana']
prices['banana'] = 90 # порядок ключей: 'apple', 'banana'
```

9.6. Представление словарей

Объекты, возвращаемые методами `keys()`, `values()`, `items()`, являются *представлениями словарей* (*dictionary view objects*). Они обеспечивают динамическое представление элементов словаря, что даже при изменении словаря позволяет отражать текущее состояние со всеми внесёнными изменениями:

```
d = {'a': 1, 'b': 2}
keys = d.keys()
list(keys) # → ['a', 'b']
d['c'] = 3
list(keys) # → ['a', 'b', 'c']
```

В этом примере изменение словаря происходит до обхода всех ключей, но также можно изменять значения в словаре во время обхода:

```
for k in keys:
    d[k] += 10
# d → {'a': 11, 'b': 12, 'c': 13}
```

Однако во время обхода нельзя ни удалять элементы, ни добавлять новые, иначе может произойти исключение `RuntimeError` в начале следующей итерации:

```
for k in keys:
    d['d'] = 10 # RuntimeError
for k in keys:
    del d[k]    # RuntimeError на следующей итерации
```

Но можно удалить элемент с последующим выходом из цикла, избегая перехода на новую итерацию:

```
for k in keys:
    del d[k]
    break
# d → {'b': 12, 'c': 13}
```

Можно выяснить размер представлений с помощью функции `len`: `len(d.items())`.

Благодаря тому, что `keys()` и `values()` обходят словарь в порядке вставки ключей, можно развернуть (поменять местами ключ/значение) список с помощью следующего кода:

```
reversed_d = dict(zip(d.values(), d.keys()))
```

Стоит быть осторожным при выполнении подобного кода. Во-первых, значения в исходном словаре могут быть объектами изменяемого типа. Во-вторых, в исходном словаре может быть несколько значений, вследствие чего некоторые значения не сохранятся.

Можно выполнить проверку вхождения элемента в представление словарей с помощью оператора `in`: `12 in d.values()`.

Представления `keys` и `items` разных словарей можно сравнивать на равенство вне зависимости от порядка вставки:

```
d, d2 = {'a': 1, 'b': 2}, {'b': 2, 'a': 1}
d.keys() == d2.keys()   # → True
d.items() == d2.items() # → True
```

Однако представления `values` нельзя сравнивать, даже если это будут представления одного и того же словаря:

```
d.values() == d2.values() # → False
d.values() == d.values()  # → False
```

9.7. Включения словарей

Включения словарей (*dictionary comprehension*) позволяют создать словарь, обходя произвольную последовательность. Как включения списков имеют синтаксис, похожий на создание списков с помощью литералов, так и включения словарей имеют синтаксис, подобный литералам словарей:

```
d = {ch:ord(ch) for ch in 'ab...z'}
# d → {'a': 97, 'b': 98, ...}
d = {v: k for k, v in d.items()}
# d → {97: 'a', 98: 'b', ...}
# фильтрация товаров, цена которых больше 100
d = {n:p for n, p in prices.items() if p > 100}
d = {n:sqrt(n) for n in [1, 2, 3]}
# d → {1: 1.0, 2: 1.4142135623730951, 3: 1.7320508075688772}
```

В первой строке создаётся словарь, где в качестве ключей выступают символы английского алфавита, а в качестве значений — их коды. Во второй строке ключи и значения меняются местами. В третьей строке фильтруются товары, которые имеют цену, большую `100`. В последней строке для ключей `1, 2, 3` создаются значений, равные квадратному корню из этих чисел.

9.8. Дополнительно

Числовые типы нарушают правила проверки чисел: два числа, эквивалентные математически (например, числа `6` и `6.0`), могут использоваться в качестве ключей взаимозаменяемо:

```
d = {1: 'a'}
print(d[1.0])              # > a
from decimal import Decimal
print(d2[Decimal('1.0')]) # > a
```

Метод `setdefault(key[, default=None])` возвращает значение ключа `key`, если ключ присутствует в словаре. Если нет, тогда в словарь помещается ключ `key` со значением `default`, которое возвращается. По умолчанию параметр `default` устанавливается в `None`:

```
d = {}
a = d.setdefault('a', 0) # a → 0; d → {'a': 0}
d['a'] = 5               # d → {'a': 5}
a = d.setdefault('a', 0) # a → 5; d → {'a': 5}
```

9.9. Резюме

Словари позволяют хранить значения, доступ к которым осуществляется по ключам. Ключами в словаре могут быть только неизменяемые объекты, а сами ключи в словаре являются уникальными. На значения никаких ограничений не накладывается. Основной операцией в словаре является доступ к значению по ключу для получения или изменения значения. Эти операции, как и проверка существования ключа в словаре, выполняются очень быстро (как индексация списка). Словари являются изменяемыми: можно изменять значения в словаре, удалять значения и добавлять новые. Словари можно сравнивать только на равенство.

Включения словарей позволяют создавать словарь путём обхода произвольной последовательности.

Обход словаря осуществляется с помощью оператора `for` по элементам, полученным вызовом методов `keys`, `values` и `items`, которые возвращают представления словарей. Эти представления являются динамическим отражением соответствующих элементов (ключей, значений и пар). Хранение и обход элементов словаря выполняются в порядке вставки.

10. Множества

10.1. Основы множеств

Множество (*set*) — *неупорядоченная* коллекция *уникальных* и *неизменяемых* объектов. Множества похожи на ключи в словаре без значений, однако, будучи неупорядоченной коллекцией, множества не запоминают ни позицию, ни порядок вставки. В Python множества представлены классом `set`.

Из определения множества следует, что его элементами могут быть числа, строки, кортежи, объекты класса `frozenset` (неизменяемые множества, которые будут рассмотрены позже).

Множества могут хранить только неизменяемые объекты, но само множество можно изменить применением методов добавления и удаления. Множества являются неупорядоченной коллекцией данных, поэтому нельзя выполнить индексацию или срез (и другие операции над последовательностями, подразумевающими порядок элементов).

Класс `set` поддерживает операции над математическими множествами, например: объединение и пересечение двух множеств.

Как и другие рассмотренные коллекции, множества хранят не сами объекты, а ссылки на них.

10.2. Создание множеств

Для создания множеств используются литералы, которые внутри фигурных скобок содержат элементы, перечисленные через запятую:

```
s = {'a', 'b', 'b'} # s → {'a', 'b'}
```

Для создания множеств используются фигурные скобки ввиду схожести словарей (а точнее ключей словаря) и множеств.

Подобно другим коллекциям поэлементный вывод множества можно выполнить с помощью функции `print`:

```
print({'a', 'b', 'b'}) # > {'a', 'b'}
```

Для создания пустого множества необходимо вызвать конструктор без параметров:

```
s = set()
```

Для создания пустого множества необходимо использовать именно конструктор, потому что пустые фигурные скобки `{}` создают пустой словарь. При этом `print` пустое множество выводит как `set()`:

```
print(set()) # > set()
```

Множество можно создать путём передачи произвольной последовательности неизменяемых объектов в конструктор класса. После создания множества в нём будут храниться элементы переданной последовательности в единственном экземпляре:

```
s = set('abb')            # s → {'a', 'b'}
s = set(['a', 'b', 'b']) # s → {'a', 'b'}
```

Множество можно скопировать из существующего, применив метод `copy`:

```
s1 = s.copy() # s1 → {'a', 'b'}
```

Множество хранит ссылки на объекты, а метод `copy` копирует эти ссылки. Однако после копирования не возникает проблем с разделяемыми ссылками благодаря тому, что множество содержит ссылки только на неизменяемые объекты.

10.3. Включения множеств

Включения множеств (*set comprehension*) подобно включениям других типов данных позволяют создавать множества, обходя произвольную последовательность:

```
s = {e for e in ['a', 'b', 'b']}          # s → {'a', 'b'}
s = {e for e in [1, 2, 3] if e % 2 == 0} # s → {2}
s = {x**2 for x in [1, 2, -2, 3]}         # s → {1, 4, 9}
s = {a+b for a in 'abc' for b in '12'}
# s → {'c1', 'a1', 'a2', 'b1', 'b2', 'c2'}
```

10.4. Проверка наличия элемента

Для проверки наличия элемента во множестве применяется оператор `in`:

```
s = {1, 2, 1}
1 in s # → True
3 in s # → False
```

Подобная проверка выполняется так же эффективно, как и проверка существования ключа в словаре.

10.5. Обход множества

Обойти все элементы множества можно с помощью цикла `for`, но из-за того, что элементы множества не имеют чёткого порядка, обход осуществляется в произвольном порядке:

```
s = {'a', 'b', 'c'}
for c in s:
    print(c) # > a c b
```

Произвольный порядок обхода элементов множества не всегда является желаемым, часто необходимо обойти элементы множества в отсортированном виде. Для этого необходимо воспользоваться рассмотренной ранее функцией `sorted`:

```
for c in sorted(s):
    print(c) # > a b c
```

10.6. Операции над множествами

Для получения количества элементов во множестве используется функция `len`.

```
len({0, 1, 0, 1}) # → 2
```

Важной операцией над множествами является операция сравнения двух множеств на равенство. Во время этого сравнения проверяется, что два множества содержат одни и те же элементы:

```
s1 = set([1, 2, 0])
s1 == set(range(3))    # → True
s1 == {1, 0, 2, 0}     # → True
s1 == {'0', '1', '2'} # → False
```

Подобное сравнение может использоватся для проверки двух последовательностей на равенство элементов без учёта порядка (в предположении, что в обоих последовательностях содержатся уникальные элементы).

Так же как для рассмотренных ранее коллекций, для проверки наличия элемента в множестве используется операторы `in` и `not in`. Эта проверка выполняется очень быстро, не требуя обхода всего множества:

```
s = {'a', 'b', 'c', 'd'}
'a' in s # → True
'f' in s # → False
```

10.7. Добавление элементов

Далее будут рассмотрены методы добавления и удаления элементов. Все эти методы работают эффективно и выполняются очень быстро.

Для добавления элементов во множество используются два метода. Метод `add` позволяет добавить один элемент в множество, если этот элемент уже там существует, ничего не происходит:

```
s = {'a', 'b'}
# Добавление нового элемента
s.add('c') # s → {'a', 'b', 'c'}
# Добавление существующего элемента
s.add('b') # s → {'a', 'b', 'c'}
```

Второй способ добавления `update` позволяет добавить элементы любой последовательности:

```
s = {'a', 'b'}
# Добавление каждого элемента последовательности
s.update('xyz') # s → {'y', 'a', 'c', 'x', 'b', 'z'}
s = {'a', 'b'}
# Добавление одного элемента
s.add('xyz') # s → {'a', 'b', 'xyz'}
```

Оба описанных в этом разделе метода ничего не возвращают.

10.8. Удаление элементов

Для удаления элемента из множества применяется метод `remove`.

```
s = {'a', 'b', 'c', 'd'}
s.remove('b') # s → {'a', 'c', 'd'}
```

Однако при попытке удалить из словаря элемент, которого нет, будет исключение:

```
s.remove('not_exist') # → исключение KeyError
```

Поэтому перед удалением элемента с помощью `remove` необходимо проверить существование элемента с применением оператора `in`. Но более удобным способом является метод `discard`, который удаляет элемент при его наличии и ничего не делает в противном случае:

```
s.discard('c')         # s → {'a', 'd'}
s.discard('not_exist') # s → {'a', 'd'}
```

Помимо удаления элементов по значению есть возможность удаления произвольного элемента с помощью метода `pop`, который в отличие от `remove` и `discard` возвращает удалённое значение:

```
s = {'a', 'b', 'c', 'd'}
t = s.pop() # s → {'a', 'b', 'd'}; t → 'c'
t = s.pop() # s → {'a', 'd'}; t → 'b'
```

С помощью этого метода можно пройтись по всему множеству, одновременно с этим удаляя посещённые элементы:

```
s = {'a', 'b', 'c', 'd'}
while s:
    print(s.pop()) # > c b d a
# s → set()
```

Для удаления всех элементов (очистки всего множества) используется метод `clear`.

10.9. Операции над математическими множествами

В Python объекты класса `set` могут применяться в качестве математических множеств. На рисунке 36 изображены диаграммы Вена: пересечение (intersection), объединение (union), разность (difference) и симметричная разность (исключающее или, symmetric difference) двух множеств A и B.

Для использования этих операций можно воспользоваться различными способами. Первый способ — операторы. Для нахождения пересечения используется оператор `&`, а объединения, разности, симметричной разности — операторы `|`, `-`, `^` соответственно. Далее показан пример использования операторов, а графическое представление этого примера продемонстрировано на рисунке 37.

```
s1 = {'a', 'b', 'c', 'd'}
s2 = {'b', 'd', 'e'}
s1 & s2 # пересечение: → {'b', 'd'}
s1 | s2 # объединение: → {'a', 'b', 'c', 'd', 'e'}
s1 - s2 # разность: → {'a', 'c'}
s1 ^ s2 # симметричная разность: → {'a', 'c', 'e'}
```

Кроме описанных выше операций можно проверить, что одно множество является надмножеством/подмножеством другого. Например, на диаграмме Эйлера (рисунок 38) множество A является подмножеством B, а множество B является надмножеством A. Оператор `>` (`<`) проверяет, является ли первое множество строгим (множества не равны) надмножеством (подмножеством). Если два множества равны, оба оператора возвращают `False`:

```
s1 > s2, s2 < s1 # → (False, False)
```

Стоит заметить, что в отличие от рассмотренных ранее типов операторы `>` (`<`) для множеств используются не для определения порядка. Порядок множеств не имеет смысла, потому что функции `<`, `==`, `>` для двух множеств, не являющихся надмножеством/подмножеством другого, всегда возвращают `False`. Таким образом, не имеет смысла выполнять поиск минимума/максимума и сортировать коллекцию множеств, если не указать свою ключевую функцию.

Операция `>=` (`<=`) для множеств проверяет, равны ли два множество или является ли первое множество надмножеством (подмножеством) второго.

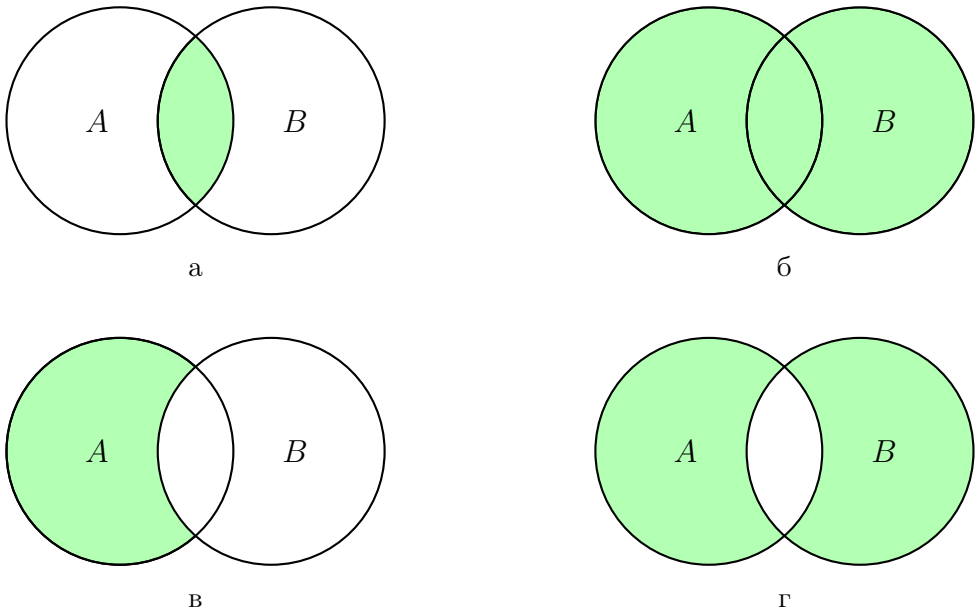

Рис. 36. Операции над математическими множествами: а — пересечение, б — объединение, в — разность, г — симметричная разность

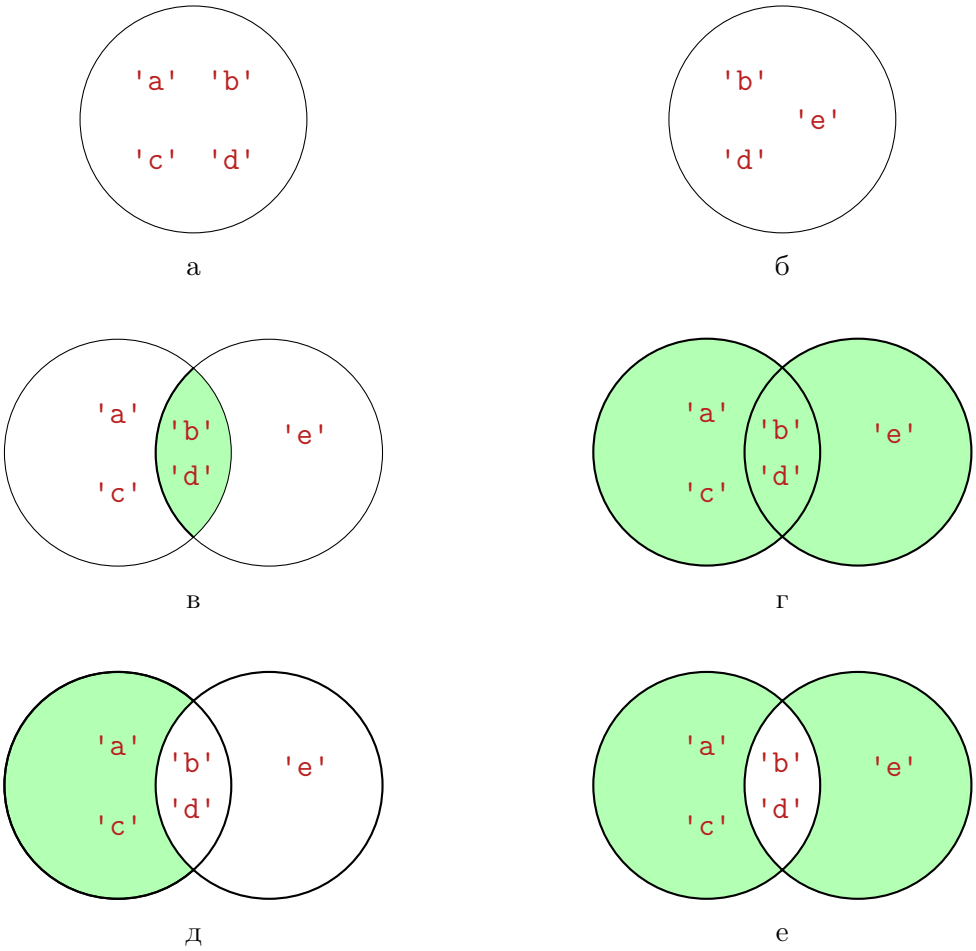

Рис. 37. Графическое представление операций над множествами: а — `s1`, б — `s2`, в — `s1 & s2`, г — `s1 | s2`, д — `s1 - s2`, е — `s1 ^ s2`

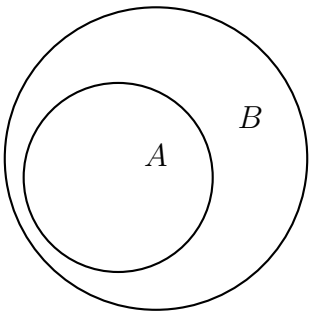

Рис. 38. Подмножество и надмножество: A — подмножество B, B — надмножество A

Вторым способом работы с математическими множествами являются методы. Для нахождения пересечения используется метод `intersection`, объединения — `union`, разности — `difference`, симметричной разности — `symmetric_difference`. Для проверки, является ли множество надмножеством/подмножеством другого, применяются методы `issuperset`/`issubset`.

Далее представлен пример использования метода нахождения разности:

```
s1.difference(s2) # → {'a', 'c'}
```

Также есть операции с обновлением, например, `s1 |= s2` и `s1.intersection_update(s2)`, позволяющие изменить множество в соответствии с результатом операции.

Операторы и методы `union`, `intersection`, `difference` позволяют передать несколько множеств:

```
s1.intersection(s2, set('cde')) # s → {'d'}
```

Методы принимают любую последовательность, однако операторы принимают только множество, поэтому `set('abc') | '123'` выбрасывает исключение.

10.10. Представления в словаре и множества

Как упоминалось в разделе 9.6, ключи словаря и множества похожи (только ключи словаря сохраняют порядок вставки), поэтому методы словаря `keys` и `items` возвращают *представления*, которые поддерживают поведение множеств, например: пересечение и объединение. Осуществляется данная поддержка только с помощью операторов, но не методов.

```
d = {'a':2, 'c': 1}
s = {'a', 'b'}
s2 = d.keys() & s # пересечение: s2 → {'a'}
d2 = {'a': 3, 'd': 4}
s2 = d.items() | d2.items() # объединение
# s2 → {('c', 1), ('a', 3), ('d', 4), ('a', 2)}
```

10.11. Класс frozenset

В Python для создания неизменяемого множества используется класс `frozenset`, который относится к множеству так же, как кортеж к списку. Конструктор `frozenset` работает аналогично конструктору `set`:

```
fs = frozenset('abc') # fs → frozenset({'c', 'b', 'a'})
```

Класс `frozenset` имеет те же методы, что и `set`, кроме тех, которые изменяют множество. В частности, попытка добавить или удалить элемент из множества приведёт к ошибке. Класс `frozenset` реализован таким образом, чтобы корректно взаимодействовать с обычными множествами:

```
s = set('cba')
fs == s     # → True
s.add('d')
fs == s     # → False
```

Благодаря неизменяемости, объекты класса `frozenset` могут использоваться в качестве ключей словаря и элементов множества:

```
s = {frozenset('ab')} # s → {frozenset({'b', 'a'})}
```

10.12. Применение множеств

Множества, благодаря уникальности своих элементов, могут использоваться для решения следующих задач:

- удаление или поиск дубликатов (при этом порядок исходной последовательности не сохраняется),
- проверка равенств коллекций/последовательностей без учета порядка (нужно удостовериться в отсутствии дубликатов, иначе такая проверка будет работать неверно),
- использование множеств для проверки существования элементов в коллекции (эффективнее списков),
- операции над математическими множествами.

10.13. Пример использования множеств

Пример. Проверить, что в списке целых чисел все элементы различные.

Для решения этой задачи воспользуемся множествами для удаления дубликатов из списка:

```
L = [1, 2, 3, 4, 1]
s = set(L)
```

Исходный список не содержит дубликатов тогда и только тогда, когда длина списка и количество элементов множества совпадают:

```
len(L) == len(s) # → False
```

Благодаря множествам рассмотренный пример решается очень просто.

Недостаток предыдущего способа решения состоит в том, что даже если первые два элемента одинаковые, конструктор `set` проходится по всему списку. Далее представлено более эффективное по времени решение задачи, в котором обход списка прекращается в момент нахождения первого дубликата:

```
def is_unique(L):
    s = {}
    for a in L:
        if a in s:
            return False
        s.add(a)
    return True
```

10.14. Резюме

В данной главе были рассмотрены множества (класс `set`), позволяющие хранить уникальные неизменяемые значения. Добавление, удаление и проверка на вхождение элемента для множеств осуществляются очень быстро.

Множества поддерживают многие операции над коллекциями: определение количества элементов, обход, проверку вхождения элемента; но не поддерживают индексацию и срез.

Множества предоставляют операции для манипулирования ими как математическими множествами.

Включения множеств позволяют создавать множества путём обхода последовательности.

Класс `frozenset` является неизменяемым аналогом `set`.

11. Работа с файлами

Файл — именованная область данных на носителе информации. Файл имеет полное название (путь к файлу + имя самого файла) и обладает атрибутами (владелец, дата создания и тому подобное).

В данном разделе будут рассмотрены способы взаимодействия с файлами при помощи инструментов, предоставляемых Python.

Общая схема работа с файлами в любом языке программирования выглядит следующим образом: открытие файла, чтение и/или запись данных, закрытие файла.

Открытие файла в Python осуществляется с помощью функции `open`, в которую необходимо передать название файла:

```
f = open(r'input.txt') # открытие файла на чтение
```

Для чтения всего файла можно воспользоваться методом `read()`:

```
s = f.read() # s → строка с содержимым файла
```

Для закрытия файла можно воспользоваться методом `close()`:

```
f.close() # закрытие файла
```

11.1. Открытие файла

Встроенная функция `open` создаёт ссылку на файл (файловый объект). Если файл открыть нельзя выбрасывается исключение `OSError`. Функция `open` имеет следующую сигнатуру:

```
open(file, mode='r', buffering=-1, encoding=None,
     errors=None, newline=None, closefd=True, opener=None)
```

Рассмотрим только первые четыре параметра, которые являются наиболее употребимыми.

Полное название файла передаётся в параметре `file`. Для задания пути до файла предпочтительнее использовать неформатированные строки. В качестве параметра `file` можно указать как абсолютный путь (например, `r'c:\input.txt'`), так и относительный путь (например, `r'input.txt'`).

Предположим, что файл Python был запущен из папки `r'C:\a'`. При задании относительного пути поиск файла осуществляется относительно папки, в которой находится запущенный файл Python. Префикс `'.\'` в пути к файлу означает, что нужно искать в текущей папке, поэтому при открытии файла `r'.\b\1.txt'` будет открыт файл `r'C:\a\b\1.txt'`. Префикс `'..\'` означает, что нужно искать в родительской папке, поэтому при открытии файла `r'..\b\2.txt'` будет открыт файл `r'C:\b\2.txt'`.

Файлы в Python разделяются на текстовые и двоичные. При чтении текстовых файлов обработка осуществляется с помощью строк, при работе с двоичными — с помощью массива байтов.

Параметр `mode` задаёт режим открытия файла. Для открытия файла на чтение необходимо передать `'r'`, если файла не существует, тогда выбрасывается исключение. Это значение является значением по умолчанию, поэтому можно опустить параметр `mode`. Параметр `'w'` позволяет открыть файл на запись. Если файл существует, файл очищается, если файла не существует, тогда файл создаётся. Также можно открыть файл для добавления информации в конец файла с помощью параметра `'a'`. Открываемый файл должен

существовать, иначе выбрасывается исключение. Этот способ открытия может быть использован, например, для записи в лог-файлы.

Кроме основных параметров открытия существуют дополнительные параметры. Использование суффикса `'+'` позволяет добавлять в файл: `'r+'` — открытие файла для чтения и записи, чтение происходит из начала файла, а запись — в конец файла; `'w+'` — открытие файла для чтения и записи, если файл существует, он очищается; `'a+'` — открытие файла как `'r+'`, однако если файла не существует, он создаётся.

Параметр `buffering` будет рассмотрен в одном из следующих разделов.

Любые текстовые данные хранятся и обрабатываются как набор чисел в виде массива байтов, для использования числового представления символов применяются различные кодировки. В частности, текстовые данные в файле хранятся в виде набора байтов, полученных при помощи кодировок. Поэтому при открытии текстовых файлов необходимо указывать кодировку для того, чтобы преобразовать байты в строки. Для указания кодировки требуется передать параметр `encoding`:

```
# открытие файла с кодировкой UTF-8 на чтение
f = open('input.txt', encoding='UTF-8')
# открытие файла с кодировкой UTF-8 на запись
f = open('input.txt', 'w', encoding='UTF-8')
```

Стоит напомнить, что строки в Python заданы в Unicode, поэтому интерпретатор сам автоматически конвертирует считанные/записанные данные с помощью указанной кодировки в строку/из строки Unicode. При отсутствии параметра кодировки файл будет открываться с кодировкой, заданной в системе по умолчанию, поэтому желательно указывать кодировку вручную для переносимости программ.

Ко всем описанным в этом разделе параметрам можно получить доступ через свойства, использованные в следующем примере:

```
f = with open('input.txt', 'r', encoding='UTF-8')
print(f'{f.name} {f.mode} {f.encoding}')
# > input.txt r UTF-8
```

Для проверки того, был ли открыт файл на чтение или запись используются методы `readable` и `writable`:

```
f = open('input.txt', 'r', encoding='UTF-8')
print(f.readable(), f.writable()) # > True False
```

11.2. Чтение текстового файла

В этом разделе будем рассматривать файл `'input.txt'`, который содержит текст `'Hello\nWorld\n123'`. Для экономии места обязательный для вызова метод `close` будет опускаться.

Метод `read()` читает весь файл, а `read(n)` читает `n` символов или пока не будет достигнут конец файла:

```
f = open('input.txt', 'r', encoding='UTF-8')
s = f.read() # s → 'Hello\nWorld\n123'
f2 = open('input.txt', 'r', encoding='UTF-8')
print(f2.read(5)) # > 'Hello'
```

Для чтения одной строки используется метод `readline()`, который возвращает строку с завершающимся символом `'\n'`. Метод `readlines()` считывает все строки и возвращает список строк, заканчивающиеся `'\n'`:

```
f = open('input.txt', 'r', encoding='UTF-8')
s = f.readline() # s → 'Hello\n'
```

Использование `read()` или `readlines()` считывает весь файл, что может вызвать проблемы, если файл слишком большой, поэтому предпочтительным способом чтения файлов является построчное считывание файла, описанное далее:

```
f = open('data.txt')
for line in f:  # построчное считывание
    print(line) # выводит каждую строку
```

Оператор `for` проходится по последовательностям, а, как видно из кода выше, файл представляет собой последовательность строк, поэтому список строк можно получить с помощью следующего кода:

```
list(f)
```

При работе со считанной строкой конец строки обозначается через `'\n'`. В Unix в качестве символов конца строки также используется `'\n'`, а в Windows — `'\r\n'`. Интерпретатор Python при чтении/записи в файл автоматически преобразует `'\n'` из/в зависимый от операционной системы формат.

Когда файл считывается построчно, строка `'\n'` представляет пустую строку.

В отличие от некоторых других языков программирования в Python отсутствует константа EOF, а конец файла обозначается через `''` (пустая строка). Помня о том, что пустая строка воспринимается как `False`, а не пустая — как `True`, проверку достижения конца файла можно организовать следующим способом:

```
f = open('input.txt', 'r', encoding='UTF-8')
s = f.readline()
while s:
    print(s)          # построчный вывод файла
    s = f.readline()
if not s:
    print('EOF')      # > EOF
```

11.3. Обработка считанных строк

Результатом вызова методов, считывающих строку, является строка `line`, оканчивающиеся на `'\n'`. Некоторые алгоритмы, работающие со строками, разрабатываются без учёта содержания конца строки, поэтому для удаления этого символа можно воспользоваться `line[:-1]`, `line.rstrip()` или `line.rstrip('\n')`. Последний способ предпочтительнее из-за того, что последняя строка может не содержать символа конца строки, а также в конце строки файла могут находиться другие пробельные символы.

Результатом чтения из текстового файла является строка. Поэтому для чтения целого числа требуется преобразовать считанную строку в целое число:

```
f = open('input.txt') # в файле числа записаны построчно
sum([int(line) for line in f]) # вычисляет сумму чисел
```

11.4. Запись в текстовый файл

Для записи строки в файл применяется метод `write`, а для перевода на новую строку необходимо явно указать `'\n'`:

```
f = open('data.txt', 'w')
f.write('hello\n')
```

Метод `writelines` выводит список строк, не выводя `'\n'` между ними, поэтому эти строки должны заканчиваться на `'\n'`:

```
f.writelines(['world\n'])
```

Для вывода объекта `obj` необходимо преобразовать его с помощью `str(obj)`:

```
f.write(str([1, 2, 3])) # выводится '[1, 2, 3]'
f.close()
```

По завершении всех операций записи необходимо закрыть файл.

11.5. Позиционирование в файле

При открытии файла с помощью режимов `'r'` или `'w'` текущая позиция файла устанавливается в начало файла. При чтении/записи `n` символов/байтов текущая позиция смещается на соответствующее количество символов/байтов. Поэтому в следующем коде первый вызов `read` возвращает содержимое всего файла, а второй — пустую строку, обозначающую конец файла:

```
f = open('input.txt')
print(f.read()) # выводит весь файл
print(f.read()) # выводит пустую строку
```

В этом примере после первого вызова `read` текущая позиция файла указывает на конец файла. Следующий вызов этого метода начинает считывать с конца файла, поэтому будет считана пустая строка, обозначающая конец файла.

Эту проблему можно решить путём закрытия и повторного открытия файла, что приведёт к установке позиции в файле в самое начало. Другой способ предполагает чтение всего файла в переменную `s = f.read()` с возможностью многократного обхода строки, что может быть неприемлемо, если файл слишком большой. Для решения, в частности, этой проблемы используется позиционирование, которое позволяет установить текущую позицию файла вручную с помощью метода `seek`.

В методе `seek(offset, whence=0)` параметр `offset` задаёт значение, на которое нужно сместить текущую позицию файла относительно параметра `whence`. Если `whence` принимает значение `0`, являющееся значением по умолчанию, позиция устанавливается относительно начала файла, поэтому значение `offset` должно быть не отрицательным. Если значение равно `1`, тогда смещение, которое может быть и положительным, и отрицательным, вычисляется относительно текущей позиции. Когда передаётся `whence`, равный `2`, текущая позиция находится относительно конца файла. При этом вышеописанная функциональность доступна лишь при работе с двоичными файлами, а в текстовых файлах разрешено лишь менять позицию относительно начала файла (исключением является `seek(0, 2)`). Таким образом, рассмотренный выше пример можно переписать с использованием `seek`:

```
f = open('input.txt')
print(f.read()) # выводит весь файл
f.seek(0)
print(f.read()) # выводит весь файл
```

Метод `tell` возвращает значение текущей позиции в байтах (даже для текстовых файлов):

```
# файл содержит 'абвгд...я'
f = open('alp.txt', encoding='UTF-8')
print(f.read(4)) # > абвг
print(f.tell())  # > 8
```

В этом примере были считаны 4 русских символа, каждый из которых занимает 2 байта, поэтому вызов `tell` вернёт 8.

При этом в качестве `offset` можно задавать только значения, которые для текущего файла возвращает метод `tell` (особенно для текстовых файлов). Например, в предыдущем примере нельзя указать `f.seek(1)`, потому что выбросится исключение `UnicodeDecodeError`, а `f.seek(2)` устанавливает позицию на символ `'б'`.

11.6. Буферизация

Рассмотрим подробнее, как происходит работа с файлами. Файлы хранятся на внешних устройствах, которые работают на порядки медленнее нежели оперативная память и центральный процессор. В частности, жёсткий диск является механическим устройством, в котором для доступа к нужной информации выполняется поворот дисков. Поэтому при работе с жёстким диском используется буфер, работа с которым осуществляется следующим образом. При чтении нескольких символов из файла с диска считывается сразу целый блок байтов, из которого выдаются запрашиваемые символы с последующим удалением из буфера. При следующем запросе к жёсткому диску байты извлекаются (считываются и удаляются) из буфера, а не считываются из жёсткого диска напрямую, что экономит время. Когда буфер становится пустым (из него будут считаны все символы), с диска считывается следующий блок. Это продолжается до тех пор, пока не будет достигнут конец файла. Описанный выше буфер называется системным буфером и он реализован в самой операционной системе. Работа с файлами в языках программирования осуществляется не напрямую обращением к внешнему устройству, а через вызов специальной функции операционной системы. Вызов этой функции использует системный буфер. Однако вызов функции операционной системы требует некоторых накладных расходов, поэтому разработчики языков программирования осуществляют собственную реализацию буфера (внутренний буфер). Таким образом, буферизированное чтение в Python работает следующим образом: функция `read` выполняет чтение из внутреннего буфера, куда посредством вызова функции операционной системы помещаются данные из системного буфера. Запись в файл работает схожим образом: при записи в файл данные помещаются во внутренний буфер, который при заполнении сбрасывается в системный буфер, что, однако, не ведёт к немедленному сбросу системного буфера. У операционной системы может быть размер буфера, отличный от размера буфера в программе, поэтому сброс системного буфера на диск может быть осуществлён гораздо позже.

Закрытие файла, открытого на запись, ведёт к сбросу буфера на носитель информации. Однако бывают ситуации, когда необходимо сбросить буфер вручную. Рассмотрим код, в котором в цикле на каждой итерации выполняются очень сложные вычисления, результат которых записывается в файл:

```
for i in range(10**6):
    # очень долгие вычисления
    f.write(result)
```

Пользователь, желающий просмотреть временные данные вычислений, должен ждать, пока интерпретатор не сбросит внутренний буфер в системный, а операционная система не сбросит системный буфер, что может не произойти до завершения программы. Для сброса внутреннего буфера вручную необходимо воспользоваться методом `f.flush()`. Как было

отмечено ранее, это не гарантирует сброса системного буфера на жёсткий диск. Чтобы убедиться, что буфер будет сброшен на диск, необходимо вызвать функцию `fsync` из модуля `os`:

```
import os
for i in range(10**6):
    # очень долгие вычисления
    f.write(result)
    f.flush()
    os.fsync(f.fileno())
```

Метод `fileno()` возвращает файловый дескриптор операционной системы.

Для того чтобы выяснить размер буфера по умолчанию, можно воспользоваться следующим кодом:

```
import io
print(io.DEFAULT_BUFFER_SIZE) # > 8192
```

11.7. Работа с двоичными файлами

Для открытия файла как двоичного требуется указать в режиме открытия суффикс `'b'`, например, `open('rb')`. Параметр `encoding` для двоичных файлов, разумеется, не задаётся.

При чтении из двоичного файла всегда возвращается массив байтов (класс `bytes`), а для записи следует передавать массив байтов. Метод `read()` считывает весь файл, а `read(n)` возвращает `n` байтов (если в файле остаётся менее `n` байтов, возвращаются эти байты). При достижении конца файла `read` возвращает пустой объект `bytes`, который при переводе в булевский тип даёт `False`. Метод `write(a)` записывает массив байтов `a` в файл. В качестве примера рассмотрим копирование из одного файла в другое по одному байту:

```
f1 = open('a.txt', 'rb')
f2 = open('b.txt', 'wb')
a = f1.read(1)
while a:
    f2.write(a)
    a = f1.read(1)
```

Для двоичных также можно задать размер буфера с помощью параметра `buffering`.

11.8. Закрытие файла

Наиболее важной функциональностью файлов для программиста является возможность чтения/записи в файл, но не менее важным является закрытие файла. Как было описано ранее, при закрытии файла происходит сброс буфера. Таким образом, если не закрыть файл, то в случае, если программа работает постоянно без завершения выполнения (например, веб-приложение), буфер может никогда не сброситься. Поэтому обязательно должно быть выполнено закрытие файла (вызов метода `close`) сразу по завершении всех операций чтения из этого или записи в него.

Хотя в некоторых реализациях языка Python сборщик мусора может вызвать метод `close` при удалении объекта, работающего с файлами, однако это свойство не является частью стандарта языка, поэтому необходимо следить, чтобы все открытые файлы были закрыты.

11.9. Диспетчеры контекста для работы с файлами

Рассмотрим следующий код:

```
f = open('input.txt')
a = int(f.readline())
f.close()
```

Предположим, что в первой строке файла `'input.txt'` находится строка, не являющаяся числом. Тогда при попытке преобразовать считанную строку произойдёт исключение. Вследствие этого строка, идущая за вызвавшей исключение строкой, не выполнится, то есть файл закрыт не будет. Для решения этой проблемной ситуации в Python используется *диспетчер контекста* (*context manager*), который применяет операторы `with`/`as`. Пример использования диспетчера контекста рассмотрен далее:

```
with open('input.txt') as f:
    a = int(f.readline())
```

В этом примере открывается файл `'input.txt'`, а ссылка на открытый файл присваивается в переменную `f`. После окончания оператора `with` (строка, идущая после оператора и имеющая такой же отступ) происходит автоматическое закрытие файла `f`. При этом, закрытие файла происходит как при штатном завершении оператора `with` (без исключений), так и после исключений. В последнем случае произойдёт закрытие файла без обработки исключения, которое будет проброшено дальше. Внутренняя реализация оператора `with` аналогична следующему коду:

```
f = open('input.txt')
try:
    f.read()
finally:
    f.close()
```

Вспомним, что `finally` выполняется как при возникновении исключения, так и при его отсутствии. Само же исключение не обрабатывается ввиду отсутствия `except`.

В диспетчере контексте можно открыть сразу несколько файлов в разных режимах, которые по завершении оператора `with` будут все закрыты:

```
with open(...) as f1, open(...) as f2:
    pass
```

Аналогами диспетчера контекстов в `C#` является `using`, в `C++` — RAII.

Просуммировав вышенаписанное, диспетчер контекста является обязательным для использования при работе с файлами.

11.10. Хранение объектов Python в файлах

Рассмотрим задачу хранения объекта(ов) Python в файле, например, хранение словаря. Можно реализовать запись и чтение словаря вручную, применяя методы описанные ранее. Но это не является самым эффективным способом. В этом разделе рассмотрим более эффективные и удобные альтернативные способы.

Предположим в файле список `a = [1, 2, 3]` записан в файл с помощью `str(a)` и хранится в виде `'[1, 2, 3]'`. Для чтения этого списка можно воспользоваться функцией `eval()`, которая выполняет параметр-строку как код и возвращает результат:

```
b = eval(f.readline()) # b → [1, 2, 3]
```

С функцией `eval` нужно быть очень осторожным, так как в файле может быть вредоносный код.

Предпочтительным способом для хранения объектов в файле является сериализация, описанная в следующем разделе.

11.10.1. Сериализация

Сериализация — процесс преобразования объекта в вид, удобный для хранения (например, в файле) или передачи (по сети). Результатом сериализации чаще всего является массив байтов. *Десериализация* — обратный сериализации процесс: преобразование массива байтов в объект. Для сериализации в Python применяются стандартные модули `pickle` и `shelve`. Сериализация выполняется с помощью функции `dump` следующим образом:

```
import pickle
a = [1, 2, 3]
with open('data.pkl', 'wb') as f:
    pickle.dump(obj, a) # сериализация
```

Стоит заметить, что файл для записи объекта открывается как двоичный. Для десериализации записанных объектов применяется следующий код с функцией `load`:

```
with open('data.pkl', 'rb') as f: # десериализация
    b = pickle.load(f) # b → [1, 2, 3]
```

Модуль `shelve` позволяет сохранять и считывать объекты по ключам с помощью `pickle`, то есть модуль `shelve` предоставляет поведение, похожее на словари:

```
with shelve.open('input') as db:
    db['list'] = [1, 2, 3]
with shelve.open('input') as db:
    a = db['list'] # a → [1, 2, 3]
```

Однако с модулями `pickle` и `shelve`, как и с функцией `eval`, нужно быть очень осторожным и не работать с файлами, полученными из непроверенных источников.

Научившись работать с файлами, в дальнейшем можно легко использовать инструменты, подобные файлам: стандартные потоки данных (`sys.stdout`), конвейеры, сокеты, интерфейсы к базам данным и так далее.

Дополнительно в Python есть модули и библиотеки для работы с файлами: `fileinput` — для быстрой обработки нескольких файлов, `csv` — для работы с csv-файлами.

11.11. Резюме

Python позволяет работать с двоичными и текстовыми файлами. Для эффективной работы файлы при чтении и записи буферизируются, а для явного сброса буфера нужно воспользоваться методами `flush` и `os.fsync`. Во время чтения и записи автоматически изменяется текущая позиция в файле, которую можно установить вручную.

Для открытия файлов предпочтительнее использовать диспетчер контекста (оператор `with`/`as`), автоматически закрывающий файл, и явное указание кодировки в функции `open` при помощи параметра `encoding`. Для чтения файла оптимальным вариантом является построчное считывание с помощью `for`. При чтении/записи символ `'\n'` автоматически преобразуется в символ, зависящий от операционной системы, а данные из файла — в кодировку Unicode.

Сериализация предоставляет удобный и простой способ для хранения объектов в файле. С помощью модулей `pickle` и `shelve` можно осуществить сериализацию в Python.

Список литературы

1. Лутц, М. Изучаем Python [Текст] : в 2 т. / М. Лутц. — 5-е изд. — СПб. : ООО «Диалектика», 2019–2020. — 2 т.

2. Любанович, Б. Простой Python. Современный стиль программирования [Текст] / Б. Любанович. — СПб. : Питер, 2018. — 480 с.

3. Гуриков, С. Р. Основы алгоритмизации и программирования на Python: учеб. пособие [Текст] / С.Р. Гуриков. — М. : ФОРУМ : ИНФРА-М, 2018. — 343 с.

4. Прохоренок, Н. А. Python 3. Самое необходимое: Пособие [Текст] / Н.А. Прохоренок, В.А. Дронов. — СПб. : БХВ-Петербург, 2016. — 464 с.

5. Прохоренок, Н. А. Python 3 и PyQt 5. Разработка приложений: Пособие [Текст] / Н.А. Прохоренок. — СПб. : БХВ-Петербург, 2016. — 833 с.

6. Седер, Н. Python. Экспресс-курс [Текст] / Н. Седер. — 3-е изд. — СПб. : Питер, 2019. — 480 с.

7. Welcome to Python.org. — URL: `https://python.org/` (дата обращения: 01.12.2020).

8. Python 3.9.1 documentation. — URL: `https://docs.python.org/3/index.html` (дата обращения: 01.12.2020).

9. The Python Tutorial. — URL: `https://docs.python.org/3/tutorial/index.html` (дата обращения: 01.12.2020).

10. Using Python on Windows. — URL: `https://docs.python.org/3/using/windows.html` (дата обращения: 01.12.2020).

11. Expressions. — URL: `https://docs.python.org/3.9/reference/expressions.html` (дата обращения: 01.12.2020).

12. Built-in Types. — URL: `https://docs.python.org/3.9/library/stdtypes.html` (дата обращения: 01.12.2020).

13. PyPI · The Python Package Index. — URL: `https://pypi.org/` (дата обращения: 01.12.2020).

14. Anaconda | The World's Most Popular Data Science Platform. — URL: `https://www.anaconda.com` (дата обращения: 01.12.2020).

15. Python(x,y) - the scientific Python distribution. — URL: `https://python-xy.github.io/` (дата обращения: 01.12.2020).

16. Pyzo. — URL: `https://pyzo.org/` (дата обращения: 01.12.2020).

17. PyPy. — URL: `https://www.pypy.org/` (дата обращения: 01.12.2020).

18. Home · stackless-dev/stackless Wiki · GitHub. — URL: `http://www.stackless.com/` (дата обращения: 01.12.2020).

19. py2exe. — URL: `https://www.py2exe.org/` (дата обращения: 01.12.2020).

20. Welcome to cx_Freeze's documentation!. — URL: `https://cx-freeze.readthedocs.io` (дата обращения: 01.12.2020).

Учебное пособие

Еникеев Разиль Радикович

ОСНОВЫ И БАЗОВЫЕ ТИПЫ ДАННЫХ В PYTHON