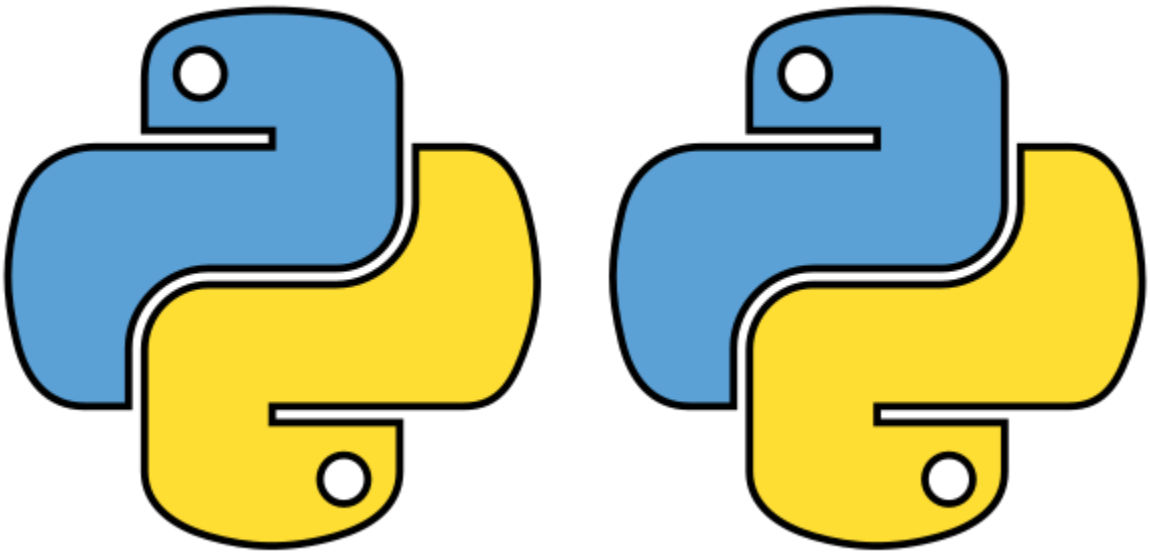
Д. Е. Шипило, А. А. Коновко,
А. А. Лукашёв, Н. А. Панов

Язык программирования Python

Семестр 3 — после C++



Москва

Физический факультет МГУ им. М.В. Ломоносова

2024

Ш и п и л о Д. Е., К о н о в к о А. А., Л у к а ш ё в А. А., П а н о в Н. А. Язык программирования Python. Семестр 3 — после C++. — М.: Физический факультет МГУ им. М.В. Ломоносова, 2024.

ISBN 978-5-8279-0293-5

Учебное пособие знакомит читателей с популярным в настоящее время языком программирования Python. Изложение материала предполагает знакомство с основами программирования на языке C++, который студенты изучают на 1-м курсе. Описание базовых синтаксических элементов, основных типов данных и классов Python основано на подробном сравнении с аналогичными конструкциями C++. Обсуждаются особенности работы с Python как с интерпретируемым языком: динамическая типизация, коллекции данных, итерационный протокол, списковые включения, «ленивые» вычисления и т. д. Приведена информация о наиболее часто использующихся модулях. Изложены сведения о библиотеках `numpy`, предназначенной для работы с массивами числовых данных, и `sympy`, ориентированной на символьные вычисления.

Учебное пособие предназначено для студентов 2-го курса физического факультета МГУ имени М. В. Ломоносова, изучающих язык программирования Python, а также для преподавателей, ведущих занятия по дисциплинам модуля «Информатика и вычислительная физика».

Рецензенты:

к.ф.-м.н. *А. И. Шувалова*, специалист АО «Тинькофф Банк».

к.ф.-м.н. *И. Е. Могилевский*, доцент кафедры математики физического факультета МГУ.

Печатается по плану, утвержденному редакционно-издательским советом и Ученым советом физического факультета Московского государственного университета имени М.В. Ломоносова.

ISBN 978-5-8279-0293-5

Содержание

Часть I. Базовый синтаксис

1. Введение

Преподавание программирования на Физическом факультете МГУ на младших курсах строится на трех основах:

- алгоритмы,
- структуры данных,
- языки программирования.

В первом семестре на базе языка программирования C обсуждаются такие фундаментальные языковые элементы как типы данных, указатели, массивы, структуры, перечисления и т. д.. Обсуждаются способы представления чисел в машинной арифметике. Рассматриваются такие фундаментальные алгоритмы как поиск, сортировка, методы численного решения уравнений, дифференцирования и интегрирования. Обсуждается понятие сложности алгоритма. Во втором семестре на базе языка C++ изучаются понятия ссылочных переменных и контейнеров (стек, очередь, дека, вектор, список), концепции объектно-ориентированного и обобщенного программирования.

Изученные на первом курсе алгоритмы и структуры данных к настоящему времени превратились в рутинный инструмент разработчика. Поэтому в ряде языков программирования такие операции как, например, поиск и сортировка реализованы в виде стандартных конструкций. Это обстоятельство позволяет второкурсникам сосредоточиться не на записи стандартных алгоритмов, а на освоении нового языка и программировании моделей физических явлений и эффектов.

В качестве такого языка можно выбрать Python (Па́йтон), который по состоянию на 2024 год уверенно входит в число самых популярных языков программирования, по многим рейтингам занимая и первое место. Среди составляющих такого успеха можно назвать простоту освоения, лаконичность кода (что ускоряет разработку и поддержку программ) и впечатляющую универсальность: огромный набор расширений-модулей Python позволяет применять его в областях от web-разработки до обработки результатов социальных опросов. Немало последователей у этого языка и в областях, подготовкой кадров для которых занимается Физический факультет МГУ. Если мы попытаемся классифицировать навыки программирования в современной физике, то получим примерно следующую картину:

- обработка (в том числе статистическая) и визуализация данных — Python видится почти идеальным инструментом для подобных задач, пока не упирается в предел скорости обработки;
- численное моделирование — пока речь не идет о задачах, которые описываются термином high-performance computing, Python предлагает достойные решения;
- символьные вычисления — Python предоставляет неплохие решения, хотя и проигрывает специализированным системам компьютерной алгебры;
- автоматизация эксперимента — с помощью соответствующих модулей из Python можно управлять внешними устройствами.

Таким образом, Python предоставляет очень много возможностей, дополняя их удобством разработки, отладки и тестирования.

В каждой из обозначенных областей существует порог вычислительной сложности задачи, начиная с которого Python будет недостаточен с точки зрения производительности, и придется переходить к более узкоспециализированным инструментам. Не является

каким-то экзотическим сценарий, в котором алгоритм разрабатывается и тестируется на Python, а для реального применения переводится на компилируемый язык, целиком либо же отдельными блоками, наиболее требовательными к ресурсам.

Настоящее пособие призвано ознакомить читателя с базовым синтаксисом Python и, частично, с его внутренней логикой, для чего мы проводим аналогии с уже известным студентам языком программирования C++. Обсуждается использование ряда модулей, полезных при решении физических задач. В сообществе разработчиков на Python достаточно много внимания уделяется «стандартам», призванным облегчить читаемость, сопровождение и поддержку программ (Python Enhancement Proposals, PEP 8 https://peps.python.org/pep-0008 и ряд других). Мы старались приводить примеры, которые соответствуют PEP, но нигде не обсуждаем его как таковой.

2. Начало работы

Существенным преимуществом Python является свободное распространение. Скачать дистрибутив Python можно на сайте https://www.python.org/. Инструкции по установке для различных операционных систем можно найти на странице

https://wiki.python.org/moin/BeginnersGuide/Download

На Windows Python устанавливается вместе с достаточно минималистической средой разработки IDLE (Integrated Development and Learning Environment). Существуют и другие среды разработки, которые могут быть удобнее. PyCharm, Spyder и Geany предоставляют синтаксический анализатор, автодополнение и т. д. Jupiter Notebook — браузерная версия Python на базе Google. Anaconda — профессиональный пакет с большим числом библиотек и средств разработки, хотя, пожалуй, слишком тяжеловесный для учебных задач.

После того, как Python и среда разработки установлены, можно переходить к работе. Во-первых, поскольку Python является интерпретатором, а не компилируемым языком программирования, он не требует полного кода программы, и ее можно писать «на ходу», в так называемом режиме интерактивной подсказки. Этот режим доступен во многих средах разработки, но можно обойтись и без него, работая в консоли. Чтобы запустить Python из командной строки, необходимо

- под Windows: выполнить команду `python` в командной строке (`cmd.exe`) или PowerShell;
- под Linux: выполнить команду `python3` в терминале;
- под MacOS: выполнить команду `python` в терминале.

Во-вторых, можно выполнить и всю программу, то есть последовательность команд, записанную в текстовый файл с расширением `.py`, выполнив команду

```
python my_program.py
```

либо открыв соответствующий файл в среде разработки и запустив интерпретатор.

Большая часть примеров, приведенных ниже, может быть выполнена в интерактивной подсказке, хотя более длинные программы удобнее писать, редактировать и выполнять с помощью среды разработки. Интерактивная подсказка также представляет собой прекрасный инструмент для проверки элементарных шагов. Мы крайне рекомендуем читать это пособие, повторяя и, что даже более важно, расширяя и изменяя примеры, приведенные здесь. Во-первых, ничто не обучает так, как чтение ошибок и предупреждений. Во-вторых, почти по всем методам в Python можно получить справку с помощью команды `help`, чтобы ознакомиться с формальным описанием.

Примеры из данного пособия писались и тестировались с использованием Python 3.11.1 на Windows, Python 3.9.2 на Linux и Python 3.11.4 на MacOS, однако едва ли хоть какие-то из них поменяют свое поведение на других системах и версиях.

3. Объекты Python. Числа и строки

Начнем с того, как программа на языке Python хранит данные в оперативной памяти (в конце концов, программы вообще именно создают и обрабатывают данные). Любые данные при выполнении программы Python существуют в виде объектов, то есть экземпляров классов с *атрибутами* — полями и методами. В отличие от программы на C++ и многих других языках программирования, для которой, например, число с плавающей точкой двойной точности — это выделенные в оперативной памяти 8 байт, о которых точно известно, какой они имеют тип и с какой переменной в коде ассоциированы, в Python число с плавающей точкой — это прежде всего объект, содержащий

- ссылку на тип, которому принадлежит этот объект (`float`),
- счетчик ссылок на объект,
- 8 байт непосредственно данных[1],

всего 24 байта. Тип объекта определяет его атрибуты, доступные с помощью синтаксиса `obj.attr`. Например, рассмотрим атрибуты числа с плавающей точкой:

```
>>> 20.125.real                  # поле
20.125
>>> 20.125.imag
0.0
>>> 20.125.as_integer_ratio()    # метод (вызывается)
(161, 8)
>>> 20.125.hex()                 # 16-ричное представление
'0x1.4200000000000p+4'
>>> 20.125.is_integer()
False
>>> 20.125.__sizeof__()          # размер объекта в байтах
24
```

(Символ # означает комментарий. Всё, что находится после него до конца строки, не воспринимается интерпретатором.) Во-первых, обратим внимание, что здесь в каждой команде первая точка — это разделитель целой и дробной части числа 20.125, а вторая — синтаксический элемент, который сообщает интерпретатору, что надо обратиться к атрибуту объекта слева от точки. Во-вторых, напомним, что методами традиционно называются атрибуты-функции, которые необходимо вызывать с помощью скобок `()` (в частном случае — без аргументов), полями — содержащиеся в объекте данные. В-третьих, опередим повествование и перечислим возвращаемые типы в данном примере: поля `real` и `imag` являются числами с плавающей точкой, как и исходный объект, метод `as_integer_ratio` вернул кортеж из двух целых чисел, `hex` — строку, а `is_integer` — логическое выражение.

Посмотреть все атрибуты объекта можно, передав его встроенной функции `dir`:

```
>>> dir(20.125)
```

Для экономии места не будем приводить здесь вывод интерпретатора. В нем можно найти атрибуты из предыдущего примера, а также около 50 имен, начинающихся

[1] Обратите внимание, что это соответствует типу `double` в C++. Чисел же одинарной точности в Python нет.

и заканчивающихся двумя символами подчеркивания, в частности, `__abs__`. Такие атрибуты указывают интерпретатору, что над объектом разрешено то или иное действие, и вызываются автоматически, когда объект передается соответствующей встроенной функции или оператору. Так, следующие команды попарно эквивалентны:

```
>>> abs(-20.125)
20.125
>>> (-20.125).__abs__()              # (i)
20.125
>>> 20.125 * 8.
161.0
>>> 20.125.__mul__(8.)
161.0
>>> 20.125 // 0.1                    # (ii)
201.0
>>> 20.125.__floordiv__(0.1)
201.0
>>> 20.125 ** 0.5                    # (iii)
4.4860896112315904
>>> 20.125.__pow__(0.5)
4.4860896112315904
>>> type(20.125)                     # (iv)
<class 'float'>
>>> 20.125.__class__
<class 'float'>
```

Прокомментируем, что (i) скобки вокруг отрицательного числа здесь необходимы, поскольку унарный минус имеет более низкий приоритет, чем обращение к атрибутам (без скобок мы бы получили результат −20.125), (ii) оператор `//` в Python 3 осуществляет деление нацело, в то время как деление `/` работает как «настоящее» деление (так и называется, `__truediv__`) даже с целыми числами, (iii) оператор `**` осуществляет возведение в степень[2], (iv) функция `type` возвращает тип или класс объекта (тип — это встроенный класс, и различать эти термины мы по ходу повествования не будем). Для проверки того, является ли объект экземпляром заданного класса (строго говоря, или класса-наследника), обычно используют встроенную функцию `isinstance`:

```
>>> isinstance(20.125, float)
True
```

Мы не будем подробно разбирать все атрибуты какого-либо типа. Они в любой момент доступны с помощью функции `dir`, их описание можно получить с помощью функции `help` (например, `help(float.__round__)`) или в подробной документации. Многие общие операторы перечислены в Приложении А.

Помимо чисел с плавающей точкой двойной точности (`float`), в Python существуют и другие числовые типы. Целые числа (`int`) поддерживают бесконечную точность:

```
>>> 3**118
19966781110160346782368664772328944885905284750420567848 9
```

[2] Возведение в степень имеет правую ассоциативность, то есть выражение `a ** b ** c` допускает группировку `a ** (b ** c)`. Например, в выражении `5 ** 4 ** 3` сначала выполняется правое возведение в степень `4 ** 3`, и потом — левое, `5 ** 64`. Для большинства других операторов порядок слева направо, например, `a / b / c` эквивалентно `(a / b) / c`.

а также побитовые операции (сдвиг влево <<, вправо >>, побитовые и &, или |, исключающее или ^, инверсию ~), представление в двоичной bin(), восьмеричной oct(), шестнадцатеричной hex() системах счисления:

```
>>> (127 << 6) & 127
64
>>> oct((127 << 6) & 127)
'0o100'
```

Ввод в соответствующих системах счисления осуществляется с помощью префиксов 0b (двоичная), 0o (восьмеричная), 0x (шестнадцатеричная):

```
>>> 0b1111 + 0o17 + 0xf          # 15 + 15 + 15
45
```

Комплексные числа (complex) базируются на представлении чисел с плавающей точкой и имеют двойную точность:

```
>>> (3+1j)**118
(9.644469823599818e+58+2.642764049565135e+58j)
```

Логический тип bool принимает два значения, True и False. В выражениях они будут преобразованы в целые 1 и 0, соответственно.

Запись операторов сравнения <, >, <=, >=, ==, != в Python идентична C++. Все они возвращают значение типа bool. Операторы <, >, <=, >= не работают для комплексных чисел, а точное равенство и неравенство (== и !=) не следует применять для чисел двойной точности (как вещественных, так и комплексных).

Обратимся теперь ко встроенному классу str, то есть строкам. Задавать строки в программе можно как с помощью одинарных ' ', так и двойных " " кавычек. При выборе одного из этих двух типов кавычек для обрамления строки другой тип кавычек можно использовать внутри строки на правах обычных символов. Внутри строки допускаются спецсимволы переноса строки \n, табуляции \t и т. д., мета-символы %d, %g, %s и т. д., на место которых предполагается вставить целое число, вещественное число, строку и т. д. Одиночный символ не образует отдельного типа и считается строкой длины 1. Разобраться с операциями, доступными для строк, нам также поможет функция dir('object'). Прежде всего, обнаружим там __add__ и, немного неожиданно для нас, __mul__ и __mod__, то есть «сложение», «умножение» и «остаток от деления»:

```
>>> 'My ' + 'object'          # конкатенация
'My object'
>>> 'object ' * 3             # повтор (работает только
'object object object '       # str * int или int * str)
>>> 'object No %d' % 3        # подстановка вместо %d
'object No 3'
```

Не будем останавливаться на сравнениях строк (работающих в соответствии с кодировкой Unicode[3]) и перейдем к более важным и более общим __len__, __contains__ и __getitem__. Соответствующий синтаксис:

```
>>> len('object')             # длина строки
6
```

[3] Не только строки, но и Python в целом поддерживает Unicode. В том числе имена переменных могут содержать символы кириллицы или даже глаголицы. Но таких примеров в нашем пособии не будет, потому что крайне неудобно работать, постоянно переключая раскладку клавиатуры.

```
>>> 'ob' in 'object'        # вхождение подстроки
True
>>> 'Je' in 'object'        # 'object'.__contains__('Je')
False
>>> 'object'[0]             # индексация
'o'
>>> 'object'[1]
'b'
>>> 'object'[3]
'e'
```

Индексация начинается с 0, как и у массивов в C++. Python допускает использование в качестве индекса отрицательного числа. К отрицательному индексу прибавится длина строки, то есть последний символ может быть получен с помощью любого из обращений:

```
>>> 'object'[-1]                    # 't'
>>> 'object'[len('object')-1]       # тоже 't'
```

Аналогично `'object'[-2]` — это `'c'` и т. д. Вообще говоря, оператор `[ ]` служит еще и для срезов строки (выделения подстроки), но подробное описание соответствующего синтаксиса мы отложим до главы 9 «Объекты `range` и `slice`. Срезы последовательностей».

Помимо перечисленных методов-операторов, есть несколько десятков строковых методов наподобие

```
>>> 'string'.upper()    # приведение к верхнему регистру
'STRING'
>>> 'string string'.find('r')   # поиск подстроки слева
2
>>> 'string string'.rfind('r')  # и справа
9
>>> 'string string'.replace('ri', 'RO')
'stROng stROng'                 # замена подстроки
>>> 'string str s'.count('s')   # посчитать вхождения
3                               # подстроки
```

Передав тот или иной метод функции `help`, можно узнать, например, что `find` возвращает `-1`, если подстрока не найдена, а `replace` можно настроить, чтобы заменить лишь первые `n` вхождений подстроки. Все эти методы не сложны и хорошо задокументированы, поэтому мы не будем описывать их поштучно и подробно, а лишь призовем вспомнить о них, когда придется работать со строками. Очень многие задачи, которые в C++ решались бы посимвольным перебором строки, в Python стоит решать, используя встроенные методы.

Любой объект может быть преобразован в строку. Именно это происходит, когда интерактивная подсказка выводит результат выражения и когда вызывается функция `print(object)`. За такое преобразование отвечают методы `__str__` и `__repr__`. Для чисел между ними нет никакой разницы, для строк же ее достаточно легко продемонстрировать:

```
>>> print(str('object'))
object
>>> print(repr('object'))
'object'
>>> print(repr(repr('object')))
"'object'"
```

Предполагается, что строку, которую вернула функция `repr` (и метод `__repr__`), можно подставить в код Python и получить точно такой же объект, который был передан функции изначально. Для этого у строк функция `repr` проставляет явным образом кавычки — без них интерпретатор не поймет, что имеет дело со строкой. В свою очередь, «вывод для пользователя» `print` не обязан поддерживать эту эквивалентность и выводит именно содержимое строки. Данное правило, впрочем, лишь при некоторых условиях соблюдается для нестандартных типов, объявленных в расширениях-модулях, и абсолютно не соблюдается для встроенных функций:

```
>>> print(repr(len))
<built-in function len>
```

(полученную строку невозможно подставить в код).

Консольный ввод в Python осуществляется функцией `input`, возвращающей строку, которую можно преобразовать в числа, применив явное преобразование типа (с помощью конструктора[4]):

```
>>> s = input('Write a power: ')
Write a power: 5
>>> s                          # s — строка
'5'
>>> p = int(s)                 # p — целое
>>> 2 ** p
32
```

Можно записать этот пример и короче:

```
>>> 2 ** int(input('Write a power: '))
Write a power: 5
32
```

4. Переменные Python

В предыдущей главе мы упоминали, что любые данные в Python существуют в виде объектов в оперативной памяти, содержащих, помимо смысловых данных, ссылку на тип объекта и счетчик ссылок на него. Поскольку тип любого объекта указан в нем самом, интерпретатору не требуется априорной информации о типе, а только положение объекта в оперативной памяти. Поэтому переменные (имена объектов, ссылки) в Python представляют собой просто указатель на область в оперативной памяти, не привязаны к какому-либо типу и не требуют заблаговременного объявления:

```
>>> x = 20.24              # тип объекта float
>>> x = "string"           # тип объекта str
>>> x = len(x)             # тип объекта int
```

В каждой из трех строчек имя `x` привязывается к объекту, который создан в результате вычисления выражения справа от оператора присваивания `=`. Сам `x`, таким образом, из синтаксиса C++ более всего похож на указатель неопределенного типа `void*`. Но поскольку в выражениях этот указатель всегда разыменовывается (то есть в последней строчке примера вызывается метод `__len__` объекта `"string"`, а не самого указателя), в терминах C++ переменная `x` является ссылкой `void&` на объект неопределенного

[4] Конструктор, то есть метод, вызываемый при создании объекта, функционирует аналогично таковому в C++.

типа[5]. Числовое значение указателя можно посмотреть, вызвав `id(x)`, однако использовать полученное целое число в качестве указателя невозможно.

Фундаментальное отличие присваивания в Python от такового в C++ заключается в том, что в Python эта операция затрагивает только имя-ссылку, и никогда — объект, который был доступен по ссылке. Рассмотрим следующий пример:

```
>>> x = 2024
>>> ix = id(x)
>>> x = x + 5508
>>> id(x) == ix
False
```

В результате присваивания, как мы видим, ссылка `x` изменилась. Казалось бы, перевод этого примера на C++ должен выглядеть следующим образом (здесь и далее редкие фрагменты кода на C++ будем обозначать вертикальной чертой слева):

```
int x = 2024;
int* ix = &x;
x = x + 5508;
cout << (&x == ix) << endl;     // Вывод: 1
```

но результат противоположный: в C++ присваивание `x = x + 5508` не меняет адреса `x`. Смысловой же перевод будет таким:

```
int* x = new int(2024);
int* ix = x;
x = new int(*x + 5508);
cout << (x == ix) << endl; // Вывод: 0
```

В результате вычисления `x + 5508` Python создает новый объект, число 7532. Присваивание перенаправляет ссылку `x` на этот новый объект, а не изменяет старый объект, число 2024. Более того, числовые и строковые объекты в Python вообще неизменяемые: из трех полей (тип, счетчик ссылок, данные) в таких объектах меняется только счетчик ссылок. С изменяемыми объектами мы познакомимся немного позже, а о том, почему важно иметь в виду разницу между изменяемыми и неизменяемыми объектами, поговорим в главе 12 “Неочевидные следствия модели «объект—переменная» в Python”. Пока лишь скажем, что наиболее значительные отличия заключаются в использовании инкрементных операций `+=`, `-=` и т. д. Эти операторы существуют в Python, но в нашем последнем примере нет никакой разницы, используем мы команду `x = x + 5508` или `x += 5508` — именно потому что числа неизменяемы.

Теперь, пожалуй, уместно задать вопрос об очистке памяти. Действительно, память под новые объекты выделяется динамически. Например, неужели тот объект 2024 из прошлого примера, ссылку на который мы утратили, когда переприсвоили `x`, будет вечно существовать в оперативной памяти? Именно чтобы этого избежать, в каждом объекте существует счетчик ссылок на этот объект. Не вдаваясь в подробности работы с модулями, скажем, что посмотреть этот счетчик можно, используя функцию `getrefcount` из стандартного модуля `sys`:

[5] В C++ ссылки — это указатели, для которых гарантировано наличие объекта по указателю и которые автоматически разыменовываются в выражениях. В C++ конструкция `void&` синтаксически некорректна, поскольку компилятор не может разыменовать указатель неопределенного типа. В Python тип известен интерпретатору, потому что ссылка на него включена в структуру *любого* объекта.

```
>>> import sys
>>> x = 4048
>>> y = x
>>> sys.getrefcount(x)
3
```

Поскольку объект справа от оператора присваивания — это объект по указателю `x`, то `y` ссылается на тот же самый объект. Итого на него ссылаются три переменных: `x`, `y` и одна внутренняя переменная функции `getrefcount` (она неизбежна, но пропадает, как только функция завершается). Если удалить переменную `x`, счетчик уменьшится на 1:

```
>>> del x
>>> sys.getrefcount(y)
2
```

Если после этого удалить еще и `y`, объект останется без ссылок и будет удален специальным процессом — сборщиком мусора. Таким образом, объекты, которые «больше не нужны», удаляются из памяти автоматически. Подчеркнем, что оператор `del` не удаляет объект и не освобождает память, а только удаляет имя, доступное программе.

Оператор `is` проверяет, что переменные ссылаются на один и тот же объект, иными словами, равенство указателей, а не объектов:

```
>>> x = 4048            # int* x = new int(4048);
>>> y = x               # int* y = x;
>>> x is y              # cout << (x == y) << endl;
True
```

Ни в коем случае не следует использовать `is` в качестве оператора сравнения ==. Если справа от оператора присваивания стоит выражение, то при его вычислении будет создан новый объект, и ссылка, вообще говоря, будет другой:

```
>>> z = x * 1           # int* z = new int(*x * 1);
>>> z == x              # cout << (*x == *z) << endl;
True
>>> z is x              # cout << (x == z) << endl;
False
```

Это означает, что создание объекта (оператор `new` в C++) осуществляется при выполнении умножения, а не присваивания.

Несколько заключительных на данном этапе комментариев по поводу оператора присваивания. Во-первых, присваивание цепочкой возможно только в виде

```
x = y = z = expression
```

в отличие от C++, где, например, возможно выражение

```
x = (y += 5) + (z = 20);
```

Внутри выражения `expression` не может быть команд присваивания, инкрементных операций типа +=, -=, удаления, импорта и т. п. Во-вторых, в главе 13 «Распаковка коллекций» мы расскажем о дополнительном функционале оператора присваивания, который делает его гораздо более гибким инструментом.

5. Коллекции. Списки

Общим термином «коллекция» называют класс или объект, который «содержит» в себе другие объекты и предоставляет к ним доступ. Наиболее простой коллекцией является массив — последовательность данных одного типа. Другими примерами коллекций могут служить стек и очередь, контейнеры стандартной библиотеки шаблонов STL. В случае

Python с его динамической типизацией слово «содержит» должно пониматься строго в кавычках — коллекции работают со ссылками на другие объекты. При этом коллекции могут содержать объекты разных типов, имеют произвольный размер и допускают произвольную вложенность (разумеется, в пределах доступной оперативной памяти). В Python реализованы пять типов коллекций, которые можно разделить на три разновидности. Списки `list` и кортежи `tuple` представляют собой упорядоченные последовательности, изменяемые и неизменяемые, соответственно. Множества `set` и неизменяемые множества `frozenset` представляют собой неупорядоченные коллекции уникальных элементов. Словари `dict` содержат пары «ключ—значение» с уникальными ключами, т.е. ключи словаря образуют множество.

На текущем этапе познакомимся только со списками `list`, которые ближе всего к массивам языков со статической типизацией, а также шаблонному классу `std::vector` библиотеки STL. Остальные коллекции опишем позднее, в главе 11 «Коллекции. Кортежи, множества и словари», а в Приложении Б перечислим наиболее важные атрибуты каждого из типов.

В наиболее простом варианте список инициализируется путем перечисления его элементов в квадратных скобках:

```
>>> L = [1, 2, 3.,              # int, int, float
...  'list', 5.5, -2-2j]        # str, float, complex
>>> L
[1, 2, 3.0, 'list', 5.5, (-2-2j)]
```

(Пользуясь случаем, скажем про «многострочные выражения» в Python. Обычно код пишется в режиме «одна строка — одна команда», но если к концу строки Python видит незакрытую скобку, круглую, квадратную или фигурную, он воспринимает следующие строки как продолжение текущей, пока скобка не будет закрыта.)

Если бы мы хотели получить что-то вроде списка `L` в C++, мы бы создали массив указателей `void*`:

```
#include <iostream>
using namespace std;

int main()
{
    void **L;
    L = new void*[6];
    L[0] = new int(1);
    L[1] = new int(2);
    L[2] = new double(3.0);
    // и так далее
    cout << *((int*)L[0]) << endl << *((int*)L[1])
         << endl << *((double*)L[2]);
    delete (int*)L[0];
    delete (int*)L[1];
    delete (double*)L[2];
    // и так далее
    delete L;
}
```

В такой массив можно было бы сложить объекты произвольных типов. Из этого примера вполне очевидно, что работа с такой структурой существенно осложнена тем, что необходимо запоминать соответствующие типы элементов, даже чтобы просто вывести содержимое `L` на экран. В Python же тип хранится в самом объекте.

Однако вернемся к спискам Python. Подобно строкам, разрешена конкатенация и повторение списков с помощью перегруженных операторов `+` и `*`, можно запрашивать отдельные элементы и срезы через оператор `[ ]` (нумерация элементов начинается с 0, подробно о срезах будет рассказано в главе 9 «Объекты `range` и `slice`. Срезы последовательностей»), длину `len`. Можно проверять, есть ли элемент в списке/кортеже, с помощью оператора `in`:

```
>>> 5.5 in L
True
```

Списки являются первыми изменяемыми объектами, с которыми мы познакомимся. Изменять отдельные элементы можно аналогично обращению на запись в C++:

```
>>> iL = id(L)  # удостоверимся, что ссылка не меняется
>>> L[1]
2
>>> L[1] = 'Change successful'
>>> L
[1, 'Change successful', 3.0, 'list', 5.5, (-2-2j)]
>>> id(L) == iL
True
```

(Отметим про себя, что у элемента `L[1]` без каких бы то ни было проблем поменялся тип.) В отличие от массивов C++, у списка могут меняться не только отдельные элементы, но и их количество, так что более корректно проводить аналогию не с массивом, а с вектором указателей непределенного типа `std::vector<void*>`. Перечислим соответствующие методы:

```
>>> L.pop()                  # извлечь элемент
(-2-2j)
>>> L.pop(1)                 # в том числе по индексу
'Change successful'
>>> L       # список после .pop() недосчитается элемента
[1, 3.0, 'list', 5.5]
>>> L.append([3, -3])        # присоединить элемент
>>> L                        # в нашем случае — список
[1, 3.0, 'list', 5.5, [3, -3]]
>>> del L[0]                 # удалить элемент,
>>> L                        # не возвращая значение
[3.0, 'list', 5.5, [3, -3]]
```

метод `extend` добавляет к списку сразу много элементов, `insert` вставляет элемент в указанную позицию, `sort` сортирует список[6], `reverse` обращает порядок следования

[6] Приведем пример сортировки с необязательными аргументами `key` и `reverse`, которые определяют функцию, по значениям которой будет проводиться сортировка, и порядок сортировки, соответственно:

```
>>> L = [1, -2, -4, 3, 5]
>>> L.sort(key=abs, reverse=True)
>>> L
[5, -4, 3, -2, 1]
```

элементов. Обратите внимание, что перечисленные методы возвращают не список, получившийся в результате, а `None`, служебный «пустой» объект Python[7]. Попытка написать что-то типа `L = L.reverse()` приведет к потере ссылки на список:

```
>>> L.reverse()
>>> L
[[3, -3], 5.5, 'list', 3.0]
>>> L = L.reverse()
>>> print(L)    # print нужен, потому что выражения
None            # None не выводятся в консоль сами
```

6. Логические выражения и условный оператор

Итак, мы познакомились с некоторым необходимым минимумом типов данных в Python и можем переходить к логике программы. До сих пор мы вводили команды строго по одной, что, конечно же, является весьма экзотическим случаем, и наконец хотим написать целую программу, которая что-то запрашивает у пользователя и обрабатывает введенные с клавиатуры данные. Перейдем временно из интерактивной подсказки в текстовый редактор и посмотрим на следующий пример:

```
s1 = input('Write something! ')
s2 = input('Write something! ')
s3 = input('Write something! ')
n = len(s1) + len(s2) + len(s3)
if n == 0:
    print('Oh you lazy one :-( ')
elif n < 15:        # в C++ мы бы написали else if
    print('Small words, big sense, I know')
    print('I especially loved this part: '+s2)
elif n < 100:
    print('That\'s a nice conversation! ')
    print('Should go, see you later')
else:
    print('A writer in you asks for freedom')
print('Bye!')
```

Вряд ли этот код нуждается в расширенных комментариях по функционалу, поэтому ограничимся комментариями по синтаксису. Оператор **`if`** `condition:` проверяет условие `condition`. Если оно истинно, выполняются команды, следующий за двоеточием (двоеточие строго обязательно, его отсутствие вызовет синтаксическую ошибку). Это может быть как одна команда, набранная в той же строке, что и `if`, так и блок команд[8], начинающийся со следующий строки и имеющий *отступ* вправо относительно `if`. О том, что блок команд закончился и происходит выход из оператора `if`, Python понимает по отступам в коде и только по ним. Никакой возможности как-либо по-другому обозначить блок команд в Python нет. Рекомендация обозначать структуру кода отступами, справедливая в любом языке программирования, в Python является жестким требованием

[7] Любые функции и методы Python обязаны возвращать значение. В случае, когда это не требуется семантически (т. е. когда в C++ была бы написана функция или метод типа `void`), возвращается `None`.

[8] А вот быть совсем пустым `if` не может, как и другие многострочные операторы. В случае, если `if` обозначает пока не написанную логику, ему можно передать на выполнение команду `pass`, которая не делает ничего, но синтаксически является командой.

и элементом синтаксиса. (Чаще всего рекомендуют отступ в 4 пробела или табуляцию, но Python поймет любой отступ, лишь бы он не менялся в пределах блока.[9]) При работе в интерактивной подсказке конец многострочного оператора обозначается дополнительной пустой строкой. Пока пустая строка не введена, можно продолжать ввод команд, в том числе блоков `elif/else`. В примерах мы будем ее опускать.

Блоки `elif` достигаются только если предыдущие `if/elif` не выполнялись, блок `else` выполняется, если не выполнился ни один `if/elif`. Как `elif`, так и `else` являются необязательными. Заметим также, что в отличие от C++ и многих других языков программирования в Python нет оператора выбора `switch/case`, вместо которого предлагается использовать цепочки `elif`, подобные нашему примеру[10].

Теперь поговорим про `condition`. Во-первых, разрешено проверять истинность любых переменных или выражений. Как `False` будут оценены числа, равные 0, пустые строки, пустые коллекции и `None`, а все остальное вернет `True`. Например, поскольку извлекать элементы можно только из непустого списка, можно перед вызовом метода `list.pop` явно это проверить (предполагаем, что список `L` объявлен заранее):

```
>>> if L: x = L.pop()
```

Во-вторых, условия можно комбинировать с помощью логических операторов `not`, `and`, `or`. Стоит отметить, что, как и в других языках, вычисление логических выражений является ленивым: как только в цепочке `and` находится первый `False` (или в цепочке `or` — первый `True`), результат становится понятен интерпретатору, и остальные элементы цепочки не вычисляются. Приведем пример:

```
>>> L = [1]
>>> if not (L.append(2) and L.pop()):
...     print(L)
[1, 2]
>>> if L.append(3) or L.pop():
...     print(L)
[1, 2]
>>> if L.pop() or L.append(4):
...     print(L)
[1]
```

В первом `if` интерпретатор выполняет функцию `append`. Поскольку она возвращает `None`, результат `and` уже понятен — что бы ни вычислялось позднее, выражение в скобках ложно, поэтому выполнять `L.pop()` нет необходимости. Во втором условном операторе стоит оператор `or`. Его значение после выполнения `L.append(3)` еще не определено, и необходимо выполнить функцию `L.pop()`, убирающую только что добавленное целое `3` и возвращающее его в качестве значения (истинного). Наконец, в последнем `if` сразу после выполнения `L.pop()` с результатом 2 становится понятно, что условие истинное и добавлять к списку 4 уже не надо. Данный пример, конечно же, должен продемонстрировать, что небезопасно вызывать функцию, которая должна поменять какие-либо данные, внутри логического выражения.

---

[9] В том числе нельзя в рамках одного блока попеременно использовать то табуляцию, то пробелы. В конце концов, отступ по табуляции зависит от настроек редактора, и блок, смешивающий табуляцию и пробелы, в другом редакторе перестанет быть выровненным.

[10] В Python 3.10 был добавлен синтаксис сопоставления с образцом (pattern matching), который в простых случаях подобен оператору выбора. Подробнее см. https://peps.python.org/pep-0636/.

Упомянем специфическую для Python особенность `and` и `or`. В предыдущем абзаце мы рассуждали так, будто они возвращают `True` или `False`. Но поскольку можно проверять истинность любого объекта, оператор возвращает «определяющий», последний действительно вычисленный операнд, то есть для цепочки `and` первый ложный или последний, для цепочки `or` первый истинный или последний. Например, `0 or [] or x` вернет `x`, а `[] or 10 or x` вернет `10`.

В-третьих, цепочки сравнений разворачиваются в отдельные сравнения, соединенные `and`. Две следующие записи полностью эквивалентны:

```
>>> 3 < 5 > 1 == 1. < 10
True
>>> 3 < 5 and 5 > 1 and 1 == 1. and 1. < 10
True
```

Это же относится и к оператору принадлежности коллекции `in`, его антониму `not in`, операторам `is` и `is not`, которые имеют равный приоритет с операторами сравнения. Следующий пример абсолютно корректен:

```
>>> 1 < 10 in [10] < [20]
True
```

поскольку преобразуется интерпретатором в выражение

```
>>> 1 < 10 and 10 in [10] and [10] < [20]
```

В-четвертых, встроенные функции `all` и `any` принимают на вход коллекцию и возвращают `True`, если все (для `all`) или хотя бы один из элементов (для `any`) истинны.

```
>>> if [0, 0, 0]: print('True')
True
>>> if any([0, 0, 0]): print('True')
... else: print('False')
False
```

Если вы пользовались тернарным оператором `? :` в C++, вам может быть интересно, что в Python существует похожая конструкция. Код

```
if condition: x = a
else: x = b
```

может быть записан одной командой

```
x = a if condition else b
```

В Python порядок отличается от аналога в C++ и применение ограничено вычислением выражений (то есть нельзя выполнить команду `x = (y = a) if condition else b`; в C++ же `x = condition ? y = a : b;` — корректное выражение). Это связано с желанием разработчиков Python избежать злоупотребления таким синтаксисом.

7. Форматированные строки

В Python существует довольно много способов сформировать строку, содержащую значения каких-либо переменных или объектов. Во-первых, самый банальный способ — воспользоваться конкатенацией строк:

```
>>> x = 8
>>> L = [1, 'a']
>>> 'x / 3 = ' + str(x/3) + '; L = ' + str(L)
"x / 3 = 2.6666666666666665; L = [1, 'a']"
```

Второй способ — уже обсуждавшаяся подстановка через `%`. В отличие от предыдущего способа, этот допускает спецификацию формата вывода, однако все типы, кроме целых и вещественных чисел, все равно должны быть преобразованы в строки явно:

```
>>> 'x / 3 = %10.5g!' % (x/3)
'x / 3 =     2.6667!'
>>> 'L = %s' % str(L)                # [11]
"L = [1, 'a']"
```

Третий способ — использовать строковые шаблоны. Они похожи на `%`-подстановку, но место под переменную зарезервировано с помощью фигурных скобок (пустых или включающих спецификатор формата), а переменные передаются через метод `format` (не будем приводить здесь полную спецификацию, которую можно найти на странице https://docs.python.org/3/library/string.html):

```
>>> 'x / 3 = {:12.5}; L = {}.'.format(x/3, L)
"x / 3 =       2.6667; L = [1, 'a']."
```

Наконец, начиная с версии Python 3.6 существуют так называемые f-строки (см. https://peps.python.org/pep-0498/), которые повторяют строковые шаблоны с той разницей, что переменные, значения которых надо подставить в строковый шаблон, записываются внутри фигурных скобок перед спецификатором формата:

```
>>> f'x / 3 = {x/3:12.5}; L = {L}.'
"x / 3 =       2.6667; L = [1, 'a']."
```

Обратите внимание, что перед такой строкой ставится префикс `f`.

Упомянем еще несколько синтаксических возможностей. В случае, когда строка содержит большое количество символов \ (например, расположение файла), ее удобно записывать с префиксом `r` (сравните вывод на печать и вывод в консоль):

```
>>> s = r'C:\Program Files\Python'
>>> s
'C:\\Program Files\\Python'
>>> print(s)
C:\Program Files\Python
```

А если строка должна содержать заметное количество переносов, то можно начать и закончить ее объявление тройными кавычками, тогда переносы строки исходного кода превратятся в переносы строки `\n`:

```
>>> s = '''A string
... containing
... several lines.'''
>>> s
'A string\ncontaining\nseveral lines.'
>>> print(s)
A string
containing
several lines.
```

[11] Чтобы подставить несколько значений, необходимо собрать их в кортеж — коллекцию, которую мы обсудим в главе 11:

```
>>> 'int %d and float %.3g' % (x, x/3)
'int 8 and float 2.67'
```

8. Циклы

Циклы являются инструментом для многократного исполнения кода. В Python представлены два вида циклов: цикл с предусловием `while` и цикл по итерируемому объекту `for`. Цикл `while` подобен таковому в языке C++, за исключением необязательной секции `else`:

```
m = int(input('Enter an integer: '))
x = 1
k = 0
while m >= x:
    x *= 2
    k += 1
    if k > 50:
        print('I am tired, goodbye')
        break
else:
    print(f'{m} is less than 2 ** {k}')
```

Код в блоке `while` (смещенном вправо относительно самого `while`, как и ранее было с блоком оператора `if`) выполняется, пока истинно условие цикла. Если оно ложно, управление передается блоку `else`. Оператор `break` служит для прерывания цикла (при этом пропускается и блок `else`). Оператор `continue` пропускает остаток тела цикла и возвращается к проверке условия.

Цикл `for` служит для обработки *итерируемых объектов*, то есть таких, которые поддерживают протокол итерации. Протокол итерации, грубо говоря, представляет собой последовательное многократное обращение к объекту, который хранит свое внутреннее состояние и поэтому при каждом обращении выдает «новый» результат, а по исчерпании своих «ресурсов» останавливает итерирование. Это пояснение может быть довольно запутанным, поэтому на текущем этапе мы обратимся к совершенно частному случаю, отложив детали до главы 15 «Итерационный протокол. Итераторы и генераторы». Список может выдавать свои элементы по порядку и остановиться, когда достигнет конца. В ручном режиме это можно делать буквально так (предполагаем, что список `L` объявлен заранее):

```
>>> while L: print(L.pop(0))
```

Этот код вычерпает список `L` до дна, выводя на экран содержимое, и остановится. В общем, список поддерживает протокол итерации и является итерируемым объектом (как и остальные коллекции). Приведем пример:

```
L = [5, 3, [1, 3], 1, 'a', 5, [3, 4]]
y = [1, 3]
for x in L:
    if x == y:
        print('I found it, y is in L')
        break
else:
    print('Sorry, y is not in L')
```

(Разумеется, гораздо проще было бы получить тот же результат, используя проверку `y in L`.) Как и в цикле `while`, разрешены, но совершенно не обязательны `break`, `continue`, `else`.

9. Объекты `range` и `slice`. Срезы последовательностей

Вероятно, обработка разнообразных коллекций — это не то, с чего обычно начинают работу с циклом `for`, и хотелось бы увидеть аналог самого простого и обычного

```
for(int x = i; x < j; x += k)
```

из языка C++. (Если `k` отрицательный, разумеется, условие должно иметь вид `x > j`, чтобы цикл не стал бесконечным.) Буквальный перевод этого кода на Python осуществляется с помощью встроенного объекта `range`:

```
for x in range(i, j, k):
```

Объект `range`, инициированный тремя целыми параметрами, обеспечит именно такое поведение итерационной переменной `x` в цикле: она будет пробегать от `i` включительно до `j` невключительно с шагом `k`. Можно опускать аргументы, при этом `range(j)` эквивалентен `range(0, j, 1)`, а `range(i, j)` эквивалентен `range(i, j, 1)`.

Поскольку значение `j` исключено из итерации, цикл по `range(j)` осуществляет ровно `j` итераций. Такой подход довольно естественен для C++, хотя существует множество языков, использующих другую нотацию. Чтобы поэлементно обратиться к списку из цикла по `range`, необходимо итерировать от 0 до длины списка невключительно (список `L` считаем объявленным заранее):

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

Однако почти всегда лучше использовать более естественное для Python итерирование по объекту

```
for x in L: print(x)
```

Здесь уместно напомнить про итераторы `begin()` и `end()` библиотеки STL, первый из которых указывает на первый элемент контейнера, а второй — на «фиктивный» элемент, следующий за последним. Например, последовательно перебрать все элементы контейнера `std::vector` с помощью этих итераторов можно следующим образом:

```
int w [] = {1, 2, 3, 4, 5};
std::vector<int> v (w, w + 5);
std::vector<int>::iterator i;
for(i = v.begin(); i != v.end(); ++i)
    std::cout << *i << std::endl;
```

Опять же, разность `v.end() - v.begin()` равна 5, числу элементов в контейнере.

Точно такая же логика «от `i` включительно до `j` невключительно с шагом `k`» применяется при взятии срезов строк и списков (а также кортежей `tuple`, которые мы обсудим в главе 11 «Коллекции. Кортежи, множества и словари»). Если оператору `[ ]` передать в качестве аргумента целое число, он вернет один символ из строки или один элемент из списка по указанному индексу (или, для краткости, один элемент последовательности). Чтобы получить сразу много элементов, то есть подпоследовательность, используется тот же самый оператор `[ ]` и специальный тип `slice`, аргументы которого полностью идентичны аргументам `range`. Обычно, впрочем, `slice` не определяется явно, а используется краткая запись `[i:j:k]`.

```
>>> '0123456789ABCDEF'[2:14:3]
'258B'
>>> [0,1,2,3,4,5,6,7,8][slice(3, 6, 1)]
[3, 4, 5]
```

В отличие от `range`, отрицательные числа в `slice` будут интерпретированы особым образом, как индексация с конца (т. е. к ним прибавится длина последовательности):

```
>>> [0,1,2,3,4,5,6,7,8,9,
...     'A','B','C','D','E','F'][-3:2:-5]
['D', 8, 3]
>>> '01234567'[-5:-2:1] == '01234567'[8-5:8-2:1]
True
```

Кроме того, при краткой записи `slice` можно опускать не только `i` и `k`, но и `j`. Шаг `k` по умолчанию все так же равен 1 (если он опущен, можно не писать и второе двоеточие), для положительных и отрицательных `k` при пустом `i` срез начнется с нулевого или последнего элемента, соответственно, при пустом `j` срез закончится концом или началом последовательности, соответственно:

```
>>> '0123456789ABCDEF'[:11:5]
'05A'
>>> [0,1,2,3,4,5,6,7,8,9,'A','B','C','D','E','F'][10::-5]
['A', 5, 0]
>>> [0,1,2,3,4,5,6,7,8][-7:-2]
[2, 3, 4, 5, 6]
```

Заметим также, что выход среза за границу последовательности не вызовет ошибку, в отличие от обращения к одному элементу:

```
>>> '0123456789ABCDEF'[13:100]
'DEF'
>>> '0123456789ABCDEF'[100]
IndexError: string index out of range
```

10. Функции

Как и в любом другом языке программирования, функции Python предназначены для структурирования кода и многократного выполнения отдельных блоков программной логики. Функции создаются с помощью оператора `def`, принимают аргументы, перечисленные в круглых скобках, и возвращают значение, обозначенное зарезервированным словом `return`:

```
def is_odd(x):
    if x % 2:
        print(x, 'is odd')
        return True
    else:
        print(x, 'is even')
        return False
if is_odd(15):
    is_odd(30)
else:
    is_odd(27)
```

Как и в предыдущих случаях, тело функции обязано быть сдвинутым вправо относительно оператора `def`. Подобно C++, выполнение функции прекращается после выполнения команды `return`. Если после `return` не указано значение, либо если функция завершилась, не достигнув команды `return`, возвращается объект `None`. Еще раз обратим внимание на динамическую типизацию: нигде нет проверок того, какой тип передается и возвращается. Так, например, вызов

```
is_odd('string with int %d')
```

не приведет к ошибке, потому что `'string with int %d' % 2` — полностью корректное выражение, оно подставит `'2'` вместо `'%d'`. В первую очередь за типами переменных, аргументов, возвращаемых значений следит программист, а Python отображает ошибки, только если что-то пошло не так, например, если передать список `is_odd([1, 2, 3])`, для которого не определен оператор `%`.

Заметим, что `def` является исполняемым оператором, и неправильно рассматривать его как декларацию. Он создает в оперативной памяти объект функции и назначает ссылку-имя. Если поток управления не достигает оператора `def`, функция не будет существовать. Так, например, в результате выполнения следующего кода

```
if condition:
    def f(x): return x**2
else:
    def g(x): return x**3
```

будет определена или функция `f`, или функция `g`, но не обе одновременно (в C++ аналогичную конструкцию можно реализовать только используя директивы препроцессора `#if #else #endif`). Более того, функции являются такими же полноправными объектами, как числа, строки, списки. Их можно включать в коллекции (`L = [f]`, см. пример в конце главы), передавать в качестве аргументов (`print(f)`), возвращать в качестве значений (`return f`) и так далее.

При объявлении функции можно объявить параметры по умолчанию, которые будут использованы, если они не указаны при вызове функции явно:

```
def h(x=2): return x ** 5
print(h(1))     # Вывод: 1
print(h())      # Вывод: 32
print(h(2))     # Вывод: 32
print(h(3))     # Вывод: 243
```

Существует еще один способ определить функцию, пригодный для небольших функций вроде `f`, `g` и `h` в последних примерах, так называемые лямбда-функции (они же анонимные или неименованные функции). Выражение

```
lambda args: expression
```

создает объект функции, которая принимает аргументы `args` (через запятую, если их несколько) и возвращает выражение `expression`, — почти как оператор `def`. Однако `def` сам по себе не может входить в выражения и связывает объект созданной функции с переменной-именем, а лямбда-функция как раз входит в какое-то другое выражение, которое и подхватывает соответствующую ссылку. Например, если нам необходима коллекция функций, бывает удобно не создавать их с помощью оператора `def`:

```
def f(x): return x ** 4
def g(x): return x ** 5
L = [f, g]              # сохраняем функции в список
del f, g                # удаляются имена, но не функции!
print(L[1](2))          # Вывод: 32
```

а определить коллекцию сразу с помощью лямбда-выражений:

```
L = [lambda x: x ** 4, lambda x: x ** 5]
print(L[1](2))          # Вывод: 32
```

Данный синтаксис уместен только в случае действительно коротких выражений. Если вы открываете скобку, чтобы перенести часть выражения на другую строку, предпочтительна будет обычная функция.

Лямбда-функции обычно применяются, чтобы не объявлять в глобальном пространстве имен функцию, которая будет использована лишь в одном выражении. Достаточно типично использование лямбда-функций в качестве аргументов других функций, таких как списковый метод `sort` или функция `sorted`. Необязательный аргумент этих двух функций `key` определяет функцию, по значениям которой будет отсортирован список. Например, если мы хотим отсортировать список не по значениям, а по тому, насколько значения далеки от числа 2.9, удобно воспользоваться лямбда-функцией:

```
>>> L = [5, 4, 3, 2, 1]
>>> L.sort()
>>> L
[1, 2, 3, 4, 5]
>>> L.sort(key=lambda x: abs(x-2.9))
>>> L
[3, 2, 4, 1, 5]
```

Напомним, что неименованые функции встречались в таком контексте и в C++. Перевод последнего примера на C++ выглядит следующим образом:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <math.h>
using namespace std;

int main()
{
    vector<double> v {5, 4, 3, 2, 1};
    sort(v.begin(), v.end());
    for (auto x : v) {cout << x << ' ';}; cout << endl;
    // Вывод: 1 2 3 4 5
    sort(v.begin(), v.end(),
        // 3-м аргументом sort может быть лямбда-функция
        // двух аргументов, задающая порядок сортировки:
        [](double x, double y)            // агрументы
        {return fabs(x-2.9) < fabs(y-2.9);} //тело
    );
    for (auto x : v) {cout << x << ' ';}; cout << endl;
    // Вывод: 3 2 4 1 5
    return 0;
}
```

11. Коллекции. Кортежи, множества и словари

Списки `list` с точностью до динамической типизации элементов списка воспроизводят шаблонный класс `std::vector` библиотеки STL и являются ближайшим аналогом массива. Рассмотрим остальные коллекции Python.

Кортеж `tuple` — это неизменяемая последовательность произвольных объектов. Инициализируется он перечислением объектов в круглых скобках

```
>>> T = (1, 2, 5.5, 'tuple', 'x')
>>> T
(1, 2, 5.5, 'tuple', 'x')
```

либо с помощью конструктора

```
>>> L = [1, 2, 'list']
>>> T = tuple(L)
>>> T
(1, 2, 'list')
>>> T = tuple('tuple')
>>> T
('t', 'u', 'p', 'l', 'e')
```

Поскольку круглые скобки (в отличие от квадратных и фигурных) используются для группировки и указания аргументов функции, для создания кортежа длины 1 необходимо добавить запятую после единственного элемента:

```
>>> T = (1,)
>>> type(T)
<class 'tuple'>
>>> x = (1)
>>> type(x)
<class 'int'>
```

Вывод кортежей в консоль следует этому же правилу:

```
>>> tuple([2024])
(2024,)
```

Кортеж является, как и список, последовательностью, и для него определены операции конкатенации, поиска, взятия среза и т. д., однако поскольку он неизменяем, мы не можем добавлять или убирать, сортировать элементы: у кортежа просто нет методов `append`, `remove`, `sort` и прочих. Предлагаем самостоятельно посмотреть на «отсутствующие» методы кортежа, выполнив следующий код:

```
for x in dir(list):
    if x not in dir(tuple):
        print(x, end=' ')
```

и убедиться, что все они изменяют список `list`, кроме `copy`.

Неизменяемость кортежа не подразумевает и не гарантирует неизменяемость *объектов*, которые он содержит. Нельзя изменить элемент кортежа как *ссылку*, то есть заменить объект другим. Однако изменить объект можно, если он это позволяет:

```
>>> T = (1, [2, 3], 4)
>>> T
(1, [2, 3], 4)
>>> T[1] = [2, 3, 5]
TypeError: 'tuple' object does not support item
assignment
>>> iT1 = id(T[1])
>>> T[1].append(5)
>>> T
(1, [2, 3, 5], 4)
>>> id(T[1]) == iT1     # id не изменился
True
```

Впрочем, пока что из изменяемых объектов мы познакомились только со списками. А числа и строки в кортеже не могут быть изменены, и выражение `T[0] += 10` вызовет ошибку.

Используются кортежи там, где имеет смысл неизменяемость последовательности. Как правило, функции, значение которых является упорядоченной парой, например, `divmod` или метод `float.as_integer_ratio`, возвращают именно кортеж. Служебные атрибуты пользовательских классов часто делают кортежами, чтобы пользователь не имел возможности изменить их.

Перейдем к более экзотическим коллекциям. Множества `set` и `frozenset` представляют собой неупорядоченные наборы уникальных элементов, аналогично `std::set` из STL. Уникальность элементов означает, что, в отличии от списка, например, `[1, 'a', 1, 'a']`, элементы не могут повторяться. Как Python гарантирует, что два разных элемента множества имеют разное содержание на протяжении всей работы программы? Он просто разрешает помещать в множество только подлинно неизменяемые объекты, которые хеширует и проверяет повторы хеш-значений при расширении коллекции[12]. На базовом уровне это означает, что элементами множества могут быть числа, строки, неизменяемые множества `frozenset`, а также кортежи, содержащие только вышеперечисленные типы данных. В качестве компенсации за такие неудобства мы получаем скорость поиска и прочих операций, которые могут быть произведены над множествами, потому что сравнение хеш-значений проще сравнения самих объектов (соответствующий пример мы поместили в Приложение В).

Слово «неупорядоченные» означает, что порядок, в котором элементы множества будут записаны в строку или консоль, зависит от самих данных, а не от того, в каком порядке мы заполняли множество, а также что обращение к элементу по индексу с помощью оператора `[ ]` отсутствует в принципе. Множества задаются перечислением элементов в фигурных скобках или с помощью конструктора:

```
>>> P = {(1, 3), 7}         # инициализация перечислением
>>> L = [5, 3, (1, 3), 1, 'a', 5, 3]
>>> # (1, 3) — кортеж длины 2, самостоятельный элемент
>>> S = set(L) # построить множество на основе списка L
>>> S
{1, 3, 5, 'a', (1, 3)}  # дубликаты убраны
```

Операции, разрешенные над множествами, включают:

```
>>> 'a' in S        # принадлежность элемента множеству
True
>>> 'a' in P
False
>>> S & P           # пересечение множеств
{(1, 3)}
>>> S | P           # объединение множеств
{1, 3, 5, 7, (1, 3), 'a'}
>>> S ^ P           # XOR-объединение
{1, 3, 5, 7, 'a'}
```

[12] Хеш-функция принимает произвольные данные и возвращает битовую строку фиксированной длины. Предполагается, что разным исходным данным соответствуют разные хеш-значения, и коллизии (то есть одинаковые хеш-значения при различных исходных данных) исчезающе маловероятны.

```
>>> S - P           # разность множеств[13]
{1, 3, 5, 'a'}
>>> P - S
{7}
>>> {1, 3} < S      # строгое подмножество
True                # (есть и нестрогое <=)
>>> S > P           # эквивалентно P < S
False
>>> S == P          # равенство множеств (есть и !=)
False
>>> len(S)          # количество элементов множества
5
```

Подобно списковому методу `append`, в множество `set` можно добавить элемент с помощью метода `add` (если элемент уже был в множестве, оно останется без изменений). Метод `discard` убирает элемент, если он был, `remove` убирает элемент, если он был, а в противном случае вызывает ошибку. Неизменяемые множества `frozenset` соотносятся с изменяемыми так же, как кортежи со списками: у `frozenset` нет методов, которые поменяли бы содержимое коллекции. Благодаря этому они сами являются хешируемыми и могут быть элементами других множеств или ключами словарей.

Как мы уже сказали, множество является достаточно экзотической коллекцией. Применять множества имеет смысл там, где формулировка задачи так или иначе опирается на уникальность элементов или на операции с множествами (пересечение и т. д.) в математическом смысле. Например, именно с помощью множеств проще всего написать проверку того, есть ли в списке повторяющиеся элементы:

```
>>> L = [1, 2, 3, 'a', 'b']
>>> len(L) == len(set(L))       # повторов нет
True
>>> L = [1, 1, 2, 3, 'a', 'b', 'b']
>>> len(L) == len(set(L))       # повторы есть
False
```

является ли одна последовательность перестановкой другой:

```
>>> set('abc') == set('cba')
True
>>> set([1, 10, 'a']) == set([10, 1, 'b'])
False
```

какие буквы алфавита пропущены в панграмме:

```
>>> (set('шифровальщица попросту забыла ряд ключевых
множителей и тэгов') - set('съешь ещё этих мягких
французских булок, да выпей чаю'))
{'ж'}
```

какие методы есть только у изменяемых коллекций:

```
>>> set(dir(list)) - set(dir(tuple))      # опустим вывод
>>> set(dir(set)) - set(dir(frozenset))   # для краткости
```

и тому подобное.

---

[13] Разностью множеств S и P называется множество всех элементов S, не принадлежащих P. XOR-объединением или симметричной разностью множеств S и P называется множество всех элементов, принадлежащих или S, или P, но не обоим множествам одновременно, т. е. `S ^ P == (S-P) | (P-S)`.

Последним типом коллекций в Python являются словари `dict` (ассоциативные массивы, `std::map` в STL). Они сопоставляют произвольные объекты-значения уникальным ключам. Как и элементы множеств, ключи должны быть хешируемы и не могут повторяться. На значения никаких требований не накладывается. Стандартные операции над словарями похожи на операции над списками:

```
>>> D = {'key': 'val', 1: 2, (3, 4): [5, 6., 'a']}
>>> (3, 4) in D              # вхождение ключа
True
>>> D[1]                     # значение по ключу
2
>>> D['key'] = 'value'       # переписать значение
>>> D['a'] = 3               # добавить ключ и значение
>>> D.pop((3, 4))            # извлечь значение по ключу
[5, 6.0, 'a']
>>> D
{'key': 'value', 1: 2, 'a': 3}
```

Иногда бывает удобно работать не с самим словарем, а с коллекцией его ключей `D.keys()`, которая поддерживает операции над множествами (`& | - ^`), коллекцией значений `D.values()`

```
>>> (3 in D, 3 in D.values())
(False, True)
```

и коллекцией кортежей — пар ключ-значение — `D.items()`. Метод `popitem` извлекает из словаря как раз такую пару:

```
>>> D.popitem()
('a', 3)
```

Порядок, в котором будет действовать `popitem`, как и порядок элементов при выводе словаря и коллекций `keys`, `values`, `items`, обычно определяется порядком, в котором словарь был заполнен[14], но полагаться на него не следует. Сравнение словарей `==` и `!=`, например, его игнорирует. Метод `update` обновит словарь ключами и значениями из другого словаря. При совпадении ключей старые значения будут утрачены:

```
>>> D.update({'key': None, 5: 7})
>>> D
{'key': None, 1: 2, 5: 7}
```

Преимущества словарей по сравнению со списками проявляются, когда обращение по целочисленному индексу становится неудобным. Например, в виде словаря естественно представить статистику, сколько раз в списке встречается каждый его элемент:

```
L = [5] * 10 + ['a'] * 3 + [(1, 2)]
D = {}
for x in set(L):
    D[x] = L.count(x)
print(D)        # Вывод: {(1, 2): 1, 5: 10, 'a': 3}
```

Иногда словари используются вместо классов при организации небольших баз данных[15] (например, пусть это будут химические элементы):

---

[14] Начиная с версии Python 3.7 такое поведение гарантировано.

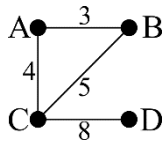
[15] Вообще говоря, поскольку требования к этой базе данных могут возрасти в будущем, разумно сразу воспользоваться фабриками классов `namedtuple` или `@dataclass`.

```
Na = {'name': 'Sodium', 'num': 11, 'mass': 22, 'ion': +1}
O  = {'name': 'Oxygen', 'num':  8, 'mass': 16, 'ion': -2}
```

Здесь словарь будет удобнее, чем, например, кортеж `Na = ('Sodium', 11, 22, +1)`, потому что вместо того, чтобы запоминать, что масса атома — это элемент кортежа по индексу `[2]`, мы будем писать интуитивно понятное `Na['mass']`.

Еще одним примером могут служить разреженные или плохо упорядоченные структуры данных типа графов:

```
Graph = {'AB': 3,
         'AC': 4,
         'BC': 5,
         'CD': 8}
```



Наконец, словарями являются многие служебные объекты Python: коллекция глобальных переменных `globals()`, коллекция атрибутов объекта `object.__dict__` и т. п.

Подчеркнем, что все коллекции являются итерируемыми объектами и могут обрабатываться с помощью цикла `for`. Для множеств это вообще единственный разумный способ получить доступ к элементам:

```
S = {1, 2, 3, 13, 15}
for x in S:
    if x + 10 in S:
        print(x, x + 10)
#Вывод: 3 13
```

Для словарей существует обращение по ключу, однако, поскольку ключи не всегда известны заранее, так же используется цикл (словарь `D` считаем определенным заранее):

```
for key in D:
for key, val in D.items():
```

Смысл второго (более удобного) выражения мы разберем в главе 13 «Распаковка коллекций».

Завершая разбор коллекций, напомним посмотреть Приложение Б и порекомендуем потестировать не описанные здесь методы в качестве творческого отдыха.

12. Неочевидные следствия модели «объект—переменная» в Python

Вернемся к переменным Python, которые, как уже упоминалось, представляют собой ссылки на объекты в оперативной памяти. При этом само по себе присваивание не создает новый объект, а только назначает новую ссылку. Изменения объекта:

```
>>> L = M = [1, 2, 3, 4]
>>> L.extend([5, 6])
>>> L += [7, 8]
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
>>> M
[1, 2, 3, 4, 5, 6, 7, 8]
```

будут видны по каждой из ссылок. (Чтобы предотвратить такое поведение, необходимо вместо присваивания `L = M` использовать копирование `L = M.copy()` или `L = M[:]`.) Здесь, однако, стоит вспомнить, что многие типы являются неизменяемыми, и несмотря на

то, что кортеж очень похож на список, предыдущий пример, будучи реализованным на кортежах, приведет к совершенно другому результату:

```
>>> T = P = (1, 2, 3, 4)
>>> T += (5, 6)
>>> P
(1, 2, 3, 4)
>>> T
(1, 2, 3, 4, 5, 6)
```

Полиморфизм языка в данном случае играет против нас: один и тот же оператор += реализует принципиально разные действия в зависимости от типа. Для списка он вызывает метод `L.__iadd__([7, 8])`, который изменяет список на месте, не создавая нового объекта. У неизменяемого кортежа нет такого метода, и интерпретатор вспоминает про то, что `T += (5, 6)` — это сокращение записи `T = T + (5, 6)`, и осуществляет команду `T = T.__add__((5, 6))`. Результатом `__add__` является новый объект, на который перенаправляется ссылка `T`. А ссылка `P` остается неизменной, как и объект, на который она указывает. Общая картина, таким образом, несколько парадоксальная: поведение изменяемых объектов похоже на работу с указателями в C++, а поведение неизменяемых объектов — на работу с переменными без использования указателей.

Ситуация только усложняется, когда мы начинаем работать с вложенными структурами, например, когда список содержит другой список. Если быть более точным, список содержит ссылки на произвольные объекты, которые сами по себе могут быть изменяемыми. Рассмотрим следующий пример:

```
>>> x = [2, 3]
>>> M = L = [1, x, 4]
>>> L[1].append(5)              # (i)
>>> x.append(6)                 # (ii)
>>> M
[1, [2, 3, 5, 6], 4]
>>> M[1] = [7, 8]               # (iii)
>>> x
[2, 3, 5, 6]
>>> L
[1, [7, 8], 4]
```

В момент выполнения команд (i) и (ii) `L[1]`, `M[1]` и `x` ссылаются на один и тот же объект, соответственно, изменение на месте этого объекта в результате выполнения `append` видно в каждом из трех списков. Напротив, присваивание (iii) — это переназначение ссылки-элемента `M`, не имеющее никакого отношения к ссылке `x`. Однако для списка `M` это все так же изменение на месте, и поэтому оно видно и в `L`.

В случае со вложенными структурами не всегда спасает даже копирование `copy`. Этот метод осуществляет *поверхностное копирование*, то есть скопированный список — это новый объект, но ссылается он все на те же элементы.

```
>>> L = [1, {2, 3}, 4]
>>> M = L.copy()
>>> M is L          # список — другой объект
False
>>> M[1] is L[1]    # а элемент списка тот же
True
```

```
>>> M.append(5)      # изменение M не видно в L
>>> M[1].add(6)      # а изменение элемента — видно
>>> L
[1, {2, 3, 6}, 4]
```

Чтобы полностью исключить возможность обратиться к вложенным структурам одного списка через другой, необходимо использовать *глубокое копирование*, вызывая функцию `deepcopy` из модуля `copy`:

```
>>> import copy
>>> L = [1, {2, 3}, 4]
>>> M = copy.deepcopy(L)
>>> L == M
True
>>> L[1] is M[1]
False
```

Из вышесказанного не следует, что все операторы присваивания в программе надо заменить на вызов `deepcopy` или заменить все списки на кортежи. Это было бы очень неэффективно по времени и памяти, а также громоздко с точки зрения читаемости кода. Что действительно стоит делать — это следить за тем, где вы (потенциально) изменяете объект, и понимать, что полиморфические операции могут быть как «безопасным» присваиванием (само по себе присваивание `name = ...` никогда не изменяет объект, на который `name` указывал ранее), так и изменением на месте.

Новые ссылки на уже существующие объекты возникают не только при присваивании, но и при обращении к элементам коллекции с помощью цикла `for`, и при передаче аргументов в функцию. Начнем с примера с циклом:

```
>>> L = [[3, 1, 2], [6, 5, 4], [7, 9, 8]]
>>> for x in L:
...     x.sort()
>>> L
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

На каждой итерации цикла `x` становится ссылкой на тот же самый объект, на который ссылается элемент списка `L`. Благодаря этому сортировка, которая является изменением `x` на месте, «сохраняется» в `L`, что может быть как желательным, так и нежелательным эффектом. Если мы не хотим изменять `L`, опять же, необходимо копировать `x`:

```
>>> L = [[3, 1, 2], [6, 5, 4], [7, 9, 8]]
>>> M = []
>>> for x in L:
...     y = x.copy()            # или M.append(sorted(x))
...     y.sort()                # функция sorted всегда
...     M.append(y)             # создает новый список
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> L
[[3, 1, 2], [6, 5, 4], [7, 9, 8]]
```

Аналогично, передача аргументов в функцию и возвращение значений осуществляются по ссылке. И для изменяемых объектов поведение точно такое, как при передаче аргумента по ссылке в C++:

```
>>> def f(x, y):
...     x += y
...     return x
>>> L = [1, 2]
>>> M = [3, 4]
>>> print(f(L, M), ';', L)
[1, 2, 3, 4] ; [1, 2, 3, 4]
>>> f(L, M) is L
True
```

Но для неизменяемых типов поведение объектов фактически воспроизводит передачу аргументов по значению в C++:

```
>>> T = (1, 2)
>>> P = (3, 4)
>>> print(f(T, P), ';', T)
(1, 2, 3, 4) ; (1, 2)
```

И здесь мы снова наблюдаем пример того, что полиморфизм Python потенциально приводит к нежелательным результатам, если не отдавать себе подробный отчет, какие операции изменяют объект, а какие создают новый, что зависит как от самой операции, так и от типа данных.

Часть II. За пределами базового синтаксиса

13. Распаковка коллекций

Распаковка коллекций позволяет с помощью одного присваивания получить ссылки на все элементы коллекции. Повторимся, что присваивание в Python — это не действие над объектом, а лишь создание или перенаправление ссылки. Ссылки создаются, однако, не только когда мы пишем явно `x = ...`, но и когда передаем аргументы в функции, возвращаем значения из функций и получаем доступ к элементам коллекций через цикл `for`. На внутреннем уровне все эти случаи реализованы почти одинаково, и распаковка возможна в каждом из них.

Начнем с хрестоматийного случая «присваивания группой»:

```
>>> a, b, c = 10, 20, 30
>>> print(a, b, c)
10 20 30
>>> c, a, b = a, b, c
>>> print(a, b, c)
20 30 10
```

Может быть не вполне очевидно, при чем здесь коллекции. На самом деле справа от оператора присваивания в обоих случаях формируется кортеж из трех элементов[16]. Далее интерпретатор Python пытается сопоставить выражение слева операнду справа, и понимает, что надо взять 0-й, 1-й и 2-й элементы кортежа для перечисленных слева переменных в порядке следования. На случай, если коллекция справа от оператора присваивания содержит больше элементов, чем доступно переменных слева (или на одну меньше), можно с помощью знака `*` указать интерпретатору *ровно одну* из переменных, которая должна собрать «лишние» элементы (или остаться пустой):

```
>>> L = [1, 2, 3, 4, 5]
>>> a, *b, c = L
>>> print('a =', a, '; b =', b, '; c =', c)
a = 1 ; b = [2, 3, 4] ; c = 5
>>> *a, b, c = L
>>> print('a =', a, '; b =', b, '; c =', c)
a = [1, 2, 3] ; b = 4 ; c = 5
>>> a, b, *c = L
>>> print('a =', a, '; b =', b, '; c =', c)
a = 1 ; b = 2 ; c = [3, 4, 5]
>>> a, b, c, *d, e, f = L
>>> print(a, b, c, d, e, f)
1 2 3 [] 4 5
```

Именно такая форма присваивания используется для удобной записи циклов `for` по объектам `zip`, `enumerate`, словарям (**`for`** `k, v` **`in`** `D.items():`) и т. д. Итератор `zip` генерирует кортежи из i-ых элементов переданных коллекций, `enumerate` генерирует

[16] Скобки вокруг записи кортежа необязательны, если только их отсутствие не меняет смысл выражения. Мы могли бы заключить правую часть в круглые или квадратные скобки, чтобы обозначить коллекцию явно: `a, b, c = [10, 20, 30]`. Строго говоря, распаковать можно любой итерируемый объект, например, строку `a, b, c = 'abc'` или множество `a, b, c = {10, 20, 30}`, хотя в случае множества мы не контролируем порядок, в котором присвоятся переменные.

кортежи из индекса и соответствующего элемента коллекции[17], метод словаря `items` генерирует кортежи из ключа и значения:

```
# zip(L, M, ...) -> (L[i], M[i], ...)
# enumerate(L)   -> (i, L[i]))
# D.items()      -> (key, D[key])
```

Код с распаковкой соответствующих кортежей для инициализации цикловых переменных выглядит следующим образом:

```
>>> L = [-1, -3, -5, -7, -9]
>>> M = [11, 23, 35, 47, 59]
>>> for i, x in enumerate(L): print(i - x, end=' ')
1 4 7 10 13
>>> for x, y in zip(L, M): print(x + y, end=' ')
10 20 30 40 50
```

Эквивалент этой записи без распаковки мог бы выглядеть так:

```
>>> for x in zip(L, M): print(x[0] + x[1], end=' ')
10 20 30 40 50
```

Аналогичным образом можно заставить функцию возвращать более одного значения, хотя на самом деле, разумеется, возвращаемое значение одно, кортеж из двух элементов, который распаковывается при присваивании:

```
>>> def sum_dif(x, y):
...     return x + y, x - y
>>> a, b = sum_dif(10, 15)      # a = 25; b = -5
```

Распаковка также позволяет передавать в функцию позиционные аргументы одним списком или кортежем. Если мы создадим кортеж `c = (30, 40)` и попытаемся вызвать `sum_dif(c)`, интерпретатор не обнаружит среди аргументов переменную `y` и выдаст ошибку. Чтобы разрешить распаковку `c`, необходимо отметить переменную звездочкой `*` при передаче в функцию:

```
>>> c = (30, 40)
>>> sum_dif(c)      # x = c, y = ???
TypeError: sum_dif() missing 1 required positional
argument: 'y'
>>> sum_dif(*c)     # x, y = c
(70, -10)
```

Последняя строчка также иллюстрирует, что возвращаемое значение является именно кортежем, а не «двумя возвращаемыми значениями» функции. Упомянем еще и возможность передачи аргументов с помощью словаря:

```
>>> D = {'x': 10, 'y': 15}
>>> sum_dif(**D)
(25, -5)
```

В данном случае происходит «двойная распаковка», чтобы передать значения по ключам, тогда как одиночная `sum_dif(*D)` передаст только коллекцию ключей и будет эквивалентна `sum_dif('x', 'y')`.

Существует и обратный случай, когда функция ожидает неизвестное число аргументов, которые будут собраны в кортеж, обозначенный в списке аргументов звездочкой:

[17] У множеств и словарей нет обращения к i-му элементу, однако когда мы обращаемся к ним в цикле, доступ к элементам мы получаем все равно последовательно, в каком-то порядке, хотя он и не известен нам заранее.

```
>>> def indexer(*p):
...     for i, x in enumerate(p):
...         print(x * (i + 1), end = ' ')
>>> indexer(1, 'a', 10, 'b') # p = 1, 'a', 10, 'b'
1 aa 30 bbbb
```

Чтобы аналогичная функция работала без использования звездочки у аргумента, потребовалось бы передавать ровно одно значение, например, список:

```
>>> def jndexer(p):
...     for i, x in enumerate(p):
...         print(x * (i + 1), end = ' ')
>>> jndexer(1, 'a', 10, 'b')     # p = 1, ??? = 'a'
TypeError: jndexer() takes 1 positional argument but 4
were given
>>> jndexer([1, 'a', 10, 'b'])   # p = [...]
1 aa 30 bbbb
```

Отметим, что наиболее общая запись аргументов функции выглядит так:

```
def my_function(*parg, **kvarg):
```

здесь кортеж `parg` соберет позиционные аргументы, а словарь `kvarg` — аргументы, передаваемые парами ключ/значение. Чтобы не занимать много места в основном тексте, мы поместили пример работы с таким синтаксисом в Приложение Г.

14. Обработка исключений

Рассмотрим следующий небольшой пример:

```
L = [1., 2., 3., 4.]
i = int(input('Write an index: '))
x = float(input('Write a value: '))
L[i] = x
print(L)
```

Казалось бы, что может пойти не так? Мягко говоря, что угодно. Пользователь может ввести строку, которую не удастся преобразовать в `int` или `float`, может указать индекс за пределами границ списка. Любое из этих событий вызовет прекращение работы программы, что абсолютно нормально в процессе отладки, но совершенно неприемлемо в рабочем приложении. Для таких и подобных ситуаций существует обработка исключений. Например, предусмотрим, что строки не удастся преобразовать в числа или не удастся обратиться к массиву по указанному индексу:

```
L = [1., 2., 3., 4.]
try:
    i = int(input('Write an index: '))
    x = float(input('Write a value: '))
    L[i] = x
except IndexError:
    print('Wrong index!')
except ValueError:
    print('Cannot convert input to numbers!')
except:
    print('Unexpected error!')
else:
    print('Assignment successful. L =', L)
```

Интерпретатор попытается выполнить блок `try`. Если это удастся, отработает блок `else` и поток управления пойдет дальше. Если возникнет ошибка, интерпретатор выполнит блок `except`, в котором указана соответствующая ошибка, или блок пустого `except`, если не найдет совпадений. Вообще говоря, пустой `except` является нежелательной конструкцией. Например, если в примере выше закомментировать самую первую строку `L = ...`, то вместо явного сообщения `NameError` мы получим лишь неинформативное сообщение, что была *какая-то* ошибка.

Упомянем, что с помощью команды `raise`, например,

```
raise IndexError
```

можно вызывать исключения вручную[18]. Также можно определять новые классы ошибок на основе встроенного типа `Exception`. За пределами классов использование `raise` достаточно нетипично, но пусть, для примера, мы хотим написать функцию, вычисляющую косинус некоторого угла по его синусу (или *y*-координату точки на единичной окружности по ее *x*-координате):

```
>>> def sin2cos(x):
...     return (1 - x**2)**0.5
>>> sin2cos(0.5)
0.8660254037844386
>>> sin2cos(0.707)
0.7072135462503529
```

Достаточно бесполезно проверять вручную тип входных данных: если мы передадим в функцию, например, строку, ошибку `TypeError` вызовет сам Python. Однако если мы передадим число, большее единицы, функция выполнится без ошибки и вернет комплексное число:

```
>>> sin2cos(1.25)
(4.592425496802574e-17+0.75j)
```

Такая ситуация, скорее всего, означает ошибку в предшествующих вычислениях, поэтому имеет смысл не дать ошибке распространиться дальше по коду и провести проверку на диапазон значений вручную:

```
>>> def sin2cos(x):
...     y = (1 - x**2)**0.5
...     if y.imag:
...         raise ValueError(
...             'sin2cos argument must be from -1 to 1')
...     return y
>>> sin2cos(0.5)
0.8660254037844386
>>> sin2cos(1.25)
ValueError: sin2cos argument must be from -1 to 1
```

15. Итерационный протокол. Итераторы и генераторы

Мы уже отмечали, что циклы `for` в Python и C++ настолько отличаются, что провести параллель между ними возможно лишь в случае, который является частным как для Python, так и для C++:

[18] Тем самым все вышесказанное, кроме секции `else`, абсолютно идентично обработке исключений в C++, где за нее отвечают операторы `try` и `catch`, а вручную исключения вызываются оператором `throw`.

```
for x in range(i, j, k):
for(int x = i; x < j; x += k)
```

Действительно, в C++ цикл `for` (по крайней мере в своем классическом виде) является «синтаксическим сахаром» для цикла `while` с дополнительными командами перед циклом и в конце тела цикла:

```
for(int x = 729, y = 0; x; y += (x>>=1) & 1);
```

в то время как в Python он, в основном, обрабатывает разнобразные коллекции или строки:

```
for x, (k, v) in zip(L, D.items()):
```

При этом достаточно безразлично, что за объект подставляется в цикл `for`, будь то упорядоченные строка и список, неупорядоченное множество или итератор вроде `zip` или `enumerate`. Точно так же, кстати, безразлично, какая коллекция или строка подставляется в конструкторы коллекций `list(...)`, `tuple(...)`, `set(...)`, функции `list.extend`, `set.update`, `str.join`, `all`, `sum`, `max`, `sorted` и т. д.:

```
>>> max({'c', 'b', 'a'})
'c'
>>> max('acb')
'c'
>>> max(['b', 'a', 'c'])
'c'
>>> max({100: 'a', 0: 'b', 10: 'c'}.values())
'c'
```

«Полиморфизм» цикла `for` и перечисленных функций обеспечивается так называемым итерационным протоколом. Рассмотрим его чуть более детально на примере списка, не оговаривая каждый раз, что итерируемый объект может быть другого типа.

Итак, незамысловатый цикл `for`:

```
>>> L = [1, 2, 3, 4]
>>> for x in L:
...     print('For', x, end='. ')
>>> else:
...     print('Else.')
For 1. For 2. For 3. For 4. Else.
```

За кадром данного цикла происходит следующее. Когда объект попадает в цикл `for`, он преобразуется в итератор при помощи функции `iter`:

```
>>> L_iter = iter(L)
```

Полученный объект принадлежит специальному классу `list_iterator` и не поддерживает никакие операции, кроме одной: его можно вызывать с помощью функции `next` и получать следующий по порядку объект из списка `L`:

```
>>> next(L_iter)
1
>>> next(L_iter)
2
>>> next(L_iter)
3
>>> next(L_iter)
4
```

пока список не закончится. После этого вызов `next` будет вызывать специальное исключение `StopIteration`:

```
>>> next(L_iter)
StopIteration
```

Цикл `for` самостоятельно перехватывает это исключение, и таким образом наш пример может быть переписан следующим образом:

```
>>> L_iter = iter(L)
>>> try:
...     while True:
...         x = next(L_iter)
...         print('For', x, end='. ')
... except StopIteration:
...     print('Else.')
For 1. For 2. For 3. For 4. Else.
```

Еще раз подчеркнем: в данных примерах нигде нет обращения к списку по индексу. Подставляя вместо `L` множество, строку, коллекцию значений словаря `values` и т. д., мы будем получать корректно работающие примеры.

Рассмотрим теперь итератор `enumerate`, который, как мы помним из главы 13, позволяет «занумеровать» объекты в коллекции. Что же такое объект `enumerate` сам по себе?

```
>>> L = [10, 20, 30, 40]
>>> en = enumerate(L)
>>> en
<enumerate object at 0x00000291A171F6A0>
```

Подобно тому как итератор списка `L_iter` не является сам по себе списком, объект `enumerate` не является списком кортежей `(i, L[i])`, хотя и сформирует таковой, будучи переданным коструктору списка:

```
>>> list(en)
[(0, 10), (1, 20), (2, 30), (3, 40)]
```

Кроме того, в отличие от списка, преобразование `en` к итератору возвращает его же:

```
>>> iter(en) is en
True
```

Объект `enumerate`, как и любой другой итератор, поддерживает только одно действие, вызов с помощью функции `next`, которая возвращает один кортеж из номера и элемента коллекции:

```
>>> en = enumerate(L)
>>> next(en)
(0, 10)
>>> next(en)
(1, 20)
```

При этом `enumerate` хранит свое состояние (текущий `i` и позицию в коллекции), поэтому если передать `en` конструктору списка на текущем этапе, он не включит первые элементы:

```
>>> list(en)
[(2, 30), (3, 40)]
```

По завершении коллекции `enumerate` вызывает исключение `StopIteration`:

```
>>> next(en)
StopIteration
```

В случае со списками `enumerate` продолжает итерацию, даже если список был изменен или расширен:

```
>>> en = enumerate(L)
>>> next(en)
(0, 10)
>>> next(en)
(1, 20)
>>> L[0] = -10
>>> L[2] = -30
>>> L.append(50)
>>> list(en)
[(2, -30), (3, 40), (4, 50)]
```

Однако среди известных нам итерируемых объектов это исключение: строки и кортежи не могут быть изменены, а множества и словари при расширении коллекции пересобирают хеш-таблицу, и `enumerate` не может найти «следующий» элемент, вызывая `RuntimeError`.

Сформулируем основные свойства итераторов и логику итерационного протокола Python следующим образом:

1. Когда объект обрабатывается циклом `for`, он преобразуется в итератор с помощью функции `iter`. Если объект уже является итератором, преобразование к итератору тривиально.
2. В цикле `for` итератор вызывается с помощью функции `next`, возвращая «элемент», то есть объект, который присваивается итерационной переменной. Итератор является «ленивым», то есть он формирует очередной элемент только по запросу `next`. Если цикл прервался, не достигая последних элементов итератора, они не формируются и никуда не передаются.
3. Когда итератор достигает конца контейнера (истощается), по запросу `next` он генерирует исключение `StopIteration`, которое автоматически перехватывается и завершает цикл `for`. Итератор является «одноразовым»: как только он истощился, обращение к нему всегда будет вызывать исключение.

Помимо итераторов `enumerate`, `zip` и `reversed` (который пробегает по последовательности в обратном порядке), в Python существует ряд реже используемых итераторов, которые мы рассмотрим в главе 24 «Модуль `itertools`». Пользовательские итераторы создаются с помощью генераторных функций.

Генераторные функции определяются с помощью оператора `def`, как и обычные функции, однако возвращают значения с помощью оператора `yield` вместо `return`, причем предполагается, что `yield` достигается неоднократно. Напишем простой генератор, который позволит пошагово отследить поток управления:

```
def gen(L):
    print('A')
    x, y, *_ = L
    yield x
    print('B')
    yield y
    print('C')
    yield 'Finish'
    print('D')
```

Генератор инициализируется путем вызова генераторной функции:

```
>>> G = gen([0, 1, 2])
```

Как можно видеть по отсутствию вывода в консоль, код генераторной функции не исполняется. Он будет выполняться только по вызову

```
>>> next(G)
A                   # вывод print('A')
0                   # вывод результата next(G)
```

Интерпретатор дошел до строки `yield x`, вернул результат (0), и на этом исполнение кода функции останавливается до следующего вызова

```
>>> next(G)
B                   # следующая после yield x строка
1
>>> next(G)
C                   # следующая после yield y строка
'Finish'
>>> next(G)         # после yield 'Finish' код
D                   # выполняется до конца функции
StopIteration       # и вызывает исключение
```

Тем самым, код генераторной функции выполняется «блоками» от `yield` до `yield`.

Удобно и эффективно использовать `yield` внутри цикла. Например, итератор `enumerate` может быть написан вручную следующим образом:

```
def enumer(iterable):
    i = 0
    for x in iterable:
        yield (i, x)
        i += 1
```

Вызов `enumer(L)` практически эквивалентен `enumerate(L)`, в каком бы контексте мы его ни использовали:

```
>>> L = 'abcd'
>>> list(enumer(L))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
>>> for i, x in enumer(L): print(i, x)    # Вывод опустим
```

Для каких целей имеет смысл применять генераторные функции? Во-первых, они могут быть удобны в смысле архитектуры программы как способ спрятать цикл (возможно, достаточно нетривиально организованный) внутрь объекта, который в дальнейшем можно передать другим функциям (обработчикам, тестировщикам и т. д.). Во-вторых, ленивость генераторов может быть полезна с точки зрения экономии машинных ресурсов. Постараемся продемонстрировать оба этих свойства на следующем примере.

Пусть мы хотим пройти по списку

```
>>> M = [1, 3, 5, 2, 4, 7, 6, 9, 8, 11]
```

и отобрать элементы, начиная с первого, таким образом, чтобы каждый следующий был строго больше предыдущего, а получившуюся в результате подпоследовательность каким-либо образом обработать (например, получить максимальную и минимальную разницу соседних элементов в ней). Сформировать такую подпоследовательность в виде отдельного списка можно циклом:

```
>>> G = []
>>> last = M[0] - 100   # для корректного 1-го шага цикла
>>> for x in M:
...     if x > last:
...         G.append(x)
...         last = x
>>> G
[1, 3, 5, 7, 9, 11]
```

Но если список `M` достаточно большой, вместо того чтобы создавать список `G` (его длину мы никак не можем узнать заранее, но потенциально он может быть просто копией `M`), мы хотели бы сформировать итератор и передать его обработчику этих данных. Напишем генераторную функцию:

```
def grower(L):
    last = L[0] - 100
    for x in L:
        if x > last:
            yield x
            last = x
```

Как видно, цикл внутри функции `grower` точно такой же, как мы использовали для формирования списка `G`, но вместо того чтобы добавить очередное значение в список, мы возвращаем его с помощью `yield`. И хотя при передаче генератора в конструктор списка

```
>>> list(grower(M))
[1, 3, 5, 7, 9, 11]
```

мы получаем точно такой же список `G`, он не будет сформирован и не займет память, пока мы не потребуем. Написав теперь обработчик, который получает из итерируемого объекта разности соседних элементов:

```
def diff(L):
    L = iter(L)
    one = next(L)
    for two in L:
        yield two - one
        one = two
```

мы можем непосредственно запросить поиск максимума и минимума с помощью встроенных функций:

```
>>> max(diff(grower(M)))
2
>>> min(diff(grower(M)))
2
```

Мы, таким образом, провели существенную декомпозицию задачи, отделив подготовку данных от их обработки, вместо того чтобы пытаться уместить все в один большой цикл по `M`. Если теперь задача поменяется (например, следующий элемент должен быть больше предыдущего как минимум на 3), мы модифицируем `grower`:

```
if x >= last + 3:
```

не трогая остальное. И наоборот, если нам потребуются другие характеристики, мы будем менять только обработчики:

```
>>> sum(diff(grower(M)))
10
```

```
>>> sum(grower(M))
36
```
но не генератор `grower`. Можно представить, насколько сложнее было бы перейти от поиска максимума к сумме, если бы мы осуществляли их в цикле по `M` аналогами функций `sum` и `max`, написанными самостоятельно.

Приведенные выше примеры носят, разумеется, упрощенный характер. Немного более масштабный пример мы поместили в Приложение Д.

16. Генераторные выражения. Списковые включения

Помимо общего синтаксиса генераторных функций, Python предоставляет возможность задавать генераторные выражения (итерируемый объект `L` считаем объявленным заранее):

```
G = (val for x in L if condition)
```

где выражения `val` и `condition`, разумеется, могут зависеть как от `x`, так и от других переменных. Приведенное генераторное выражение эквивалентно генератору, созданному при помощи следующей генераторной функции:

```
def gen(iterable):
    for x in iterable:
        if condition:
            yield val
G = gen(L)
```

Если условный оператор не нужен, его можно опустить: `(val for x in L)`. Генераторные выражения перенимают все свойства генераторов: они поддерживают итерационный протокол, вычисляются «лениво» и являются одноразовыми.

Например, если мы хотим посчитать сумму кубов чисел в списке `L`

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

мы можем передать функции `sum` генераторное выражение:

```
>>> sum((x**3 for x in L))
100
```

А если нас интересует сумма кубов нечетных чисел списка `L`, дополнить генераторное выражение условием:

```
>>> sum(x**3 for x in L if x%2)
28
```

Здесь мы опустили скобки вокруг генераторного выражения, поскольку оно является единственным аргументом функции. В более сложных случаях интерпретатор может потребовать явно обозначить генераторное выражение скобками.

Подобно любому итератору, генераторное выражение можно передать конструктору списка или множества, однако и для этого случая существует краткая запись, так называемые списковые включения (list comprehension):

```
>>> [x**3 for x in L]       # список
[1, 8, 27, 64]
>>> {x % 3 for x in L}      # множество — дубликат убран
{0, 1, 2}
```

Чтобы создать словарь, необходимо в фигурных скобках разделить двоеточием ключ и значение:

```
>>> {x: x**3 for x in L}    # словарь
{1: 1, 2: 8, 3: 27, 4: 64}
```

По сравнению с поэлементным заполнением списков (множеств, словарей) такие выражения вычисляются несколько быстрее (см. Приложение Е), а также выглядят более лаконично.

Генераторные выражения и списковые включения поддерживают и итерацию по нескольким объектам:

```
>>> L = [1, 3, 5]
>>> M = [11, 23, 35]
>>> [x+y for x in L for y in M]
[12, 24, 36, 14, 26, 38, 16, 28, 40]
```

При множественной итерации циклы являются вложенными (и последний по порядку является внутренним), но результат является «плоским» списком. Чтобы получить вложенные списки, необходимо явно обозначить списковое включение у первого цикла:

```
>>> [[x+y for x in L] for y in M]
[[12, 14, 16], [24, 26, 28], [36, 38, 40]]
```

При этом вложенность циклов перевернется: последний окажется внешним (ср. порядок чисел с предыдущим примером).

17. Области видимости

Подобно другим языкам программирования, в Python функции представляют собой «изолированные» блоки программы. Зачастую они должны работать только с данными, которые им переданы, и всего лишь возвращать какое-то значение. Таковы, например, все математические функции: когда мы вызываем `y = abs(x)`, функция `abs` не использует никаких данных, кроме `x`, а в результате выполнения команды не изменяются никакие данные, кроме `y`.

С точки зрения синтаксиса изолированность функции достигается за счет предоставления ей собственной, локальной области видимости (а с точки зрения машинного кода — еще и собственной области памяти в стеке). В рамках локальной области видимости функция может создавать, изменять, удалять любые объекты, как и основная программа. При этом переменные, переданные в качестве аргументов, также становятся локальными. В частности, мы уже отмечали, что *изменения на месте* аргументов функции остаются в силе и после ее завершения. А как происходит взаимодействие с объектами вне локальной области видимости? Рассмотрим следующий простой пример:

```
def f(x):
    def g(u):
        print(u + x + y + z)
    y = 9
    g(6)
z = 8
f(7)
```

В результате исполнения этого кода на экран будет выведено число 30. В функции `g` интерпретатор попытается вычислить значение `u + x + y + z`. В локальной области видимости он обнаружит только переменную `u`. Не найдя `x`, `y`, `z`, он поднимется на уровень выше, в область видимости объемлющей функции `f`, и найдет там `x` и `y`. Наконец, чтобы обнаружить, что имелось в виду под `z`, он поднимется еще выше, на уровень модуля или глобальный уровень видимости. Если бы `z` не обнаружилось и там, последовала бы попытка искать его во встроенной области видимости `builtins`, где живут имена стандартных объектов (типов, исключений, функций, в том числе вызываемой в `g(u)` функции `print`).

Итак, функция g, помимо своей области видимости, имеет доступ в область видимости объемлющей функции и глобальную (т.е. модуля или скриптового файла). Может ли она менять объекты во внешних областях? Давайте попробуем:

```
def f():
    def g():
        L = [13, 14, 15]
        M[0] = 10
        M[1] = 11
        M[2] = 12
        print('L inside: ', L)
        print('M inside: ', M)
    M = [0, 1, 2]
    g()
    print('M outside:', M)
L = [3, 4, 5]
f()
print('L outside:', L)
```

Ответ на этот вопрос снова нас отсылает к различию присваивания и изменения на месте:

```
L inside:  [13, 14, 15]
M inside:  [10, 11, 12]
M outside: [10, 11, 12]
L outside: [3, 4, 5]
```

Присваивание `L = ...` создает новую переменную, никак не связанную с переменной L на верхнем уровне. Поэтому внутри функции мы вообще не обращаемся к глобальной переменной L и соответствующему списку. А вот обращение по индексу (которое является вызовом `M.__setitem__(0, 10)` и т. д., а совсем не присваиванием) и прочие изменения на месте вызываются для объектов с любого уровня видимости, поэтому изменен оказывается именно список M из области видимости объемлющей функции.

Чтобы разрешить любые манипуляции с переменными объемлющей функций и модуля, надо явно указать интерпретатору, что они взяты из чужой области видимости, при помощи зарезервированных слов nonlocal или global:

```
# глобальная область видимости всех функций программы
def f():
    # локальная область видимости функции f
    # она же нелокальная область видимости функции g
    def g():
        # локальная область видимости функции g
        global L
        nonlocal M
        L = [13, 14, 15]
        M = [10, 11, 12]
    M = [0, 1, 2]
    g()
    print('M outside:', M)
L = [3, 4, 5]
f()
print('L outside:', L)
```

```
# вывод при выполнении данного кода:
M outside: [10, 11, 12]
L outside: [13, 14, 15]
```

Эти же зарезервированные слова нужно использовать, если мы хотим создать переменную вне локальной области видимости, чтобы сохранить ее и после того как функция завершится:

```
def createS():
    global S
    S = 'I exist!'
createS()
print(S)
```

Без объявления `global` попытка выполнить `print(S)` вызовет ошибку `NameError`.

В справочном порядке упомянем стандартные функции `locals` и `globals`, которые возвращают словари со всеми локальными и глобальными переменными, соответственно (на верхнем уровне программы это один и тот же словарь).

На будущее заметим, что область видимости не включает в себя имена, объявленные в объектах и модулях. Если вы написали класс или модуль `name`, содержащий объект `obj`, ваш код обязан обращаться к этому объекту только через соответствующий класс или модуль: `name.obj`. Интерпретатор последовательно ищет имена переменных в локальной, объемлющей, глобальной, встроенной областях видимости, но не среди атрибутов какого-либо имени (т. е. в списке `dir()` какого-либо объекта).

В заключение настоящей главы скажем, что функции без побочных эффектов («чистые»), то есть такие, выполнение которых не вызывает изменения глобальных переменных, файлов и т. д., почти всегда являются предпочтительным архитектурным решением. Три следующих примера идентичны по функционалу:

```
def arr_abs1(L):
    for i in range(len(L)):
        L[i] = abs(L[i])
L = [-2, -1, 0, 1, 2]
arr_abs1(L)
print(L)

def arr_abs2():
    global L
    L = [abs(x) for x in L]
L = [-2, -1, 0, 1, 2]
arr_abs2()
print(L)

def arr_abs3(L):
    return [abs(x) for x in L]
L = [-2, -1, 0, 1, 2]                    # (*)
L = arr_abs3(L)
print(L)
```

но только последний явно указывает, что `L` в момент вызова функции `print` уже не такой, каким он был создан в строке `(*)`, что может быть неочевидно, если функция объявлена десятками строк ранее или в другом модуле.

18. Классы

Во многих современных языках программирования парадигма объектно-ориентированного программирования реализуется посредством возможности создавать классы — пользовательские типы данных, которые объединяют поля (данные) и методы (функции для обработки данных). Вместе поля и методы называются атрибутами класса. Объектно-ориентированный подход — основа языка Python, и классы являются его естественной частью. В силу того, что Python является интерпретируемым динамически типизированным языком, класс и экземпляр класса фактически представляют собой пространства имен (`namespace` в C++) или расширенную версию словаря. Так, ничто не мешает создать пустой класс и наполнить его произвольными данными:

```
>>> class Absurd: pass
>>> a = Absurd()                # создаем экземпляры
>>> b = Absurd()                # a, b = dict(), dict()
>>> a.X = 1                     # a['X'] = 1
>>> a.Y = [1, 2, 3, 4, 5]       # a['Y'] = [...]
>>> b.L = [5, 4, 3, 2, 1]       # b['L'] = [...]
>>> b.M = 'string'              # b['M'] = 'string'
>>> type(a),type(b)
(<class '__main__.Absurd'>, <class '__main__.Absurd'>)
>>> a.Y + b.L
[1, 2, 3, 4, 5, 5, 4, 3, 2, 1]
>>> a.L
AttributeError: 'Absurd' object has no attribute 'L'
```

Абсурдность данного примера заключается в том, что, хотя `a` и `b` формально являются экземплярами одного класса, ни о какой однородности данных в них речи не идет (узнав, какие данные содержит экземпляр `a`, мы не получим информации о структуре экземпляра `b`; функция, которая сможет обработать экземпляр `a`, вызовет ошибки при работе с экземпляром `b`). В данном случае мы с легкостью заменим эти объекты словарями или другими коллекциями.

Разумеется, на самом деле классы полезны для хранения и обработки более однородных данных. Обратимся к чему-нибудь более реалистичному и напишем класс `Vector`, который описывает двумерные векторы и содержит методы для работы с ними. Сначала вспомним, как это реализуется в C++[19]:

```
class Vector {
    public:
        double x, y;                // поля
        Vector() {
            this->x = 0.;           // конструктор
            this->y = 0.;           // без параметров
        }
```

[19] Указатель `this` является адресом экземпляра, а оператор `->` соответствует разыменованию с одновременным доступом к атрибуту. В приведенном фрагменте все конструкции `this->` можно убрать без ущерба для кода, а кроме того, обычно поля описывают в секции `private`. Мы же написали данный фрагмент именно так для дальнейшей аналогии с кодом на Python.

```
        Vector(double X, double Y) {
            this->x = X;            // конструктор
            this->y = Y;            // с параметрами
        }
        double length() {           // метод
            return sqrt(this->x*this->x
                        + this->y*this->y);
        }
};
```

Здесь пока не перегружены операторы сложения и умножения — к ним вернемся позднее.

А теперь напишем аналогичный код на Python:

```
class Vector:
    def __init__(self, X=0, Y=0):    # конструктор
        self.x = X
        self.y = Y
    def length(self):                # метод
        return (self.x**2 + self.y**2)**0.5
```

Сопоставим эти два фрагмента кода. Во-первых, в Python можно определить только один конструктор класса, который для любого класса называется `__init__` и заведомо не является полиморфным. Чтобы предоставить возможность вызывать его без параметров, мы задали значения аргументов по умолчанию (в C++ это также было бы возможно).

Во-вторых, все методы класса являются функциями с как минимум одним аргументом, который по соглашению называется `self` и служит для передачи конкретного экземпляра класса. Внутри методов класса «`self.`» выполняет ту же роль, что и «`this->`» в C++. В Python вообще невозможно обратиться к полям экземпляра без использования `self`. Если мы напишем в `__init__` команду `x, y = X, Y`, мы создадим локальные переменные функции-метода, которые будут уничтожены по его завершении, а экземпляр так и останется пустым.

Итак, конструктор `__init__` получает на вход пустой экземпляр `self` и пару инициирующих значений `X`, `Y`, и заполняет (по сути, создает) соответствующие поля в `self`. Посмотрим теперь, как это работает:

```
>>> p = Vector(3., 4.)  # вызывает __init__
>>> (p.x, p.y)
(3.0, 4.0)
>>> p.length()
5.0
>>> Vector.length(p)
5.0
```

Метод `length` класса, таким образом, может быть вызван как атрибут экземпляра `p`, тогда аргумент передается автоматически, и как вызов метода класса с явной передачей экземпляра.

Как и у встроенных типов, методы, обозначенные двойными подчеркиваниями в начале и конце имени, отвечают за перегрузку стандартных операций или операторов. Добавим в класс приведение в строковую форму и сложение векторов:

```
    def __repr__(self):
        return (f'Vector ({self.x}, {self.y})')
```

```
    def __add__(self, other):
        return Vector(self.x+other.x, self.y+other.y)
```

и убедимся, что они работают:

```
>>> q = Vector(7., 6.)
>>> p + q
Vector (10.0, 10.0)
```

Приведенный код перегруженного оператора сложения аналогичен таковому на C++:

```
Vector Vector::operator+(Vector other)
    {return Vector(this->x+other.x, this->y+other.y);}
```

Однако, если мы вернемся к определению класса, то заметим, что в C++ тип `x` и `y` явно указан как `double`. В Python же ничто не гарантирует, что экземпляр не будет инициализирован другими типами.

```
>>> s = Vector('al', 'be')
>>> t = Vector('pha', 'ta')
>>> s + t
Vector (alpha, beta)
```

Хотя, разумеется, попытка вызвать `s.length` вызовет ошибку, потому что для строк не определен оператор возведения в степень. Точно так же попытка сложить `s` и `p` вызовет ошибку `TypeError`, потому что складывать строки с числами нельзя. В отличие от C++, который проверит соответствие типов на этапе компиляции, интерпретатор Python обнаружит, что операция некорректна, лишь при исполнении кода. Если не вызывать метод `length`, то мы можем сколько угодно работать с векторами, содержащими строки или любые другие типы, для которых определено сложение.

Рассмотрим теперь случай, когда тип аргументов принципиально важен. Оператор умножения естественно перегрузить для двух случаев: умножения вектора на число и скалярного умножения двух векторов. C++ позволяет различить тип операнда на стадии компиляции:

```
double Vector::operator * (Vector other)
    {return this->x*other.x + this->y*other.y;}
Vector Vector::operator * (double other)
    {return Vector(this->x*other, this->y*other);}
```

В Python этой возможности нет, и проверки такого рода приходится выполнять явно[20]:

```
    def __mul__(self, other):
        if isinstance(other, Vector):
            return self.x * other.x + self.y * other.y
        elif isinstance(other, (int, float)):
            # и другие числовые типы, если требуется
            return Vector(self.x * other, self.y * other)
        else:
            raise TypeError
```

Разумеется, помимо функционирования с векторами, на которое мы надеемся:

[20] В качестве альтернативы можно проверять не тип, а наличие необходимых для операции атрибутов:

```
    def __mul__(self, other):
        try:
            return self.x * other.x + self.y * other.y
        except AttributeError:
            return Vector(self.x * other, self.y * other)
```

```
>>> p * q
45.0
>>> p * 5
Vector (15.0, 20.0)
```
при подстановке данных другого типа проявится незапланированное поведение:
```
>>> Vector(2, ['a', 5]) * Vector([8, 'f'], 3)
[8, 'f', 8, 'f', 'a', 5, 'a', 5, 'a', 5]
```

Мы могли бы отсечь все подобные возможности явными проверками типов либо явным приведением аргументов к `float` повсеместно в методах класса. Однако такая архитектура кода в целом считается не соответствующей парадигме Python. Она не позволила бы, например, в рамках одного кода пользоваться любым числовым типом (целые, рациональные, десятичные с бесконечной точностью; вещественные с увеличенной точностью; комплексные, хотя для математической корректности пришлось бы прописать комплексное сопряжение в `__mul__` и `length`). Ответственность за корректное поведение класса `Vector` возложена не только на сам класс, но и на вызывающий его код.

Основные встроенные методы, которые может иметь смысл перегрузить, перечислены в таблице «Основные перегружаемые операторы и методы» (Приложение А).

Мы оставили за рамками повествования такие понятия объектно-ориентированного программирования как наследование и инкапсуляция (в смысле «скрытые данные — открытые интерфейсы»; в C++ за такое разделение отвечают модификаторы `private` и `public`). Инкапсуляция в Python реализована *только* для встроенных типов. О приемах, позволяющих в некоторой степени защитить внутренние переменные объектов от воздействия извне класса, можно прочитать в разделе «Эмуляция защиты атрибутов экземпляра» главы 30 учебника [4] и в пособии [6]. Наследование широко используется как в стандартной библиотеке Python, так и в сторонних модулях. О реализации наследования в Python также можно прочитать в главе 29 учебника [4] и в пособии [6].

19. Работа с файлами

При работе с файлами в Python необходимо создать файловую переменную функцией `open` с ключами, определяющими режим работы с файлом (чтение `'r'` по умолчанию, очистка файла и запись в него `'w'`, добавление в конец файла `'a'`, бинарный режим `'br'/'bw'/'ba'`). После этого чтение и запись происходят с помощью команд `read` (читает все содержимое в одну строку), `readline` (читает одну строку), `readlines` (читает все содержимое файла и возвращает список строк), `write` (записывает переданную строку), `writelines` (записывает переданный список строк). По завершении работы с файлом его необходимо закрыть, вызвав метод `close`.
```
f = open('my_file.txt', 'r')
S = f.read()
f.close()
f = open('my_copy.txt', 'w')
f.write(S)
f.close()
```
Символы конца строки `'\n'` не проставляются автоматически при записи и не убираются из результата при чтении. Следить за ними необходимо программисту.

Файловые переменные поддерживают итерационный протокол, поэтому читать информацию из файла можно и используя цикл `for`:

```
for line in open('my_file.txt', 'r'):
    print(line)
```

или даже списковые включения:

```
L = [float(line) for line in open('my_file.txt', 'r')]
```

В последней строчке мы предполагаем, что в файле записаны вещественные числа, по одному в строке. Разумеется, в файле могут быть строки, которые невозможно преобразовать в числа, и в этом случае программа прервется, но файл может остаться не закрытым (и, например, операционная система Windows может считать, что Python все еще использует `my_file.txt`, и не позволять пользователю другие действия с ним). Как удостовериться, что файл корректно закроется? Для этого существуют две примерно равноценные конструкции. Во-первых, можно использовать `try` с секцией `finally`, содержимое которой выполнится после блока `try` независимо от того, возникнет ошибка или нет:

```
f = open('my_file.txt')
try:
    L = [float(line) for line in f]
except ValueError:
    print('Cannot convert to float')
finally:
    f.close()
print('f is closed:', f.closed) # True при любом исходе
```

Во-вторых, существует протокол `with`, который обеспечен наличием у файловой переменной методов `__enter__` и `__exit__`:

```
with open('my_file.txt') as f:
    L = [float(line) for line in f]
```

При входе в блок `with` выполняется метод `__enter__` у объекта, который вернула функция `open`, и результат записывается в переменную `f` (в случае файловой переменной ее метод `__enter__` возвращает ссылку на саму файловую переменную). По завершении блока `with`, независимо от того, завершился он из-за ошибки или корректно, запускается метод файловой переменной `__exit__`, в котором прописано закрытие файла. И только после этого управление передается обратно основной программе, которая продолжает работу или прерывается из-за ошибки[21].

[21] Блок `with ... as` (менеджер контекста) используется не только для работы с файлами, но и в других случаях, когда работа объекта обязана быть корректно завершена. Одним из примеров является создание многопоточных программ (модули `threading`, `multiprocessing`). В этом случае в методе `__enter__` прописано создание потоков/процессов, в методе `__exit__` — их уничтожение, и соответствующие команды не требуется прописывать явно.

Часть III. Модули

20. Общие сведения о модулях

До настоящего момента мы пользовались только встроенным функционалом Python. Во многих случаях этого функционала недостаточно, и нужно пользоваться расширениями Python, или модулями. В простейшем случае модуль представляет собой просто скриптовый файл с кодом Python, содержащий необходимые объявления переменных, функций, классов. При загрузке модуля с помощью команды `import` все объекты, созданные внутри модуля, оказываются доступны текущей программе. Строго говоря, никаких ограничений на код модуля нет — он может проводить вычисления, читать или записывать файлы, содержать тестовые выводы и т. д., но обычно соответствующие операции (кроме тестов) стараются помещать в функции или классы, а не на верхнем уровне файла. Создадим отдельный файл `sequences.py`, который будет содержать всего несколько определений и вызов `print`:

```
fibbo_default = (1, 1, 1000)
def fibbo(first=1, second=1, maximum=1000):
    yield first
    while second < maximum:
        yield second
        first, second = second, first + second
collatz_default = 3
def collatz(first=3):
    while first != 1:
        yield first
        first = 3*first+1 if first%2 else first//2
    else:
        yield first
__doc__ = 'Sample module for tutorial "Python after C++"'
print('Module sequences.py is ready')
```

Теперь откроем новый файл или интерактивную подсказку, импортируем наш модуль и воспользуемся объявленными в нем объектами:

```
import sequences
f = sequences.fibbo()
print(list(f))
print(sequences.collatz_default)
c = sequences.collatz(5)
print(list(c))
```

Таким образом, всё, что мы определили в модуле, теперь доступно и в основной программе. Кроме того, в консоли мы увидим и строку «`Module sequences.py is ready`». В настоящем, не учебном модуле такой вывод не имеет смысла — отсутствие ошибок при импорте и означает, что модуль загрузился корректно. И тем более мы бы не хотели, чтобы модуль *во время загрузки* проводил долгие вычисления, ожидал ввода через `input` и т. д. Такие операции обычно заключают в функции и вызывают явно.

Разумеется, кому-то может показаться не очень удобным обращение вида `module_with_long_accurate_name.object_with_long_solid_name`. На этот случай, во-первых, всегда можно привязать подобный объект к короткой переменной простым присваиванием:

```
x = sequences.fibbo
f = x()
```

Во-вторых, если нам нужны только несколько объектов, можно импортировать именно их:

```
from sequences import fibbo, fibbo_default
f = fibbo()
print(list(f))
```

или даже совместить с предыдущей возможностью:

```
from sequences import fibbo as x, fibbo_default as y
```

При этом модуль загрузится «целиком» (в частности, будет и вывод в консоль в нашем примере), но в нашу область видимости попадут только те объекты и под теми именами, что мы запросили. В-третьих, для краткости можно переобозначить сам модуль:

```
import sequences as seq
f = seq.fibbo()
c = seq.collatz()
```

или вообще сделать доступными все его объекты в глобальной области видимости:

```
from sequences import *
f = fibbo()
c = collatz()
```

Последний вариант выглядит, пожалуй, наиболее привлекательным, но надо помнить, что импортированные таким образом имена могут экранировать имена, определенные ранее, стандартные имена и имена из других модулей, импортированных со звездочкой:

```
>>> from numpy import *
>>> x = arange(3)
>>> exp(x)
array([1.        , 2.71828183, 7.3890561 ])
>>> from math import *
>>> exp(x)
TypeError: only size-1 arrays can be converted to Python
scalars
```

Поскольку импорт со звездочкой не оставляет имени самого модуля, после двойного импорта со звездочкой нет никакой возможности вызвать `exp` из модуля `numpy`, чтобы потенцировать массив `x`.

Заметим также, что повторный `import` модуля не приведет к ошибке. Стандартные модули часто импортируют для своих нужд другие стандартные модули:

```
>>> import math
>>> import numpy
>>> import sys
>>> numpy.math is math
True
>>> numpy.sys is sys
True
```

которые могли быть запрошены вашей программой раньше или позже, и ни к каким конфликтам это не приводит. При этом модуль не загружается заново, просто создаются новые ссылки на уже существующие объекты. Так, мы можем импортировать `sequences.py` сколько угодно раз, но строку «`Module sequences.py is ready`» увидим только при самом первом импорте.

Разумеется, если вы пишете модуль для широкой публики, желательно настроить документацию, чтобы `help(module)` выводил что-то более полезное, чем в нашем примере, но мы не будем подробно останавливаться на этой теме.

В следующих главах мы опишем модули, которые, на наш взгляд, будут наиболее полезны читателю. Часть из них устанавливается вместе с Python. Другие же написаны сторонними разработчиками и не входят в дистрибутив Python (хотя входят, например, в дистрибутив Anaconda). Для установки стороннего модуля `module` необходимо:

- под Windows выполнить в терминале любую из команд
  ```
  python -m pip install module
  pip install module
  ```
- на системах Ubuntu/Debian выполнить в терминале команду
  ```
  sudo apt install python3-module
  ```
- на MacOS выполнить в терминале любую из команд
  ```
  python -m pip install module
  pip install module
  ```

Те сторонние модули, о которых будем говорить мы, достаточно стабильны и хорошо документированы. Однако даже в них не все функции, формально должные работать, уже реализованы (при их использовании, как правило, модуль вызывает исключение `NotImplementedError`). Некоторые функции, существуя в текущей версии модуля ради обратной совместимости, уже планируются к удалению в будущих версиях (это отмечают в документации, а при их использовании вызывается предупреждение `DeprecationWarning`), и так далее. Мы не будем конкурировать с документацией и пройдем мимо таких случаев.

21. Модули `math`, `cmath`, `random`

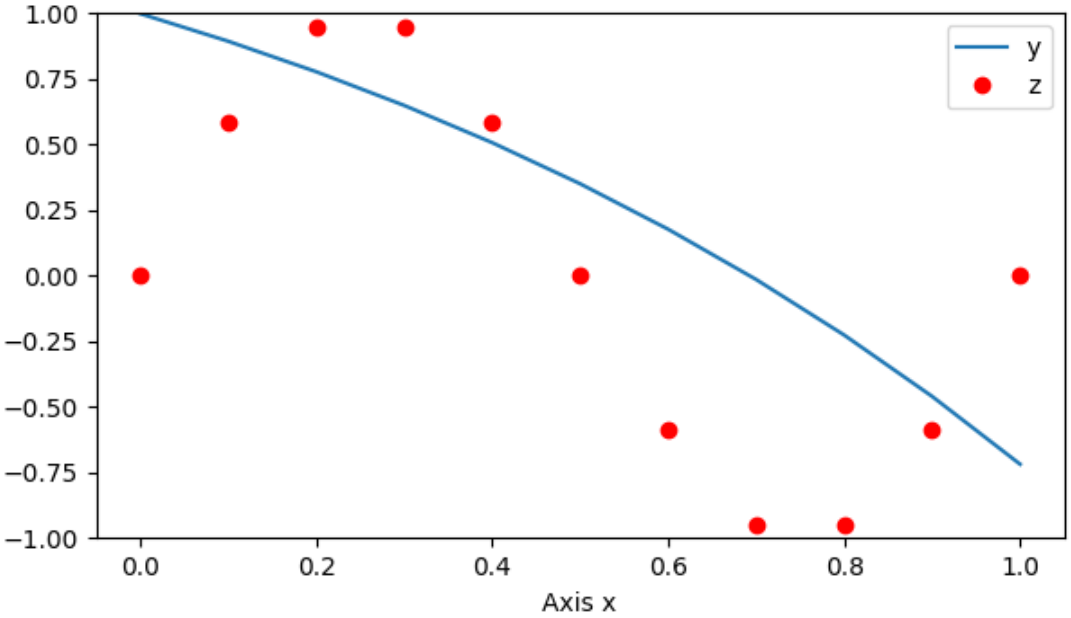
Пожалуй, это первые модули, которые понадобятся в работе. Модули `math` и `cmath` определяют почти одинаковый набор функций вещественных и комплексных переменных, соответственно. Как обычно, `dir` и `help` расскажут подробно про весь функционал. В обоих модулях определены константы `e` и `pi` (e и π), экспонента, логарифмы, тригонометрические и гиперболические функции. Функция `isclose` с настраиваемыми погрешностями нужна, чтобы заменить оператор сравнения `==`, не пригодный для вещественных чисел. В `cmath` также определены функции `polar`, `rect` и `phase` для преобразования комплексных чисел из декартовых в полярные координаты и обратно. В `math` определены преобразование градусов в радианы и наоборот, некоторые функции целых аргументов (факториал, число сочетаний и т. д.) и ряд специальных функций (гамма-функция, функция ошибок).

Модуль `random` предоставляет инструменты для генерации псевдослучайных чисел. Функция `random` (то есть вызов `xi = random.random()`) возвращает равномерно распределенное случайное вещественное число в диапазоне [0, 1), `randrange(i, j, k)` — случайное целое из набора, соответствующему объекту `range` с теми же аргументами, `randint(a, b)` — целое от `a` до `b` включительно, `normalvariate(m, s)` — распределенное нормально со средним `m` и стандартным отклонением `s`, `choice(L)` выбирает случайный элемент последовательности `L`.

22. Модуль `matplotlib`

Модуль `matplotlib` содержит инструменты для построения графиков. Рассмотрим пример работы и разберем его построчно:

```
import matplotlib.pyplot as plt          # 1
from math import sin, exp, pi            # 2
x = [0.1*i for i in range(11)]           # 3
y = [2 - exp(xx) for xx in x]            # 4
z = [sin(2*pi*xx) for xx in x]           # 5
plt.plot(x, y)                           # 6
plt.plot(x, z, 'ro')                     # 7
plt.xlabel('Axis x')                     # 8
plt.ylim(-1, 1)                          # 9
plt.legend(['y', 'z'])                   # 10
plt.show()                               # 11
```



(1) Импортируем модуль и присвоим объекту `pyplot` короткое имя. (2) Импортируем нужные нам объекты из модуля `math`. (3–5) Создадим коллекции, определяющие точки будущих графиков. (6) Отправим коллекции объекту `plt`. Коллекции должны быть одинаковой длины и содержать только числа. У комплексных чисел будет отброшена мнимая часть, при этом будет выведено предупреждение. (7) Необязательные аргументы определяют то, как будет выведен график — символы, тип линии, цвета. В частности, `'ro'` — красные (*red*) кружочки (o). (8) Метод `xlabel` (и аналогичный ему `ylabel`) подписывает ось. (9) Метод `ylim` (и аналогичный ему `xlim`) используется, чтобы определить область графика по оси ординат (абсцисс). (10) Метод `legend` принимает коллекцию строк и создает на графике легенду. (11) Наконец, когда все готово, метод `show` отображает график и останавливает поток управления, пока график не будет закрыт.

Помимо построения графика в линейном масштабе с помощью `plot`, можно строить графики в логарифмическом с помощью `semilogx`, `semilogy` и в двойном логарифмическом масштабе с помощью `loglog`. Вместо `show`, который остановит программу, пока график не будет закрыт, можно использовать функции `pause` и `waitforbuttonpress`, которые вернут поток управления в программу после

определенной паузы или нажатия кнопки клавиатуры или мыши. С помощью команд `subplot` можно расположить несколько осей в одном окошке.

Более детальные примеры использования `matplotlib` мы поместили в Приложение Ж, а еще больше примеров и справочной информации информации можно найти на странице https://matplotlib.org/cheatsheets/.

23. Модуль `time`

Время может быть представлено (i) в секундах от какого-либо начального момента (обычно используется полночь 1 января 1970 года по Гринвичу) или (ii) в виде структуры-кортежа `struct_time`, хранящей отдельно год, месяц и т. д. Получить текущее время в секундах и наносекундах можно с помощью функций `time` и `time_ns`, соответственно. Для преобразования между форматами даты-времени и строковым представлением используются функции `asctime` (преобразует кортеж в строку); `ctime` (преобразует секунды в строку), `gmtime` и `localtime` (преобразую секунды в кортеж с учетом местного времени), `mktime` (преобразует кортеж в секунды), `strptime` (преобразует строку в кортеж).

Функция `sleep(seconds)` приостанавливает программу на указанное время.

Для оценок времени выполнения кода рекомендуется использовать `perf_counter` и `perf_counter_ns`. Эти функции возвращают время с учетом наилучшего временного разрешения, доступного операционной системе (обычно менее 1 мкс), в то время как `time` может иметь на порядки более грубое разрешение.

24. Модуль `itertools`

Мы уже успели несколько раз увидеть в действии объект `zip`, который принимает на вход несколько итерируемых объектов и группирует их элементы в кортежи:

```
>>> L = [1, 2, 3, 4, 5, 6, 7, 8]
>>> M = [10, 20, 30]
>>> [x + y for x, y in zip(L, M)]
[11, 22, 33]
```

Как видно из примера, `zip` останавливается, как только *один* из переданных ему итераторов возвращает `StopIteration`. Как заставить его продолжать итерацию, пока не закончится *каждый* из итераторов? Именно для того, чтобы программист в этом случае не занимался нагромождением сомнительных и не особо универсальных решений (мы могли бы дополнить список нулями до нужной длины, но такой подход не сработает, если вместо `M` передать `set(M)`), существует модуль `itertools` и, в частности, класс `zip_longest`.

```
>>> from itertools import *
>>> [x + y for x, y
...      in zip_longest(L, M, fillvalue=0)]
[11, 22, 33, 4, 5, 6, 7, 8]
```

Пока не закончатся все переданные итераторы, `zip_longest` будет возвращать очередной кортеж, в котором на место элементов уже закончившихся итераторов будет подставляться `fillvalue` (по умолчанию `None`).

Перечислим и другие итераторы: `count(start, step)` работает как `range` с бесконечным `stop`, то есть никогда не закончится самостоятельно. В следующем примере итерация останавливается, когда истощается `L`:

```
>>> [x * y for x, y in zip(L, count(-5, 2))]
[-5, -6, -3, 4, 15, 30, 49, 72]
```

`repeat` будет возвращать переданный ему объект бесконечно или указанное число раз, `cycle` бесконечно итерирует по кругу переданный ему итерируемый объект.

```
>>> [x * y for x, y in zip(repeat(5, 8), cycle(M))]
[50, 100, 150, 50, 100, 150, 50, 100]
```

`islice` возвращает «срез» переданного ему итератора, то есть `islice(L, i, j, k)`, вообще говоря, эквивалентен `iter(list(L)[i:j:k])`, но более эффективен по времени и памяти, потому что список и его срез не создаются как отдельные объекты. `compress` переберет одновременно два итератора, подобно `zip`, но будет возвращать только элементы первого, при которых элементы второго истинны. Например, скомбинируем `compress` и `cycle`, чтобы отбросить каждый третий элемент переданной коллекции:

```
>>> L = [1, 2, 3, 4, 5, 6, 7, 8]
>>> list(       compress(L, cycle([1, 1, 0]))  )
[1, 2, 4, 5, 7, 8]
```

`chain` последовательно переберет элементы всех переданных ему итерируемых объектов:

```
>>> list(       chain(L, M)          )
[1, 2, 3, 4, 5, 6, 7, 8, 10, 20, 30]
```

`accumulate` по умолчанию накапливает сумму элементов:

```
>>> list(       accumulate(L)                  )
[1, 3, 6, 10, 15, 21, 28, 36]
```

но возможно передать ему и другие бинарные операции, и ненулевое начальное значение.

Итераторы `filter` (встроенный) и `filterfalse` принимают функцию и итератор и возвращают только те объекты из итератора, на которых функция, соответственно, истинна или ложна:

```
>>> def mod3(x): return x % 3
>>> list(       filter(mod3, L)                )
[1, 2, 4, 5, 7, 8]
>>> list(       filterfalse(mod3, L)           )
[3, 6]
```

`takewhile` также принимает функцию и итератор; он остановит итерацию, как только функция, оцененная на очередном элементе, вернет `False`. `dropwhile`, наоборот, отбросит элементы из начала последовательности, для которых функция истинна:

```
>>> list(       takewhile(mod3, L)             )
[1, 2]
>>> list(       dropwhile(mod3, L)             )
[3, 4, 5, 6, 7, 8]
```

Встроенный итератор `map` применяет функцию к аргументам, полученным из переданных итераторов:

```
>>> def mul(x, y): return x * y
>>> list(       map(mul, L, cycle([1, 1, 2]))  )
[1, 2, 6, 4, 5, 12, 7, 8]
```

Итератор `starmap` функционирует аналогично, но аргументы для функции получает не из нескольких итераторов, а при распаковке коллекции, полученной из одного итератора:

```
>>> D = {1: 2, 3: 4, 5: 6}
>>> list(       starmap(mul, D.items())        )
[2, 12, 30]
```

Безусловно, все перечисленные выше итераторы не так сложно воспроизвести и без применения `itertools`, воспользовавшись вложенными циклами, дополнительными условиями, операторами `continue` или `break` в цикле, генераторными выражениями и т. п. Так, последний пример вполне эквивалентен списковому включению:

```
>>> [mul(*x) for x in D.items()]
[2, 12, 30]
```

а сам объект `starmap` — соответствующему генераторному выражению. Тем не менее, основная функция `itertools` — сделать код более лаконичным и читаемым, сохранив возможность обрабатывать любой итерируемый объект (строку, коллекцию, генераторное выражение, генераторную функцию, итератор). С этой точки зрения бо́льшая часть приведенных примеров существенно пострадает, если воспроизвести их, используя только встроенные объекты Python.

Немного менее тривиальны итераторы `combinations`, возвращающий сочетания из `n` различных элементов переданного итератора, `combinations_with_replacement` — сочетания `n` элементов, но с возможными повторениями, и `permutations` — перестановки `n` элементов (для краткости выводов в примере всюду `n = 2`):

```
>>> list(combinations([4, 5, 6], 2)                    )
[(4, 5), (4, 6), (5, 6)]
>>> list(combinations_with_replacement([4, 5, 6], 2) )
[(4, 4), (4, 5), (4, 6), (5, 5), (5, 6), (6, 6)]
>>> list(permutations([4, 5, 6], 2)                    )
[(4, 5), (4, 6), (5, 4), (5, 6), (6, 4), (6, 5)]
```

Стоит заметить, что элементы переданной коллекции не проверяются на равенство, в чем можно убедиться, поменяв в предыдущих примерах список на `[4, 4, 4]`, например.

Наконец, `groupby` группирует элементы итерируемого объекта по значениям переданной функции, формируя отдельный итератор на каждую группу:

```
>>> L = [[], [1], [2], [3, 4], [5, 6], [7, 8, 9]]
>>> for key, group in groupby(L, len):
...     print(key, end=': ')
...     for x in group:
...         print(x, end=' ')
...     print()
0: []
1: [1] [2]
2: [3, 4] [5, 6]
3: [7, 8, 9]
```

25. Модуль numpy. Массивы и функции

Модуль `numpy` предназначен для быстрой и эффективной работы с массивами чисел, поэтому ядро библиотеки написано на C. Основной тип `ndarray` — это массивы именно в том смысле, какой обычно используется в C++ и многих других языках, то есть набор элементов одного типа, расположенных в оперативной памяти непрерывным (связным) блоком. Более того, поскольку операции Python не слишком быстрые, в `numpy`

переопределены числовые типы, как то: int32, int64, float32 (одинарная точность), float64, complex128. Размер массива также фиксирован. На чуть более глубоком уровне объект ndarray состоит из непосредственно массива данных (data buffer) и заголовка, который описывает, как с этими данными работать. Для обработки массивов в numpy определен класс универсальных функций ufunc. Перегруженные операторы, как правило, вызывают эти функции. Отдельные функции вынесены в суб-модули linalg (линейная алгебра), fft (преобразование Фурье), random и другие.

Полный обзор всех возможностей numpy превысил бы размер данного пособия, поэтому мы разберем подробно только архитектуру массивов ndarray, работу функций ufunc и способы индексации массивов. Все остальное, как обычно, доступно в документации. Для доступа к оффлайн-документации numpy следует использовать функцию numpy.info вместо стандартного help. Пример быстродействия numpy по сравнению с базовыми возможностями Python мы поместили в Приложение Е. Приложение 3 перечисляет наиболее употребимые функции numpy.

Начнем работу с того, что создадим массив ndarray на основе списка:

```
>>> import numpy as np          # общепринятое сокращение
>>> x = np.array([-3, -1, 1, 3])
>>> x
array([-3, -1,  1,  3])
```

Поскольку числа в нашем массиве образуют арифметическую прогрессию, можно было создать его и более регулярным образом:

```
>>> x = np.arange(-3, 5, 2)
>>> x = np.linspace(-3, 3, num=4, dtype='int')
```

Функция arange имеет аргументы, аналогичные объекту range, но поддерживает вещественный тип данных, функция linspace задает num чисел, расположенных равномерно на заданном отрезке (по умолчанию включая обе границы), а аргумент dtype указывает, какого типа должны быть числа в массиве. (Обычно numpy сам выбирает минимально необходимый тип данных, но linspace является исключением и по умолчанию создает массив вещественных чисел.)

Подобно спискам list, массивы ndarray изменяемы, и все проблемы ссылочной безопасности (см. главу 12 “Неочевидные следствия модели «объект—переменная» в Python”) относятся к ним в полной мере:

```
>>> z = x
>>> z[3] = 4
>>> x
array([-3, -1,  1,  4])
```

Копировать массив, чтобы независимо работать с x и z, можно с помощью метода copy или еще раз вызвав функцию array:

```
>>> z = x.copy()
>>> z = np.array(x)
```

Каковы отличия x от исходного списка [-3, -1, 1, 3]? Во-первых, x хранит не целые числа Python бесконечной точности, а 32-битные целые:

```
>>> x.dtype
dtype('int32')
>>> type(x[0])
<class 'numpy.int32'>
```

Во-вторых, он хранит не ссылки на какие-то абстрактные объекты, которые могут лежать где угодно в оперативной памяти, а связный блок из 4 чисел (соответственно, 16 байт). Размер и тип данных `x` фиксирован. Мы не можем ни присоединить новые элементы, ни записать в него произвольные объекты — при записи по индексу объект будет преобразован к `int32`. Следующие команды приведут к одному и тому же результату:

```
>>> x[-1] = 5
>>> x[-1] = 5.2
>>> x[-1] = '5'
```

а вот команды `x[-1] = '5.2'` и `x[-1] = 2**33 + 5` уже вызовут ошибки. Неявное приведение элементов к типу массива является распространенным источником ошибок при работе с комплексными массивами. Если не указывать тип явно, большая часть операций выберет для массива вещественный тип. При дальнейшей работе, если мы попытаемся записать в такой массив комплексное число, мнимая часть будет отброшена:

```
>>> x[-1] = x[-1] + 1j*x[0]
>>> x[-1] *= 1+1j
ComplexWarning: Casting complex values to real discards
the imaginary part
```

При этом приведение типов может как сопровождаться предупреждением, так и нет. В лучшем случае `numpy` явно вызовет исключение:

```
>>> x[-1] = 5 + 4j
TypeError: int() argument must be a string, a bytes-like
object or a real number, not 'complex'
```

Однако надеяться на это не вполне разумно. Если программа должна работать с комплексными массивами, программисту необходимо применять навыки работы со статической типизацией.

В-третьих, перегруженные операторы имеют для `x` иное значение, чем для списков, и применяются к операндам поэлементно, возвращая также массивы `ndarray`:

```
>>> x
array([-3, -1,  1,  5])
>>> y = np.arange(-2, 2)
>>> y
array([-2, -1,  0,  1])
>>> x + y
array([-5, -2,  1,  6])
>>> x * y
array([6, 1, 0, 5])
>>> y / x
array([0.66666667, 1.        , 0.        , 0.2       ])
>>> x > y
array([False, False,  True,  True])
```

Вообще говоря, каждому перегруженному оператору соответствует универсальная функция, которая и принимает операнды в качестве аргументов. Ее можно вызвать и явно:

```
>>> np.add(x, y)
array([-5, -2,  1,  6])
```

Это бывает удобно, если мы хотим использовать необязательные аргументы универсальных функций, хотя почти все применения относятся к случаям, когда необходимо серьезно экономить вычислительные ресурсы. Так, можно указать тип `dtype`, к которому будет преобразован результат:

```
>>> q = np.add(x, y, dtype='float')
>>> q
array([-5., -2.,  1.,  6.])
```

Вызвав преобразование типа `(x + y).astype('float')`, можно получить такой же массив, однако придется сначала выделить память под результат суммы, а потом — под результат `astype`. Можно указать массив `out`, в который будет записан результат функции (функция при этом возвращает новую ссылку на массив `out`):

```
>>> np.subtract(x, y, out=q)
array([-1.,  0.,  1.,  4.])
>>> q
array([-1.,  0.,  1.,  4.])
```

При этом, в отличие от `q = x - y`, результат будет записан в память, которая уже выделена под массив `q`, то есть программе не придется выделять блок памяти под новый массив и очищать память, занятую старым массивом. Если функция является арифметической операцией, а массив `out` совпадает с первым аргументом, то использование `out` эквивалентно инкрементному присваиванию. Команда

```
>>> np.multiply(q, x, out=q)
array([ 3., -0.,  1., 20.])
```

обеспечит тот же самый результат, что и `q *= x`. Пример быстродействия таких операций помещен в Приложение E.

Аргументами универсальных функций не обязаны быть массивы `ndarray`. В одномерном случае операнды могут быть скалярами (числовыми типами Python или `numpy`):

```
>>> np.subtract(q, 10)
array([ -7., -10.,  -9.,  10.])
>>> q - 10
array([ -7., -10.,  -9.,  10.])
>>> 1 / (q - 10)
array([-0.14285714, -0.1       , -0.11111111,  0.1     ])
>>> np.gcd(x + 19, 15)  # наибольший общий делитель
array([1, 3, 5, 3])
```

а также списками или кортежами одинаковой длины:

```
>>> [1, 2, 3, 4] / (q + [2, 2, 3, -16])
array([0.2 , 1.  , 0.75, 1.  ])
>>> np.power([4, 3, 2, 1], [1, 2, 3, 4])
array([4, 9, 8, 1])
```

В документации типы, с которыми могут работать универсальные функции (массивы `ndarray`, скаляры, списки `list` и кортежи `tuple`), вместе называются array-like objects.

Смешанное использование списков и массивов `numpy`, вообще говоря, довольно бессмысленно, потому что подрывает главное преимущество `numpy` — скорость вычислений. Практически любая работа с большими объемами данных должна выполняться исключительно средствами `numpy`. Для обработки массивов следует использовать универсальные функции вместо циклов. Конкатенация выполняется как `np.concatenate((x, y))` вместо `np.array(list(x) + list(y))`. Файловые ввод и вывод будут существенно быстрее, если использовать `load`/`loadtxt` и `save`/`savetxt` вместо обычных `f.read`/`f.write` (см. главу 19 «Работа с файлами») и так далее.

Верно и обратное: бессмысленно использовать массивы `ndarray` в качестве списков. Хотя они поддерживают итерационный протокол и даже могут хранить ссылки на произвольные объекты Python, если указать `dtype=object`, они не являются настолько гибкими динамическими структурами, как `list`. В частности, `np.append`, `np.insert`, `np.delete` будут существенно медленнее, поскольку требуют выделить новую память, куда будет (с изменениями) скопирован массив.

26. Модуль `numpy`. Многомерные массивы

Модуль `numpy` поддерживает и многомерные массивы. Создадим двумерный `ndarray` на основе списка списков:

```
>>> import numpy as np
>>> L = np.array([  [0, 1, 2, 3],
...                 [10, 11, 12, 13],
...                 [20, 21, 22, 23]])
>>> L
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Как и в предыдущей главе, при этом мы потеряли возможность хранить целые числа произвольной длины, и под наш массив выделена связная область в оперативной памяти на 48 байт (12 чисел по 4 байта).

```
>>> L.dtype
dtype('int32')
>>> L.itemsize  # байт на элемент
4
>>> L.size      # количество элементов массива
12
>>> L.nbytes    # байт всего
48
```

Поскольку адрес в оперативной памяти — это одно число, информация о том, что этот массив является матрицей 3×4, содержится лишь в заголовке массива — поле `ndim`, указывающем на размерность массива, и поле `shape`, содержащем длину по каждой размерности:

```
>>> L.ndim  # ранг массива (количество осей-размерностей)
2
>>> L.shape # кортеж длин по каждой оси-размерности
(3, 4)
```

В документации для описания одной из размерностей массива обычно используется термин ось (axis). Обратиться к одному элементу массива можно так же, как в случае списка списков:

```
>>> L[2]
array([20, 21, 22, 23])
>>> L[2][1]
21
```

хотя обычно пользуются возможностью передать индексы вместе через запятую:

```
>>> L[2, 1]
21
```

По умолчанию numpy использует конвенцию C (row-major, «быстрый — последний»). Переход к следующему элементу в линейном массиве, смещение на 4 байта, означает переход от элемента [i, j] к элементу [i, j+1] (на границе строки, разумеется, к [i+1, 0]). Для увеличения же первого индекса, т. е. перехода к элементу [i+1, j], нужно смещаться сразу на 16 байт. Эти числа можно увидеть в поле strides:

```
>>> L.strides   # +1 по первой размерности — +16 байт
(16, 4)         # адреса, +1 по второй — +4 байта
```

Таким образом, чтобы добраться до L[2][1], необходимо взять адрес первого элемента и сместиться на 2×16 + 1×4 = 36 байт. Посмотрим также поле flags, в котором из важного упомянуто, что используется конвенция C, а не Fortran[22] (C_CONTIGUOUS, но не F_CONTIGUOUS), что массив изменяемый (WRITEABLE) и это его собственные данные (OWNDATA), то есть он не является срезом (частью) какого-то другого массива.

```
>>> L.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ...
```

Существенная часть операций с массивами не копирует данные, а только нужным образом меняет заголовок. Например, возьмем диагональ L:

```
>>> D = L.diagonal()
>>> D
array([ 0, 11, 22])
```

Массив D использует ту же память, что и исходный L, но по выделенным на L 48 байтам он пробегает с шагом 20 байт, смещаясь одновременно на следующую строку и следующий столбец:

```
>>> D.strides       # элементы разделяет 20 байт
(20,)
>>> D.base is L     # D является срезом массива L
True
>>> D.shape         # одномерный массив длины 3
(3,)
```

Флаги указывают, что диагональный срез уже не является связным массивом, использует память другого массива (поле OWNDATA) и предназначен только для чтения[23] (поле WRITEABLE), поэтому мы не можем через него изменить исходный массив.

```
>>> D.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : False
...
```

[22] Конвенция Fortran (column-major, «быстрый — первый»), помимо языка Fortran, системы Matlab и других, используется в графике. Пиксельные изображения обычно хранятся строками, но при указании координаты пикселя (x, y) первой идет позиция в строке, то есть номер колонки. Соответственно, смещение в линейном массиве на размер элемента — это переход к пикселю ($x + 1$, y).

[23] Начиная с версии numpy 1.19.

Однако последнее — это скорее исключение, и другие срезы, как правило, не ограничивают запись. Рассмотрим срез массива, оставив только колонки с индексами 0 и 2:

```
>>> M = L[:, ::2]
>>> M
array([[ 0,  2],
       [10, 12],
       [20, 22]])
>>> M.strides   # в строчке шагаем сразу на 2 элемента
(16, 8)
>>> M.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ...
>>> M.base is L
True
```

Поскольку `M` использует данные `L`, его изменения отразятся и на `L`:

```
>>> M[1, 1] = 112
>>> L[1]
array([ 10,  11, 112,  13])
```

Если такое поведение нежелательно, необходимо использовать метод `copy`, как и в случае со встроенными коллекциями Python.

Помимо срезов, изменение размерности `L.reshape`,

```
>>> L.reshape(2, 6)
array([[  0,   1,   2,   3,  10,  11],
       [112,  13,  20,  21,  22,  23]])
```

транспонирование `L.T` или `L.transpose()` и некоторые другие операции не выделяют отдельную память под результат и не копируют никакие данные, а только создают новый заголовок (за счет чего работают очень быстро независимо от размера массива).

Унарные операторы и заметная часть функций не различают одно- и многомерные массивы: `abs(x)`, `-x`, `np.exp(x)` применятся поэлементно независимо от размерности `x`. Рассмотрим характерные случаи, когда это не так.

Разумеется, не является поэлементным матричное умножение `matmul` (оператор `@`). Умножение двумерных матриц ведет себя вполне предсказуемо:

```
>>> M.T              # матрица 2 × 3
array([[  0,  10,  20],
       [  2, 112,  22]])
>>> M.T @ L          # (2 × 3) @ (3 × 4) = (2 × 4)
array([[  500,   530,  1560,   590],
       [ 1560,  1696, 13032,  1968]])
```

Если первый операнд является одномерным массивом, то он представляется как строка, если второй — он представляется как столбец, а результат произведения приводится обратно к одномерному массиву.

```
>>> L @ np.array([3, 2, 1, 0])  # (3 × 4) @ (4,) = (3,)
array([  4, 164, 124])
>>> [3, 2, 1] @ L               # (3,) @ (3 × 4) = (4,)
array([ 40,  46, 252,  58])
```

Когда оба операнда одномерные, то фактически вычисляется скалярное произведение векторов:

```
>>> np.array([1, 2, 3, 4]) @ np.array([4, 3, 2, 1])
20
```

Многие функции имеют управляющий параметр `axis`. При использовании этого параметра функция независимо обрабатывает одномерные срезы, у которых оставлена только указанная ось-размерность. Нумерация осей совпадает с индексацией кортежа `shape`, в том числе `axis=-1` — это всегда последняя ось. Так, функция `sum` по умолчанию суммирует все элементы массива, но можно настроить ее, чтобы она вычисляла сумму строк или сумму столбцов:

```
>>> np.sum(L)
237
>>> np.sum(L, axis=0) # [sum(L[:, i]) for i in range(4)]
array([ 30,  33, 136,  39])
>>> np.sum(L, axis=1) # [sum(L[i, :]) for i in range(3)]
array([  6, 146,  86])
```

Функция `diff` вычисляет разности соседних элементов в каждой строке, но с помощью параметра `axis` можно заставить ее работать по столбцам:

```
>>> np.diff(L)
array([[  1,   1,   1],
       [  1, 101, -99],
       [  1,   1,   1]])
>>> np.diff(L, axis=0)
array([[ 10,  10, 110,  10],
       [ 10,  10, -90,  10]])
```

Бинарные операторы и функции многих аргументов пытаются по мере возможности согласовать размерности аргументов, если они не совпадают буквально. Массивы транслируются (broadcast) к общей размерности, если выполнено следующее: размерности двух массивов по каждой оси, начиная с последней (то есть элементы кортежа `shape`, начиная с последнего), должны либо совпадать, либо одна из них должна быть 1. Например, массив `L` в нашем примере имеет размерность `(3, 4)`. С ним будут согласованы массивы размерностей `(1,)` — скаляр, `(4,)`, `(1, 4)`, `(3, 1)`, `(3, 4)`, а также любые массивы более высокого ранга, у которых последние две оси имеют указанные размерности, например, `(100, 3, 4)`. Иными словами, для массивов `L` и `S` потребуем

```
>>> all(p == q or p == 1 or q == 1 for p, q in
            zip(reversed(L.shape), reversed(S.shape)))
```

Например, массив `L` можно складывать со скаляром, строкой длины 4 и столбцом длины 3:

```
>>> L + 50
array([[ 50,  51,  52,  53],
       [ 60,  61, 162,  63],
       [ 70,  71,  72,  73]])
```

```
>>> S = np.array([0, 2, 4, 6])
>>> L + S
array([[  0,   3,   6,   9],
       [ 10,  13, 116,  19],
       [ 20,  23,  26,  29]])
>>> S = np.array([[-20], [-10], [0]])
>>> L + S
array([[-20, -19, -18, -17],
       [  0,   1, 102,   3],
       [ 20,  21,  22,  23]])
```

В последнем примере также можно стартовать со строчки, преобразовав ее размерность:

```
>>> S = np.array([-20, -10, 0]).reshape(3, 1)
>>> S = np.array([-20, -10, 0])[:, np.newaxis]
```

Константа `np.newaxis` означает, что в срез необходимо добавить ось длины 1.

```
>>> M = np.zeros((10, 10, 10))
>>> M.shape
(10, 10, 10)
>>> M[:,:,np.newaxis,:].shape
(10, 10, 1, 10)
```

27. Модуль numpy. Срезы массивов

Мы уже использовали обращения к одному элементу `[i]` и регулярному (эквидистантному) срезу `[i:j:k]` массива `numpy` подобно тому, как работали со списками Python. Они работают как на чтение, так и на запись:

```
>>> import numpy as np
>>> x = np.arange(8)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> x[0] = 1
>>> x[2:4] = np.array([-9, -8]) # заменить [2, 3]
>>> x[4:6] += np.array([5, 4])  # [4, 5] += [5, 4]
>>> x[6:8] *= -1                # [6, 7] *= -1
>>> x
array([ 1,  1, -9, -8,  9,  9, -6, -7])
```

При присвоении срезу массива операнды должны быть согласованы по размерности, то есть их длина по каждой из осей должна либо совпадать, либо быть равной 1 у второго операнда. Обращения

```
>>> x[0] = np.array([2, 3])         # (1,) = (2,)
>>> x[0:3] = np.array([2, 3])       # (3,) = (2,)
```

вызовут ошибку из-за несовместимости размерностей.

При присвоении многомерному сечению одномерных массивов они транслируются по общим для бинарных операций правилам:

```
>>> A = np.arange(16).reshape(4, 4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
>>> A[1:, :] = np.array([100, 200, 300, 400])
>>> A
array([[  0,   1,   2,   3],
       [100, 200, 300, 400],
       [100, 200, 300, 400],
       [100, 200, 300, 400]])
>>> A[1:, ::2] = np.array([500, 600, 700])[:, np.newaxis]
>>> A
array([[  0,   1,   2,   3],
       [500, 200, 500, 400],
       [600, 200, 600, 400],
       [700, 200, 700, 400]])
```

В первом случае мы пытаемся присвоить срезу размером `(3, 4)` массив размером `(4,)`, во втором — срезу размером `(3, 2)` массив размером `(3, 1)`. Соответственно, `numpy` повторяет строку `(4,)` три раза или расширяет последнюю размерность столбца `(3, 1)`.

Помимо обращения к регулярному срезу, можно передавать в оператор `[ ]` массив или список целых индексов. Как и в предыдущем случае, второй операнд должен быть скаляром либо согласованным по размерности с запрошенным сечением:

```
>>> x = np.arange(8)
>>> x[[0, 3, 5, 6]] = -2
>>> x
array([-2,  1,  2, -2,  4, -2, -2,  7])
>>> x[[1, 2, 4, 7]] *= [-1, 2, -4, -7]
>>> x[[1, 2, 4, 7]]
array([ -1,   4, -16, -49])
>>> x
array([ -2,  -1,   4,  -2, -16,  -2,  -2, -49])
```

Наконец, существует так называемое обращение по маске, при котором в качестве аргумента передается массив логических переменных. Такие операции удобны, потому что операции сравнения массивов выполняются поэлементно и возвращают именно булевы массивы. В следующем примере мы изменяем только те элементы массива `x`, которые меньше соответствующих элементов массива `y`:

```
>>> x = np.arange(8)
>>> y = np.arange(-4, 12, 2)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> y
array([-4, -2,  0,  2,  4,  6,  8, 10])
>>> m = x < y
>>> m
array([False, False, False, False, False,  True,  True,
True])
>>> x[m] -= y[m]
>>> x
array([ 0,  1,  2,  3,  4, -1, -2, -3])
```

Маску `m` можно преобразовать в массив индексов с помощью метода `nonzero`:

```
>>> mm = m.nonzero()
>>> mm
(array([5, 6, 7], dtype=int64),)
```
Обращения `x[m]` и `x[mm]` будут полностью эквивалентны.

Универсальные функции также поддерживают обращения по маске с помощью необязательного аргумента `where` и атрибута `at`. Аргумент `where` принимает булевский массив и ограничивает применение функции только теми индексами, под которыми в массиве `where` хранится истинное значение. Например
```
>>> np.sum(x, where=x>0)
10
>>> np.multiply(x, y, out=y, where=x<0)
array([ -4,  -2,   0,   2,   4,  -6, -16, -30])
```
Атрибут `at` является функцией, которая применяется к указанным элементам массива, и записывает результат в него же. Для бинарных функций второй аргумент указывается после массива индексов:
```
>>> np.multiply.at(x, x<0, -4)
>>> x
array([ 0,  1,  2,  3,  4,  4,  8, 12])
>>> np.add.at(x, (x%4 != 0), [1, 2, 3])
>>> x
array([ 0,  2,  4,  6,  4,  4,  8, 12])
```
Отметим существенное отличие `where` от `at` в случае бинарных функций: при использовании `where` должны совпадать размерности операндов (или должна быть возможность трансляции к одной размерности), а при использовании `at` — размерность второго операнда и размерность сечения. В последнем примере сечение `(x%4 != 0)` имеет три истинных элемента, и передаваемый список также имеет длину 3.

В совокупности обращения к массивам и применение универсальных функций по регулярным срезам, массивам индексов и маскам предоставляют возможность достаточно гибко работать с содержимым массива, избегая лишнего копирования и обработки массивов в цикле с проверкой условий.

28. Модуль `sympy`. Символьные выражения

Почти все обсуждавшееся выше, в том числе модуль `numpy`, предназначено для работы с целыми числами или числами с плавающей точкой. Числа с плавающей точкой хранят конечное число двоичных разрядов и, таким образом, представляют числа точно, только если они являются не слишком длинными двоичными дробями. При попытке хранить дробь со знаменателем, не являющимся степенью двойки, числа представлены приближенно, следствием чего является следующий хрестоматийный пример:
```
>>> u, v, w = 1.1, 2.2, 3.3
>>> print(u, v, w)
1.1 2.2 3.3
>>> u + v == w
False
>>> u + v - w
4.440892098500626e-16
```
Именно из-за этого обстоятельства в `math` и `numpy` введены функции `isclose`.

Для точных вычислений и работы с математическими выражениями (аналитическое дифференцирование, интегрирование, решение уравнений и т. д.) разработаны системы символьных вычислений. Модуль sympy предоставляет широкие возможности в этой области.

Импортируем модуль sympy

```
>>> import sympy as sm
```

и исправим неточность в предыдущем примере:

```
>>> u = sm.Rational(11, 10)
>>> v = sm.Integer(22)/10
>>> w = sm.S('33/10')
>>> print(u, v, w)
11/10 11/5 33/10
>>> u + v == w
True
>>> u + v - w
0
```

Здесь u, v, w являются объектами класса рациональных чисел Rational, хотя мы и определили их тремя разными способами:

u. непосредственно вызвав конструктор и передав ему числитель и знаменатель,

v. с помощью символьного деления целого числа Integer на целое число Python (для объектов sympy перегружены арифметические операции, в том числе деление; перегруженный метод принимает в качестве второго операнда целое число int и преобразует его в Integer(10) самостоятельно),

w. использовав функцию sympy.S, которая преобразует объект (им может быть как число Python, так и строка) к наиболее подходящему символьному выражению.

Все арифметические действия над символьным объектом обрабатываются символьно. Это относится, разумеется, и к более сложным математическим операциям, например, вычислению корней:

```
>>> x = sm.sqrt(sm.S(5))
>>> x
sqrt(5)
>>> x**2
5
>>> y = 5 ** sm.S('1/4')
>>> y
5**(1/4)
>>> y**4
5
```

Подчеркнем, что хотя u и y выводятся в виде '11/10' и '5**(1/4)', соответственно, в коде эти выражения будут восприняты интерпретатором как встроенные числа Python, а не как символьные выражения:

```
>>> 5**(1/4)
1.4953487812212205
>>> sm.S(5)**(1/4)
1.49534878122122
```

Чтобы выражения обрабатывал sympy, необходимо, чтобы к нему обращалась старшая по приоритету операция:

```
>>> sm.S('5**(1/4)')
5**(1/4)
>>> 5**(sm.S(1)/4)
5**(1/4)
>>> 5**(1/sm.S(4))
5**(1/4)
```

В качестве альтернативы можно вызывать функцию корня произвольной степени `root`

```
>>> sm.root(5, 4)
5**(1/4)
```

Необходимо помнить, что при вычислении корней из отрицательных чисел `sympy` переходит в комплексную плоскость:

```
>>> sm.S(-32)**sm.S('1/5')
2*(-1)**(1/5)
>>> sm.root(-32, 5)
2*(-1)**(1/5)
>>> complex(sm.root(-32, 5))
(1.618033988749895+1.1755705045849463j)
```

Избежать этого можно, либо указав, который из комплексных корней необходим, либо воспользовавшись функцией `real_root`:

```
>>> sm.root(-32, 5, 2)      # третий корень
-2
>>> sm.real_root(-32, 5)
-2
```

Если в выражении присутствуют числа с плавающей точкой, то есть конечной точности, все числовые коэффициенты также становятся числами с плавающей точкой:

```
>>> sm.sqrt(5)/2, sm.sqrt(5)/2., sm.sqrt(5)/2/2.
(sqrt(5)/2, 0.5*sqrt(5), 0.25*sqrt(5))
```

Из-за этого, в частности, для работы с комплексными числами нельзя использовать стандартный синтаксис Python (тип `complex` имеет двойную точность, и мнимая единица `1j` будет проинтерпретирована как число двойной точности), и необходимо явно использовать для мнимой единицы константу `I`:

```
>>> sm.S(1j) * 5 / 2, sm.I *5 / 2
(2.5*I, 5*I/2)
```

Помимо мнимой единицы `I`, в `sympy` объявлены константы `E`, `pi`, положительная вещественная бесконечность `oo`, комплексная бесконечность `zoo` и некоторые другие.

```
>>> print(sm.pi, float(sm.pi), sm.E, float(sm.E))
pi 3.141592653589793 E 2.718281828459045
>>> print(sm.sin(sm.pi/3), sm.log(sm.E))
sqrt(3)/2 1
>>> print(sm.oo, float(sm.oo), sm.oo > 1)
oo inf True
```

В отличие от числовых объектов (например, `float`), которые хранят некоторое значение, символьное выражение, представляемое объектом `sympy`, — это «инструкция» по вычислению требуемого значения. Так, тип выражения `y = 5 ** sm.S('1/4')` — это класс `Pow`, возведение в степень:

```
>>> type(y)
<class 'sympy.core.power.Pow'>
```

аргументами которого были некоторые символьные выражения:

```
>>> y.args
(5, 1/4)
>>> type(y.args[0]), type(y.args[1])
(<class 'sympy.core.numbers.Integer'>, <class
'sympy.core.numbers.Rational'>)
```

Вложенность символьных выражений не ограничена. Не всегда, впрочем, возведение в степень вернет именно объект класса `Pow`. Если выражение можно упростить, конструктор это сделает, и, например, при возведении числа в целую степень вернет число:

```
>>> type(sm.S(5) ** sm.S(4))    # 5**4 == 625
<class 'sympy.core.numbers.Integer'>
```

Это верно и для других операций:

```
>>> x*y          # вызывается конструктор Mul(x, y)
5**(3/4)         # но возвращается Pow(5, Rational(3, 4))
```

Аналогично, выражение

$$\frac{\sqrt{7}-5}{2}$$

будет преобразовано к

```
>>> q = (sm.sqrt(7)-5)/2
>>> q
-5/2 + sqrt(7)/2
```

Для того, чтобы увидеть символьное выражение в более привычном виде с радикалами, дробями и т. д., можно использовать функцию модуля `sympy` `pprint`:

```
>>> sm.pprint(q)
  5   √7
- ─ + ──
  2   2
```

Существует возможность вывести формулу в нотации пакета LaTeX с помощью функции `latex`:

```
>>> print(sm.latex(q))
- \frac{5}{2} + \frac{\sqrt{7}}{2}
```

Для использования результатов, полученных средствами `sympy`, другими модулями (например, `numpy`, `matplotlib`), символьные выражения можно преобразовать в вещественные числа с помощью метода `evalf`[24]:

```
>>> q.evalf()
-1.17712434446770
```

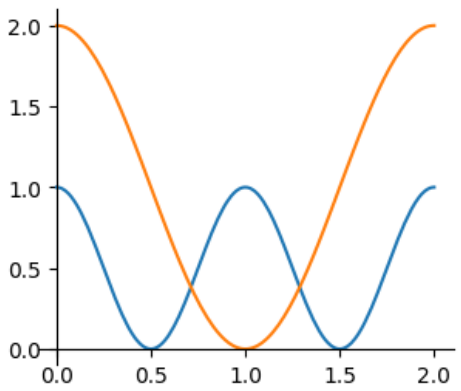
Поскольку символьное выражение можно понимать как инструкцию по вычислению некоторого числа, части этой инструкции могут быть переменными в математическом смысле слова. Действительно, значение вычисляется не в момент создания символьного выражения, а только при явном вызове метода `evalf`. Символьные переменные объявляются как экземпляры класса `Symbol`:

```
>>> z = sm.symbols('z')     # или from sympy.abc import z
>>> z * 2024
2024*z
```

[24] Метод `expr.evalf` (evaluate float) дублирован методом `expr.n` и функцией `sympy.N(expr)`.

```
>>> type(z)
<class 'sympy.core.symbol.Symbol'>
```
На математические переменные можно накладывать ограничения, явно указывая, являются ли они целыми, вещественными и т. д.:
```
>>> m = sm.symbols('m', integer=True)
>>> t = sm.symbols('t', positive=True)
>>> x = sm.symbols('x', real=True)
>>> A, B = sm.symbols('A B', commutative=False)
```
Эти ограничения влияют на обработку символьных выражений:
```
>>> sm.cos(sm.pi*x), sm.cos(sm.pi*m)
(cos(pi*x), (-1)**m)
>>> t - abs(t), x - abs(x)
(0, x - Abs(x))
>>> x**2 - abs(x)**2, z**2 - abs(z)**2
(0, z**2 - Abs(z)**2)
>>> m*A - A*m, A*B - B*A
(0, A*B - B*A)
```
и в сложных расчетах могут стать абсолютно необходимыми для упрощения результата.

Пока символьное выражение содержит переменные, его преобразование в число, разумеется, невозможно. Численные значения можно получить, передав конкретные значения переменных:
```
>>> f = sm.cos(sm.pi*x)
>>> f.evalf(subs={x: 1/3})
0.500000000000000
```
Если эту процедуру необходимо повторять много раз (например, чтобы построить график функции), можно преобразовать символьное выражение в функцию `numpy`, которая сможет производить вычисления только с двойной точностью, но быстро и сразу с массивом входных данных.
```
>>> F = sm.lambdify(x, f)
# почти то же самое, что
# F = lambda x: numpy.cos(numpy.pi*x)
>>> F(1/3)                          # обратите внимание
0.5000000000000001                 # на потерю точности
>>> from numpy import array
>>> F(array([0, 1/3, 1, 1.5]))
array([ 1.0000000e+00,  5.0000000e-01, -1.0000000e+00,
-1.8369702e-16])
```
Для построения графика функции одной переменной, заданной символьным выражением, можно использовать функцию `plot`:
```
>>> sm.plot(f**2, f + 1, (x, 0, 2))
```
Здесь последний из аргументов функции `plot` определяет область значений `x`, в которой будет построен график. С помощью необязательных аргументов `plot` можно настроить внешний вид графика, число точек и т. д., хотя проще это сделать, используя `numpy` и `matplotlib`.



29. Модуль sympy. Преобразование выражений

Достаточно регулярно символьные вычисления используются для упрощения выражений. В самых простых случаях (например, `sm.cos(sm.pi)`) преобразования будут выполнены конструктором. Однако в большинстве случаев выражение будет преобразовано или обработано только по явному вызову той или иной функции. Вычислим, например, значение следующего выражения:

$$\frac{\sqrt{11+\sqrt{3}}}{\sqrt{59}}\times\sqrt{4+\sqrt{5+\sqrt{3}}}\times\sqrt{3+\sqrt{5+\sqrt{5+\sqrt{3}}}}\times\sqrt{3-\sqrt{5+\sqrt{5+\sqrt{3}}}}$$

```
>>> import sympy as sm
>>> X = (sm.sqrt(11+sm.sqrt(3)) / sm.sqrt(59)
...      * sm.sqrt(4+sm.sqrt(5+sm.sqrt(3)))
...      * sm.sqrt(3+sm.sqrt(5+sm.sqrt(5+sm.sqrt(3))))
...      * sm.sqrt(3-sm.sqrt(5+sm.sqrt(5+sm.sqrt(3))))    )
```

Воспользуемся функцией `simplify` для его упрощения:

```
>>> sm.simplify(X)
sqrt(2)
```

Теперь рассмотрим немного более сложное выражение

$$\frac{\sqrt{\dfrac{9-2\sqrt{3}}{\sqrt{3}-\sqrt[3]{2}}+3\sqrt[3]{2}}}{3+\sqrt[6]{108}},$$

тождественно равное $1/\sqrt[4]{3}$. Присвоим символьной переменной `X` соответствующее значение

```
>>> X = (sm.sqrt((9 - 2*sm.sqrt(3)) /
            (sm.sqrt(3) - 2**sm.S('1/3'))
           + 3 * 2**sm.S('1/3')) /
           (3 + 108**sm.S('1/6')) )
```

В данном случае вызов функции `simplify`

```
>>> sm.simplify(X)
sqrt(-2*sqrt(3) - 3*2**(1/3)*(-sqrt(3) + 2**(1/3)) + 9)/
(sqrt(-2**(1/3) + sqrt(3))*(2**(1/3)*sqrt(3) + 3))
```

не ведет к искомому ответу. Таким образом, пользователю необходимо каким-то образом упростить выражение, чтобы `sympy` дальше справился с задачей. Попробуем, например, избавиться от общего корня в числителе:

```
>>> sm.simplify(X**2)
(-9 + 3*2**(1/3)*(-sqrt(3) + 2**(1/3)) + 2*sqrt(3))/
((-sqrt(3) + 2**(1/3))*(2**(1/3)*sqrt(3) + 3)**2)
```

Этого оказывается недостаточно. Теперь раскроем скобки с помощью функции `expand`:

```
>>> sm.simplify(sm.expand(X**2))
(-532*2**(1/3)*sqrt(3) + 117*sqrt(3) +
345*2**(2/3)*sqrt(3))/(-1596*2**(1/3) + 351 +
1035*2**(2/3))
```

```
>>> sm.pprint(sm.simplify(sm.expand(X**2)))
      3 __                      2/3
- 532·╲╱ 2 ·√3 + 117·√3 + 345·2    ·√3
──────────────────────────────────────
            3 ___                 2/3
    - 1596·╲╱ 2  + 351 + 1035·2
```

Данное выражение все еще далеко от искомого ответа, но, по крайней мере, теперь мы получили единственную дробь, у которой можно попытаться найти и сократить общие множители в числителе и знаменателе с помощью функции `cancel`:

```
>>> sm.cancel(sm.simplify(sm.expand(X**2)))
sqrt(3)/3
```

Теперь необходимо вернуться обратно к выражению для `X`, взяв квадратный корень:

```
>>> sm.sqrt(sm.cancel(sm.simplify(sm.expand(X**2))))
3**(3/4)/3          # == 3**sm.S('-1/4')
```

Ответ получен.

Внутренняя логика работы `simplify` такова, что с помощью некоторого перебора математических методов она пытается получить более «короткое» выражение, чем исходное. Однако на предыдущем примере мы увидели, что ей не всегда удается найти «направление», в котором следует выполнять преобразования. В таких случаях используются более специализированные функции: `expand` (раскрыть скобки), `factor` (разложить выражение на множители), `cancel` (сократить дробь), `together` (привести дроби к общему знаменателю), `apart` (представить дробь в виде суммы более простых) и т. д., поскольку эти алгоритмы четко определены.

Модуль `sympy` содержит также много специализированных функций, предназначенных для работы с конкретными выражениями. Часть из них можно увидеть, выполнив следующие команды (вывод опустим для краткости):

```
>>> [name for name in dir(sm) if 'simp' in name]
>>> [name for name in dir(sm) if 'expand' in name]
```

Например, функция `expand_trig` «раскроет скобки» тригонометрических функций, чем можно воспользоваться для представления $\sin(\pi/16)$ в алгебраической форме:

```
>>> sm.sin(sm.pi/16)
sin(pi/16)
>>> sm.expand_trig(sm.sin(sm.pi/16))
sqrt(1/2 - sqrt(sqrt(2)/4 + 1/2)/2)
```

а функция `expand_complex` явным образом выделит зависимости от действительной и мнимой частей комплексного числа:

```
>>> z = sm.symbols('z')
>>> sm.expand_complex(sm.exp(z))
I*exp(re(z))*sin(im(z)) + exp(re(z))*cos(im(z))
```

Мощным инструментом в символьных вычислениях является подстановка выражений, которую реализует метод `subs`. Например, если необходимо преобразовать выражение

$$\sin^6\frac{a}{2}-\cos^6\frac{a}{2}$$

таким образом, чтобы избавиться от дробей в аргументах тригонометрических функций, ни `symplify`, ни `expand_trig` не приведут к искомому результату. Применим подстановку:

```
>>> a = sm.symbols('a', real=True)
>>> P = sm.sin(a/2)**6 - sm.cos(a/2)**6
>>> T = P.subs([(  sm.sin(a/2)**2, (1-sm.cos(a))/2  ),
...             (  sm.cos(a/2)**2, (1+sm.cos(a))/2  )])
>>> T
(1/2 - cos(a)/2)**3 - (cos(a)/2 + 1/2)**3
```

(Символьные выражения неизменяемые, и `subs` возвращает новый объект, в котором выполнена подстановка, не изменяя исходное символьное выражение `P`.) После этого раскрытие скобок и приведение подобных может быть выполнено стандартно:

```
>>> sm.simplify(T)
(sin(a)**2 - 4)*cos(a)/4
```

Теперь попробуем упростить выражение

$$\frac{\sin 13a + \sin 14a + \sin 15a + \sin 16a}{\cos 13a + \cos 14a + \cos 15a + \cos 16a}.$$

```
>>> sin, cos = sm.sin, sm.cos
>>> S = (sin(13*a) + sin(14*a) + sin(15*a) + sin(16*a)
... )/(cos(13*a) + cos(14*a) + cos(15*a) + cos(16*a))
```

Получить короткое выражение, используя `simplify` и `expand`, в данном случае нам не удалось:

```
>>> sm.simplify(sm.expand(S, trig=True))
# выражение более чем на 10 строк
```

Однако для человека симметрия исходного выражения очевидна. Подчеркнем ее для `sympy`, выделив среднее значение аргументов синуса и косинуса (`29*a/2`) в явном виде. Для этого прибегнем к вспомогательным переменным `b` и `c`, которые мы будем полагать равными `a` (но `sympy` об этом не знает и не будет приводить их как подобные):

```
>>> b, c = sm.symbols('b c', real=True)
>>> P = S
>>> for i in range(13, 17):
...     P = P.subs(i*a, 29*b/2 + (2*i-29)*c/2)
>>> P
(sin(29*b/2 - 3*c/2) + sin(29*b/2 - c/2) + sin(29*b/2 +
c/2) + sin(29*b/2 + 3*c/2))/(cos(29*b/2 - 3*c/2) +
cos(29*b/2 - c/2) + cos(29*b/2 + c/2) + cos(29*b/2 +
3*c/2))
>>> sm.expand_trig(P)
(2*sin(29*b/2)*cos(c/2) + 2*sin(29*b/2)*cos(3*c/2)) /
(2*cos(29*b/2)*cos(c/2) + 2*cos(29*b/2)*cos(3*c/2))
>>> sm.simplify(sm.expand_trig(P)).subs(b, a)
tan(29*a/2)
```

30. Модуль `sympy`. Уравнения, математический анализ, линейная алгебра

Перейдем к решению уравнений и систем уравнений. Функция `solve` ищет корни переданного ей символьного выражения (или уравнения, если выражение явно задано как экземпляр класса `sympy.Eq`) и возвращает их в виде списка. Начнем с алгебраического уравнения

$$10x - \frac{1}{\sqrt{x^2+1}} = 0.$$

```
>>> import sympy as sm
>>> x = sm.symbols('x', real=True)
>>> sm.solve(10*x - 1/sm.sqrt(x**2+1))
[sqrt(-1/2 + sqrt(26)/10)]
```

Здесь мы видим еще один пример того, что ограничения на переменную важны. Если не ограничить `x` вещественной осью, `solve` найдет два корня:

```
>>> z = sm.symbols('z')
>>> sm.solve(10*z - 1/sm.sqrt(z**2+1))
[-I*sqrt(1/2 + sqrt(26)/10), sqrt(-1/2 + sqrt(26)/10)]
```

один из которых, как легко видеть, чисто мнимый. Впрочем, не всегда комплексные корни отбрасываются корректно:

```
>>> sm.solve(x**5 + 32)
[-2, -sqrt(5)/2 + 1/2 - sqrt(5)*I*sqrt(5/8 - sqrt(5)/8) -
I*sqrt(5/8 - sqrt(5)/8), # и другие комплексные корни
...]
```

Если символьное выражение, корни которого необходимо найти, содержит несколько переменных, то `sympy` может «решить» уравнение относительно любого из них:

$$x^2 - 2ax + 1 = 0$$

```
>>> a = sm.symbols('a', real=True)
>>> sm.solve(x**2 - 2*a*x + 1)
[{a: (x**2 + 1)/(2*x)}]
```

Поэтому необходимо явно указать переменную, которую надо выразить:

```
>>> sm.solve(x**2 - 2*a*x + 1, x)
[a - sqrt(a**2 - 1), a + sqrt(a**2 - 1)]
```

Для решения систем уравнений необходимо передать функции `solve` список выражений или уравнений `sympy.Eq`:

$$\begin{cases}(x+y)(x^2-y^2) = 16\\(x-y)(x^2+y^2) = 40\end{cases}$$

```
>>> x, y = sm.symbols('x y', real=True)
>>> sm.solve([sm.Eq((x+y)*(x**2-y**2), 16),
...           sm.Eq((x-y)*(x**2+y**2), 40)])
[{x: 1, y: -3}, {x: 3, y: -1}]
```

В этом случае `solve` вернет список словарей, в которых ключами будут переменные, относительно которых решена система (символьные выражения неизменяемы и хешируемы, поэтому могут быть ключами словаря).

Следует, однако, иметь в виду, что иногда решение может быть получено только в специальных функциях:

$$ax - \exp(-x^2) = 0$$

```
>>> sm.solve(a*x - sm.exp(-x**2), x)
[exp(-LambertW(2/a**2)/2)/a]
```

(функция Ламберта `LambertW` не может быть выражена в элементарных функциях). Весьма вероятно, что таким решением все равно можно будет пользоваться —

продифференцировать, получить численное значение, построить график и т. д. Но в некоторых случаях решение, которое заведомо существует, вообще не будет найдено:

$$10x - \cos(x) = 0$$

```
>>> sm.solve(10*x - sm.cos(x))
NotImplementedError: multiple generators [x, cos(x)]
No algorithms are implemented to solve equation 10*x -
cos(x)
```

В этом случае можно найти численное решение, для получения которого существует функция `nsolve`, принимающая символьное выражение, зависящее только от одной переменной, и начальное приближение для корня:

```
>>> sm.nsolve(10*x - sm.cos(x), 0)
0.0995053426873878
```

Модуль `sympy` позволяет автоматизировать рутинные операции математического анализа. Для примера выберем функцию

$$f(x) = \frac{x+1}{\sqrt{x^2+1}}.$$

Стоит отметить, что не слишком удачно было бы определить функцию `f` с помощью оператора `def`. Если бы мы сделали так, то не смогли бы использовать `sympy` для работы с этой функцией. Вместо этого определим `f` как соответствующее символьное выражение:

```
>>> f = (x+1)/sm.sqrt(x**2 + 1)
```

Если же нам потребуются значения $f(x)$ в какой-то конкретной точке *x*, мы просто выполним подстановку `f.subs(x, ...)`. Перечислим классические операции математического анализа:

1. Дифференцирование:

```
>>> sm.diff(f, x)   # первая производная по переменной x
-x*(x + 1)/(x**2 + 1)**(3/2) + 1/sqrt(x**2 + 1)
>>> f.diff(x, 2)    # вторая производная
(-2*x + (x + 1)*(3*x**2/(x**2 + 1) - 1))/(x**2+1)**(3/2)
```

2. Разложение в ряд Тейлора:

```
>>> f.series() # по умолчанию: 6 членов относительно x=0
1 + x - x**2/2 - x**3/2 + 3*x**4/8 + 3*x**5/8 + O(x**6)
>>> f.series(n=3)   # три члена ряда относительно x=0
1 + x - x**2/2 + O(x**3)
>>> f.series(x0=1)  # 6 членов относительно x=1
sqrt(2) - sqrt(2)*(x - 1)**2/8 + sqrt(2)*(x - 1)**3/8
- 9*sqrt(2)*(x - 1)**4/128 + sqrt(2)*(x - 1)**5/64
+ O((x - 1)**6, (x, 1))
```

Для остаточного члена формулы Тейлора `sympy` использует нотацию О-большого. Убрать остаточный член из выражения можно с помощью метода `f.series().removeO()`.

3. Нахождение пределов:

```
>>> sm.limit(f, x, sm.oo)
1
```

4. Нахождение первообразной:

```
>>> sm.integrate(f, x)
sqrt(x**2 + 1) + asinh(x)
```

5. Нахождение определенных интегралов:

```
>>> sm.integrate(f, (x, 0, 1))
-1 + log(1 + sqrt(2)) + sqrt(2)
```

в том числе несобственных:

```
>>> sm.integrate(sm.exp(-x**2), (x, 0, sm.oo))
sqrt(pi)/2
```

Подобно решениям уравнений, не каждый интеграл может быть выражен в элементарных функциях:

```
>>> sm.integrate(sm.sin(x)/x**2, x)
-log(x) + log(x**2)/2 + Ci(x) - sin(x)/x
```

(`Ci` — интегральный косинус), но в случае, когда `sympy` не может выразить решение даже в специальных функциях, `integrate` возвращает формально объявленный интеграл:

```
>>> sm.integrate(sm.sin(x+x**2)/x, x)
Integral(sin(x**2 + x)/x, x)
```

С этим выражением можно продолжать вычисления, в частности, результатом его дифференцирования будет подынтегральное выражение.

Чтобы решать дифференциальные уравнения (ограничимся обыкновенными), необходимо объявить символьную функцию следующим образом:

```
>>> y = sm.Function('y')(x) # x становится аргументом y
```

Используя функцию `dsolve`, решим уравнение

$$\frac{dy(x)}{dx} - 2\sqrt{y(x)} = 0.$$

```
>>> sol = sm.dsolve(sm.diff(y, x) - 2*sm.sqrt(y))
>>> sol
Eq(y(x), C1**2/4 + C1*x + x**2)
>>> sm.pprint(sol.subs('C1', 2*a))
        2            2
y(x) = a  + 2·a·x + x
```

Извлечь из равенства `Eq` правую часть можно, обратившись к полю `rhs` (right-hand side):

```
>>> Y = sol.rhs
>>> Y
C1**2/4 + C1*x + x**2
```

Разумеется, и при решении дифференциальных уравнений достаточно часто будут получаться решения в специальных функциях или ошибки `NotImplementedError`, если решение найти не удалось.

Помимо одиночных символьных выражений, в `sympy` существует класс матриц `Matrix`. Матрица является массивом символьных выражений. Однако матрицы всегда двумерны (и если конструктору передан одномерный список, он преобразуется в столбец):

```
>>> v = sm.Matrix([sm.sin(x), sm.cos(x)])
>>> v
Matrix([
[sin(x)],
[cos(x)]])
>>> v.T
Matrix([[sin(x), cos(x)]])
>>> print(v.shape, v.T.shape)
(2, 1) (1, 2)
```

```
>>> a, b, c, d = sm.symbols('a:d', real=True)
>>> A = sm.Matrix([[a, b], [c, d]])
>>> A
Matrix([
[a, b],
[c, d]])
```

Матрицы поддерживают индексацию и срезы подобно массивам `numpy`:

```
>>> A[1, :]
Matrix([[c, d]])
>>> A[1, 0]
c
```

Однако, если в оператор `[ ]` передана не пара чисел, а один индекс, он воспринимается не как срез, а как сквозная нумерация:

```
>>> [A[i] for i in range(4)]
[a, b, c, d]
```

Матрицы изменяемы (в отличие от символьных выражений):

```
>>> A[1] = -b
>>> A
Matrix([
[a, -b],
[c,  d]])
```

Но подстановки через `subs` и прочие функции `sympy`, выполняющие преобразования выражений, вернут независимую копию матрицы.

Для матриц перегружены операторы сложения и вычитания, а также умножения и возведения в степень (матричного, не поэлементного; поэлементное умножение определено в отдельных функциях). Все они строго следят за размерностью операндов в математическом смысле.

```
>>> A * v
Matrix([
[a*sin(x) - b*cos(x)],
[c*sin(x) + d*cos(x)]])
>>> A**2
Matrix([
[a**2 - b*c,  -a*b - b*d],
[ a*c + c*d, -b*c + d**2]])
>>> A + v
sympy.matrices.common.ShapeError: Matrix size mismatch:
(2, 2) + (2, 1)
>>> v**2
sympy.matrices.common.NonSquareMatrixError
```

В том числе реализовано «возведение в отрицательную степень», то есть взятие обратной матрицы:

```
>>> A.inv() == A**(-1)
True
>>> A.inv()
Matrix([
[ d/(a*d + b*c), b/(a*d + b*c)],
[-c/(a*d + b*c), a/(a*d + b*c)]])
```

Из широкоупотребимых методов матриц отметим определитель `A.det()` и нахождение собственных значений и векторов `A.eigenvects()`.

31. Коротко о других модулях

Чтобы сориентировать читателя в море возможностей, которые предоставляет Python, коротко упомянем о нескольких других модулях. Модуль `scipy` является «расширением» `numpy` и предоставляет методы численного решения дифференциальных уравнений, пакет линейной алгебры, класс разреженных матриц, статистические распределения и многое другое. Например, поисковый запрос «как посчитать матричную экспоненту на Python» выведет вас прямо на модуль `scipy`.

Модуль `pandas` также основан на `numpy` и предназначен для обработки и анализа баз данных, то есть когда данные представляют не математические объекты (матрицы, тензоры), а, например, таблицу сведений о сотрудниках организации.

Совершенствование вычислительной техники и, в частности, создание многопроцессорных графических карт, привело к бурному развитию методов машинного обучения на основе искуственных нейронных сетей. Нейронную сеть можно представить как граф, состоящий из узлов (нейронов), связаных друг с другом ребрами. Для работы с нейронными сетями на языке программирования Python разработан модуль `scikit-learn` (`sklearn`), в состав которого входят различные алгоритмы, в том числе предназначенные для решения задач классификации, регрессионного и кластерного анализа данных, включая метод опорных векторов, метод случайного леса, алгоритм усиления градиента, метод k-средних и DBSCAN. Модуль `scikit-learn` разработан с опорой на `numpy` и рассчитан на взаимодействие с `pandas`, `scipy` и др.

Модуль `PyTorch` (`torch`) подобен `numpy` (синтаксис и свойства базового класса `tensor` аналогичны `ndarray`), но приспособлен к вычислениям на графических картах. Этот модуль также широко используется для работы над нейронными сетями.

Стандартный модуль `tkinter` отвечает за графический интерфейс (в частности, среда разработки IDLE написана с помощью этой библиотеки). Работа с ним похожа на любой другой графический интерфейс (Visual C, Delphi, графика в Matlab и т. д.) — он позволяет реализовать окна, кнопки, поля ввода, меню программы и т. д.

Модуль `numba` предоставляет средства для компиляции отдельных функций, что позволяет существенно ускорить вычисления. Так называемый just-in-time компилятор (`@njit`) при первом вызове функции определяет типы всех задействованных в ней переменных и фактически подменяет код Python на статически типизированный низкоуровневый код. Компилятор `numba` может работать с числами, массивами `numpy`, и лишь немногими другими типами, и поэтому основной его функционал — (внушительное) ускорение математических вычислений, особенно тех, которые имеют принципиально цикловой характер и не могут быть реализованы через универсальные функции `numpy`.

Модули `pyserial` и `pyvisa` предоставляют интерфейсы для работы с различными портами и различным оборудованием (например, если необходимо управлять платой Arduino, подключенной к USB-порту). Кроме того, достаточно многие производители предоставляют библиотеки Python для управления своим оборудованием.

Литература

Основная литература

[1] А. Н. Васильев. «Python на примерах. Практический курс по программированию». СПб.: Наука и техника, 2016. — 432 с.

[2] Д. Хеллман. «Стандартная библиотека Python 3: справочник с примерами» 2-е изд. Пер. с англ. — СПб. : ООО «Диалектика», 2019. — 1376 с.

[3] Н. Седер. «Python. Экспресс-курс» 3-е изд. — СПб.: Питер, 2019. — 480 с.

[4] М. Лутц. «Изучаем Python», в двух томах, 5-е изд.: СПб.: ООО «Диалектика», 2019–2020.

[5] М. Саммерфилд. «Программирование на Python 3. Подробное руководство». — Пер. с англ. — СПб.: Символ-Плюс, 2009. — 608 с.

[6] С. С. Задорожный, Е. П. Фадеев. «Объектно-ориентированное программирование на языке Python» — М., Физический факультет МГУ имени М. В. Ломоносова, 2022. — 40 с.

https://cmp.phys.msu.ru/sites/default/files/ООП_на_Python_Учебное%20пособие_var7.pdf

Дополнительная литература

[7] Дж. Вандер Плас. «Python для сложных задач: наука о данных и машинное обучение» — СПб.: Питер, 2018. — 576 с.

[8] Т. Антао. «Сверхбыстрый Python. Эффективные техники для работы с большими наборами данных», пер. с англ. А. Ю. Гинько. — М.: ДМК Пресс, 2023. — 370 с.

[9] С. Рашка, В. Мирджалили. «Python и машинное обучение: машинное и глубокое обучение с использованием Python, scikit-learn и TensorFlow 2», 3-е изд. — СПб.: ООО «Диалектика», 2020. — 848 с.

[10] М. Яворски, Т. Зиаде. «Python. Лучшие практики и инструменты». — СПб.: Питер, 2021. — 560 с.

[11] У. Маккинли. «Python и анализ данных» — М.: ДМК Пресс, 2015. — 482 с.

Литература по C/C++

[12] С. Прата. «Язык программирования C++. Лекции и упражнения» Пер. с англ. — СПб.: ООО «Диалектика», 2018. — 1248 с.

[13] В. А. Антонюк и др. «Язык программирования Си. Учебно-методическое пособие (I семестр)». М.: Физический факультет МГУ имени М. В. Ломоносова, 2022. — 108 с.

https://cmp.phys.msu.ru/sites/default/files/MetC_complete.pdf

[14] С. А. Шленов, А. А. Лукашев. «Язык программирования Си++. Учебно-методическое пособие по дисциплине Программирование и информатика». — М.: Физический факультет МГУ имени М. В. Ломоносова, 2016. — 56 с.

https://istina.msu.ru/download/22478362/1rhto0:3hpWaEk1-7u28fKtUhPqWEDdBbY/

Онлайн-документация

Встроенные модули	https://docs.python.org/3/library/index.html
`matplotlib`	https://matplotlib.org/stable/index.html
`numpy`	https://numpy.org/doc/stable/user/index.html
`sympy`	https://docs.sympy.org/latest/index.html
`scipy`	https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide
`pandas`	https://pandas.pydata.org/docs/user_guide/index.html#user-guide
`scikit-learn`	https://scikit-learn.org/stable/user_guide.html
`PyTorch`	https://pytorch.org/docs/stable/index.html
`numba`	https://numba.readthedocs.io/en/stable/index.html
`pyserial`	https://pythonhosted.org/pyserial/
`pyvisa`	https://pyvisa.readthedocs.io/en/1.8/index.html

Приложение А. Основные перегружаемые операторы и методы

Метод	Вызывающий код	Примечание

`__add__ __sub__`
`__mul__ __pow__`
`__truediv__`
`__floordiv__`
`__mod__`
`__matmul__`
`__pos__ __neg__`
	`a + b  a - b`
	`a * b  a ** b`
	`a / b`
	`a // b`
	`a % b`
	`a @ b`
	`+a  -a`
		Обычные алгебраические операции. `__matmul__` для матричного умножения, `__pos__` и `__neg__` для унарных + и –.

`__and__`
`__or__`
`__xor__`
`__invert__`
`__lshift__`
`__rshift__`
	`a & b`
	`a | b`
	`a ^ b`
	`~a`
	`a << b`
	`a >> b`
		Побитовые операции.

`__eq__`
`__ne__`
`__gt__`
`__lt__`
`__ge__`
`__le__`
	`a == b`
	`a != b`
	`a > b`
	`a < b`
	`a >= b`
	`a <= b`
		Сравнения. Если не определен `__gt__`, но определен `__lt__`, то `a > b` заменится при выполнении на `b < a`, и наоборот, и аналогично для других пар операторов.

`__new__`
`__init__`
`__del__`
	`a = ClassName()` вызывает конструкторы объекта `__new__`, после него `__init__`. Деструктор `__del__` вызывается при удалении объекта, т. е. когда счетчик ссылок достигает 0.	

`__radd__`
`__rand__`
`__rmul__`
и т. д.
	`x + a`
	`x & a`
	`x * a`
	и т. д.
		Правосторонние операторы вызываются, если левый операнд не может вызвать свой `__add__`, то есть когда `x.__add__(a)` некорректен.

`__iadd__`
`__imul__`
и т. д.
	`a += b`
	`a *= b`
	и т. д.
		Если инкрементный оператор `__iadd__` не перегружен, вместо него выполняется `a = a + b` и т. п.

`__call__`	`a(*p, **kv)`	Вызов объекта как функции.
`__len__` `__bool__`	`len(a)` `bool(a)` или **`if`** `a:`	Если отсутствует `__bool__`, вместо него вызывается `__len__`.

`__contains__`
`__getitem__`
`__setitem__`
`__delitem__`
	`x` **`in`** `a`
	`a[i]`
	`a[i] = ...`
	`del` `a[i]`
		Проверка наличия элемента в коллекции, обращение к элементу коллекции по индексу.

`__iter__` `__next__`	**`for`** `_` **`in`** `a:` `next(a)`	Итерационные контексты. Если эти методы не перегружены, циклы будут использовать обращения по индексу.

`__repr__`
 	
`__str__`
	`repr(a)`
	`a # в консоли`
	`print(a)`
	`str(a)`
		Строковый вывод «для отладки» и «для пользователя». Если отсутствует `__str__`, используется `__repr__`.

`__enter__` `__exit__`	**`with`** `a:` или **`with`** `a` **`as`** `_:`	Менеджер контекста. `__exit__` срабатывает при выходе из блока.

`__getattribute__`
`__setattr__`
`__delattr__`
	`a.x`
	`a.x = ...`
	`del` `a.x`
		Перегрузка обращения к атрибутам экземпляра. Могут использоваться для защиты атрибутов от изменения.

Приложение Б. Коллекции Python и их атрибуты

	Список `list` Кортеж `tuple`	Множества `set` `frozenset`	Словарь `dict`
Пример коллекции	`[1, 'list', 2]` `(1, 'tuple')`	`{1, 'set', 2}` `frozenset([1])`	`{1: 'dict',` `2: 1024}`
Объекты, входящие в коллекцию	Произвольные	Неизменяемые	Неизм. ключи, произв. значения
Общие операции	Итерация `for x in a:` Размер коллекции `len(a)` Вхождение элемента `x in a` Равенство коллекций `a == b, a != b`		
Сравнения `>  <  >=  <=`	Поэлементное сравнение посл-тей	Подмножества (строгие/нестрогие)	
Операторы	`a + b` `a * n  n * a`	`a & b a | b` `a ^ b a - b` Прим. 1	`a | b` Прим. 2
Обращение к элементам	`a[n]` `a[i:j:k]` Прим. 3		`a[key]` `a.get(key)`

Другие методы	`count`
	`index`
		`isdisjoint`
		`issubset`
		`issuperset`
			`a.keys()`
			`a.values()`
			`a.items()`

Методы изменяемых коллекций:			
	`list:`	`set:`	`dict:`
Общие операции	Поверхностная копия коллекции `a.copy()` Очистить коллекцию `a.clear()`		
Упорядочивание	`sort  reverse`		
Расширение	`extend`	`update` Прим. 4	`update` Прим. 4
Инкрементные операторы	`a += b` `a *= n`	`a&=b  a|=b` `a^=b  a-=b` Прим. 5	`a |= b` Прим. 2
Добавить элемент	`append insert`	`add`	`a[key] = x`
Убрать элемент	`remove` `del a[n]`	`remove` `discard`	`del a[key]`
Извлечь элемент	`a.pop(n)` `a.pop()`	`a.pop()`	`a.pop(key)` `a.popitem()`

В таблице везде считается, что `a` и `b` — коллекции заданного типа, `n` — целое число.

[1] Операции дублированы методами `intersection`, `union`, `symmetric_difference`, `difference`, которые принимают произвольный итерируемый объект.

[2] Начиная с версии Python 3.9.

[3] Для кортежей обращение по индексу и срезы работают только на чтение, для списков — также на изменение и удаление.

[4] Операция `a.update(b)` эквивалентна `a |= b`, и для словарей перепишет значения `a` по ключам, которые были также в `b`.

[5] Операции дублированы методами `intersection_update`, `update`, `symmetric_difference_update`, `difference_update`.

Приложение B. Сравнение скорости поиска по спискам и множествам

Чтобы продемонстрировать, что поиск по множествам происходит существенно быстрее, чем по спискам, рассмотрим следующую задачу. Сгенерируем два набора из `N` случайных чисел от `0` до `Upper`, преобразуем их в множества и посчитаем несколькими методами, сколько чисел `k` входят в оба множества.

В приведенной программе множества названы `S1` и `S2`. Их длина близка к `N`. Для сравнения скорости множества также преобразованы в списки `L1` и `L2`. Мы реализовали следующие способы найти `k`. Самый естественный вариант (i) — воспользоваться встроенным функционалом Python, чтобы построить пересечение множеств `S1 & S2` и узнать количество элементов в нем. Он предсказуемо оказывается и самым быстрым, однако сравнивать его со списками было бы не совсем корректно, поскольку для списков нет похожего готового решения. Чтобы сравнивать скорость поиска в коллекции с помощью оператора `in`, напишем варианты (ii) и (iii) — прямой перебор элементов по обоим множествам. Эти варианты в 2–3 раза медленнее (i), однако между собой они конкурируют в зависимости от реализаций псевдослучайных последовательностей. Списковое включение в варианте (iii) создает такой же объект, что и оператор пересечения множеств, поэтому, формально говоря, вариант (ii) эффективнее по памяти, однако для нашего примера это несущественно. Наконец, повторим те же два алгоритма с использованием списков `L1` и `L2` в вариантах (iv) и (v) и получим замедление в ~10 000 раз. Не составит труда убедиться, что в данном случае важно не то, по какой коллекции идет цикл (`for x in S1` или `for x in L1`), а то, по какой происходит поиск элемента (`x in S2` или `x in L2`). Действительно, поиск по списку занимает линейное по количеству элементов время, тогда как поиск по множеству на самом деле является поиском по отсортированному массиву хеш-значений и выполняется за логарифмическое время.

```
from time   import perf_counter_ns as timer
from random import randrange       as rnd

N = 2 ** 15
Upper = 2 ** 20
S1 = {rnd(Upper) for _ in range(N)}
S2 = {rnd(Upper) for _ in range(N)}
L1 = list(S1)
L2 = list(S2)
print(f'Intersection of {len(S1)} x {len(S2)} sets')

t = timer()
k = len(S1 & S2)                          # (i)
t = timer() - t
t = round(t/1e6, 3)
print(f'& operator:\t\t{k} items;\t{t} ms')
```

```
k = 0
t = timer()
for x in S1:                                    # (ii)
    if x in S2:
        k += 1
t = timer() - t
t = round(t/1e6, 3)
print(f'Set brute force:\t{k} items;\t{t} ms')

t = timer()
k = len({x for x in S1 if x in S2})             # (iii)
t = timer() - t
t = round(t/1e6, 3)
print(f'Set comprehensions:\t{k} items;\t{t} ms')

k = 0
t = timer()
for x in L1:                                    # (iv)
    if x in L2:
        k += 1
t = timer() - t
t = round(t/1e6, 3)
print(f'List brute force:\t{k} items;\t{t} ms')

t = timer()
k = len([x for x in L1 if x in L2])             # (v)
t = timer() - t
t = round(t/1e6, 3)
print(f'List comprehensions:\t{k} items;\t{t} ms')
```

Вывод данной программы, разумеется, будет отличаться от запуска к запуску и зависеть от конкретного компьютера, поэтому к результату следует относиться как к типичному:

```
Intersection of 32292 x 32278 sets
& operator:             971 items;  1.95 ms
Set brute force:        971 items;  3.903 ms
Set comprehensions:     971 items;  3.726 ms
List brute force:       971 items;  22491.631 ms
List comprehensions:    971 items;  22608.704 ms
```

Приложение Г. Аргументы функции

В справочном порядке напишем подробнее, как работает передача аргументов в функцию. Различают позиционные аргументы и аргументы, передаваемые по ключу.

```
>>> def f(a, b, *c, p=-1, q=-2, **s):
...     print(f'a: {a}; b: {b}; c: {c}; '
...         + f'p: {p}; q: {q}; s: {s}')
```

Минимально необходимо передать этой функции два позиционных аргумента `a` и `b`, хотя сделать это можно далеко не одним способом:

```
>>> f(1, 2)
a: 1; b: 2; c: (); p: -1; q: -2; s: {}
>>> f(a=1, b=2)                  # вывод будет
>>> f(b=2, a=1)                  # таким же
```

Попытки вызвать функцию, не передав ей `a` и `b` (например, `f()`, `f(p=-3)`, `f(1)`) приведут к ошибке `TypeError`. Если же переданных аргументов больше двух (их может быть произвольно много), «лишние» неименованные аргументы будут собраны в кортеж `c`:

```
>>> f(1, 2, 3, 4, 5)
a: 1; b: 2; c: (3, 4, 5); p: -1; q: -2; s: {}
```

Обратим внимание, что аргументы по умолчанию `p` и `q` доступны в функции в любом случае. Теперь перейдем к аргументам, передаваемым по ключу:

```
>>> f(1, 2, x=3, y=4, p=6)
a: 1; b: 2; c: (); p: 6; q: -2; s: {'x': 3, 'y': 4}
```

В этом случае словарь `s` соберет пары «ключ—значение» для всех аргументов, кроме тех, что явно обозначены при объявлении функции. При этом если позиционных аргументов `c` нет, можно обозначить именами `a` и `b` и перечислять их в произвольном порядке:

```
>>> f(x=3, y=4, p=6, a=1, b=2)
a: 1; b: 2; c: (); p: 6; q: -2; s: {'x': 3, 'y': 4}
```

Наконец, максимальная форма (примерно такой синтаксис у функции `print`):

```
>>> f(1, 2, 3, 4, 5, p=6, q=7, t=8)
a: 1; b: 2; c: (3, 4, 5); p: 6; q: 7; s: {'t': 8}
```

Если исключить возможность передавать неопределенное число позиционных аргументов, то аргументы, имеющие значения по умолчанию, можно передавать в функцию как позиционные, не обозначая их именами при вызове функции:

```
>>> def g(a, b, p=-1, q=-2, **s):
...     print(f'a: {a}; b: {b}; '
...         + f'p: {p}; q: {q}; s: {s}')
>>> g(1, 2, 3)
a: 1; b: 2; p: 3; q: -2; s: {}
>>> g(1, 2, 3, 4, t=5)
a: 1; b: 2; p: 3; q: 4; s: {'t': 5}
```

Приложение Д. Пример работы с генераторными функциями

Пусть перед нами стоит задача написать несколько генераторов псевдослучайных чисел (ГСЧ), протестировать их и сравнить со стандартным ГСЧ. Постараемся сформировать архитектурное решение, основываясь на следующих предпосылках:

1. Генераторов может быть много, и у них могут быть настраиваемые параметры.
2. Тестов может быть много (хотя мы напишем только один).
3. В каждом тесте ГСЧ может вызываться огромное и заранее неопределенное количество раз, поэтому нежелательно формировать массив случайных чисел заранее.

Мы хотели бы, таким образом, чтобы генератор существовал в виде объекта, который за одно обращение выдает одно значение, а тестировщик существовал в виде функции, которая принимает указанный объект:

```
test(generator)
```

Такое поведение проще всего реализовать с помощью генераторных функций. Напишем две такие функции, линейный конгруэнтный генератор `lcg` и 32-битный `xorshift`. Первый является простейшим ГСЧ и, как можно будет видеть в результате тестирования, «недостаточно случайным» — его периодичность определяется простым числом `modulus`. Второй является предшественником наиболее современных 256-битных ГСЧ `xor-shift-rotate` и его результаты достаточно близки ко встроенным решениям, по крайней мере на нашем несложном тесте.

```
def lcg(start=4967, mult=7043, incr=3581, modulus=6323):
    '''Linear congruent generator'''
    while True:
        yield start / modulus
        start = (start*mult + incr) % modulus

def xorshift(start=7043):
    '''Shift-register generator'''
    up, modulus = (1<<32) - 1, float(1<<32)
    x = start
    while True:
        yield x / modulus
        x ^= (x << 13) & up
        x ^= (x >> 17)
        x ^= (x << 5) & up
```

Генераторные функции, когда мы их используем в качестве ГСЧ, имеют иной синтаксис, чем встроенная `random.random`. Действительно, ГСЧ используют статические переменные, которые в Python разумнее всего эмулировать с помощью классов. Напишем класс `LCG`, эквивалентный функции `lcg`:

```
class LCG:
    def __init__(self, start=4967, mult=7043,
                       incr=3581,  modulus=6323):
        self.x = start
        self.mult = mult
        self.incr = incr
        self.modulus = modulus
```

```
    def __call__(self):
        out = self.x / self.modulus
        self.x=(self.x*self.mult+self.incr)%self.modulus
        return out
```

Наконец, напишем собственно тестировщик, который будет запрашивать N случайных чисел и смотреть, насколько равномерно они распределены по nbox одинаковым интервалам в диапазоне [0, 1). Предусмотрим, что тестировщику может быть передан класс, генераторная функция или стандартная функция.

```
from itertools import islice
from time import perf_counter_ns as timer
from inspect import isgeneratorfunction
from inspect import isclass, isroutine
# эти функции модуля inspect проверяют, является ли
# переданный им объект генераторной функцией, классом
# или функцией (генераторной или обычной), соответственно

def test_uniformness(gen, args=None):
    if args is None: args = {}

    if isclass(gen):
        G = iter(gen(**args), None)
    elif isgeneratorfunction(gen):
        G = gen(**args)
    elif isroutine(gen):
        G = iter(gen, None)
    else:
        raise ValueError('Wrong gen argument')

    nbox, N = 100, 10**5
    box, stat = 1/nbox, [0]*nbox

    t = timer()
    for x in islice(G, N):
        stat[int(x // box)] += 1
    t = timer() - t

    name = gen.__name__ + (str(args) if args else '')
    std = (sum((x-N/nbox)**2 for x in stat)/nbox) ** 0.5
    print(f'{name: <24}'
        + f'{min(stat): >8}'
        + f'{max(stat): >8}'
        + f'{std:12.5}'
        + f'{t/1e6:12.5}')
```

Поскольку архитектура программы на Python интересует нас гораздо больше, чем тестирование ГСЧ, мы написали только один тестировщик, причем довольно примитивный. Серьезная проверка ГСЧ включает множество тестов, например, соответствие вероятности возникновения возрастающих и убывающих последовательностей заданной длины теоретическим значениям.

Теперь, после всей подготовительной работы, рабочий код программы выглядит следующим образом:

```
from random import random
print(f"{'RNG': ^24}{'min': >8}{'max': >8}"
    + f"{'Std. dev.': >12}{'Runtime, ms': >12}")
test_uniformness(LCG)
test_uniformness(lcg)
test_uniformness(lcg, {'start': 0})
test_uniformness(lcg, {'modulus': 100003})
test_uniformness(xorshift)
test_uniformness(random)
```

Результат выполнения этого кода получился у нас следующим:

```
          RNG                  min     max   Std. dev. Runtime, ms
LCG                            982    1019      7.5273      72.758
lcg                            982    1019      7.5273       47.86
lcg{'start': 0}                980    1016      7.0725       47.86
lcg{'modulus': 100003}         770    1344      114.43      48.074
xorshift                       917    1075      34.048      94.067
random                         929    1075      27.899      33.549
```

На полях хотелось бы отметить, что генераторная функция `lcg` работает заметно быстрее, чем класс `LCG`, поскольку она избегает обращения к полям класса и работает исключительно с переменными в своем локальном пространстве имен.

Смысл нашего примера, впрочем, не столько в скорости выполнения, сколько в демонстрации подхода. Если бы задача ограничивалась одним генератором и одним тестом, мы бы написали один цикл на верхнем уровне модуля:

```
x, mult, incr, modulus = 4967, 7043, 3581, 6323
nbox, N = 100, 10**5
box, stat = 1/nbox, [0]*nbox

t = timer()
for _ in range(N):
    stat[int(x / modulus // box)] += 1
    x = (x*mult + incr) % modulus
t = timer() - t

std = (sum( (x-N/nbox)**2 for x in stat )/nbox) ** 0.5
print(f'{min(stat): >8}' + f'{max(stat): >8}'
    + f'{std:12.5}' + f'{t/1e6:12.5}')
```

Но масштабирование и расширение задачи без разбиения ее на составные части привело бы к чрезмерному дублированию кода. Поэтому «генеративная» часть цикла заключена в отдельные генераторные функции (или классы), а статистическая обработка — в независимую функцию, которая обращается к произвольному генератору.

Приложение Е. Скорость работы циклов, списковых включений и numpy

Чтобы дать представление о преимуществе в скорости списковых включений по сравнению с поэлементными операциями, реализуем несколькими методами поэлементное сложение массивов. Чтобы время выполнения программы было заметным (секунды), на тестовом компьютере мы подобрали длину массивов $N = 2^{23}$. Такой код потребует около 1–1.5 ГБ свободной памяти (увы, ненулевое целое число в Python занимает минимум 28 байт). Разумеется, для таких задач более естественно использовать `numpy`, поэтому рассмотрим также его производительность.

```
from time   import perf_counter_ns as timer
from random import randrange       as rnd
import numpy as np

N = 2 ** 23
Upper = 2 ** 30
L1 = [rnd(Upper) for _ in range(N)]
L2 = [rnd(Upper) for _ in range(N)]
print(f'Addition of {N} long lists')

M = [0]*N
t = timer()
for i in range(len(L1)):
    M[i] = L1[i] + L2[i]                    # (i)
t = round((timer() - t)/1e6, 3)
print(f'Using indices:\t\t{t} ms')

M = []
t = timer()
for x, y in zip(L1, L2):
    M.append(x+y)                           # (ii)
t = round((timer() - t)/1e6, 3)
print(f'Append:\t\t\t{t} ms')

t = timer()
M = [x + y for x, y in zip(L1,L2)]          # (iii)
t = round((timer() - t)/1e6, 3)
print(f'List comprehensions:\t{t} ms')

L1 = np.array(L1, dtype=object)
L2 = np.array(L2, dtype=object)
t = timer()
M = L1 + L2                                 # (iv)
t = round((timer() - t)/1e6, 3)
print(f'Numpy (object) add:\t{t} ms')
```

```
M = L1.copy()
t = timer()
M += L2                                    # (v)
t = round((timer() - t)/1e6, 3)
print(f'Numpy (object) iadd:\t{t} ms')

L1 = np.array(L1, dtype='int32')
L2 = np.array(L2, dtype='int32')
t = timer()
M = L1 + L2                                # (vi)
t = round((timer() - t)/1e6, 3)
print(f'Numpy (int32) add:\t{t} ms')

t = timer()
L1 += L2                                   # (vii)
t = round((timer() - t)/1e6, 3)
print(f'Numpy (int32) iadd:\t{t} ms')
```

Типичные времена получаются примерно такими:

```
Addition of 8388608 long lists
Using indices:              1567.386 ms
Append:                     1548.222 ms
List comprehensions:        984.727 ms
Numpy (object) add:         443.278 ms
Numpy (object) iadd:        297.573 ms
Numpy (int32) add:          146.585 ms
Numpy (int32) iadd:         10.861 ms
```

Итак, вариант (i) — заранее создается массив нужного размера, сумма вычисляется поэлементно. Вариант (ii) — используется метод `append`, то есть массив с суммой постоянно меняет размер. От случая к случаю варианты (i) и (ii) конкурируют друг с другом по скорости. Вариант (iii) — используется списковое включение, что уже дает преимущество в скорости в ~1.5 раза, а также выглядит куда лаконичнее.

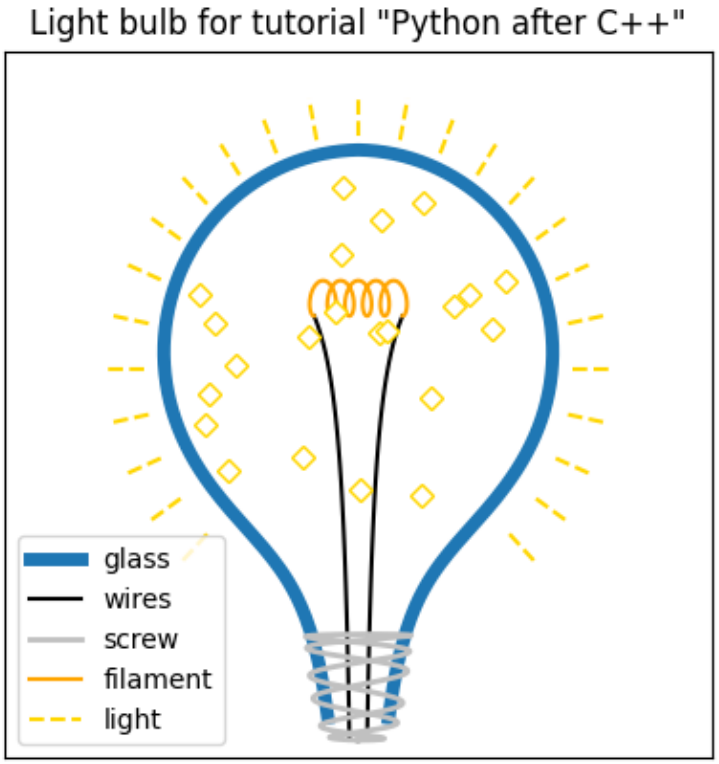
Для полноты картины мы также реализовали решение с использованием `numpy`. Вариант (iv) — превращаем список в `ndarray`, но оставляем типом данных целые числа Python. То есть организацию массивов и все внутренние циклы берет на себя `numpy`, однако сложение, как и прежде, выполняет Python. Уже это ускоряет сложение еще в два раза. Если же теперь применить инкрементное сложение (v), то есть устранить необходимость выделять память под новый массив, время выполнения падает еще в полтора раза. На этом этапе почти все время уходит именно на сложение `int + int`. Наконец, перейдем к тому, как на самом деле должна решаться эта задача — в варианте (vi) мы складываем два массива 32-битных целых чисел. Здесь `numpy` уже берет на себя всю работу. А если нам по каким-либо причинам не нужен массив `L1` после сложения, мы применим инкрементное сложение (vii), для которого даже не требуется выделять новую память, и сократим время выполнения в ~100 раз по сравнению с реализациями на чистом Python.

Приложение Ж. Примеры работы с `matplotlib`

В этом приложении приведем два более изощренных, чем в основном тексте, примера работы с `matplotlib`, но не будем их подробно комментировать. Понадеемся, что поочередно исключая те или иные управляющие параметры и проверяя результат, не так сложно разобраться, за что они отвечают.

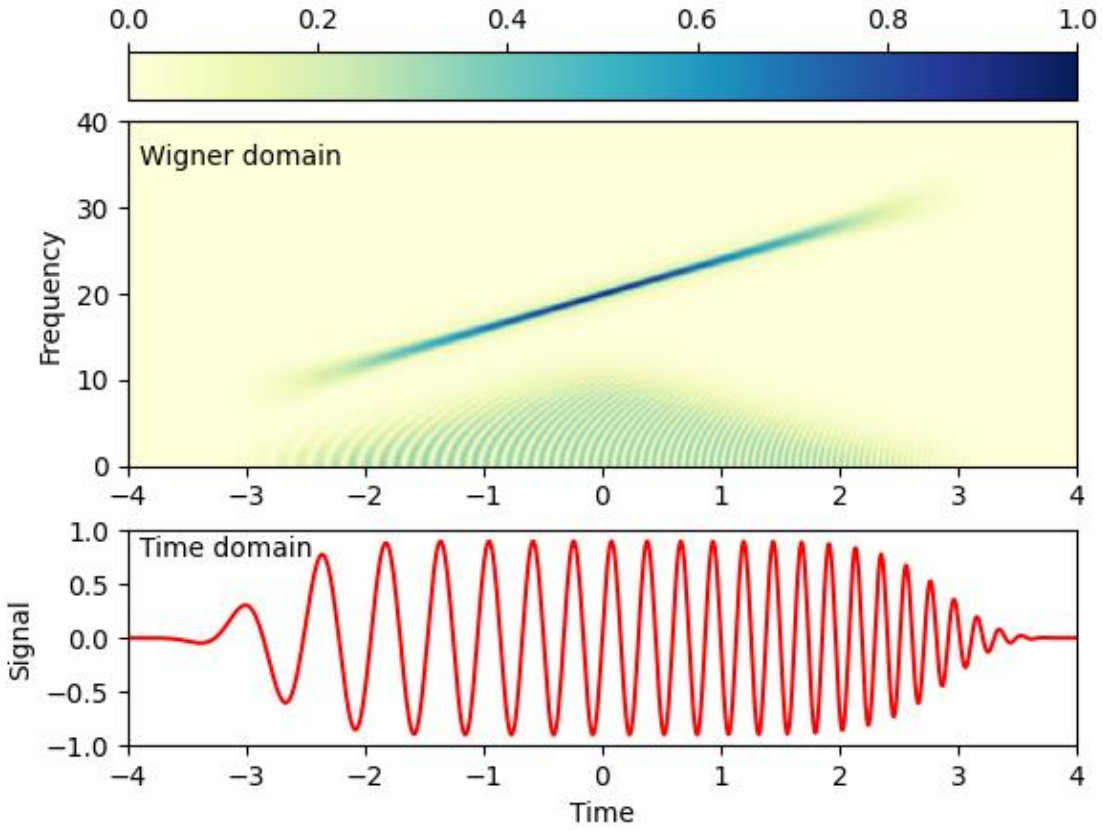
```
import matplotlib.pyplot as plt
from numpy import linspace, array, sin, cos, pi, cbrt
from numpy.random import default_rng

nphi = 256
phi = linspace(-pi, pi, nphi+1)

plt.plot(1.1*sin(0.95*phi),
        cos(phi) - 0.2/(1.05*pi-abs(phi)),
        'C0', lw=5)             #lw = linewidth

xx = linspace(0.05, 0.25, 50)
yy = 0.6 - 0.15/xx

plt.plot(-xx, yy, 'k', label='_nolegend_')
plt.plot(xx, yy, 'k')               # 'k' = 'black'

plt.plot(0.3*sin(6*phi+pi/12)*cbrt(cos(phi/2)),
        -2.1 + 0.3*cos(phi), 'silver', lw=2)

plt.plot(xx[-1]/pi * (phi+0.6*sin(5*phi)),
         yy[-1] + 0.1*(1+cos(5*phi)),
         'orange')

nb = 32
ray = array([1.1, 1.3])
for ib in range(nb//8, nb-nb//8+1):
    tht = 2*pi/nb * ib - pi/2
    plt.plot(ray*cos(tht)*1.1,
             (ray*sin(tht)-0.25)*1.2,
             ls='--', color='gold')        #ls = linestyle

rng = default_rng()
nr = 30
Rx = 2*rng.random(size=nr) - 1
Ry = 2*rng.random(size=nr) - 1
Rin = (Rx**2 + Ry**2) < 1
Rx, Ry = Rx[Rin], Ry[Rin]

plt.plot(Rx, Ry-0.25, ls='none', marker='D',
        mec='gold',     #mec = markeredgecolor
        mfc='white')    #mfc = markerfacecolor
```

```
plt.xlim(-2, 2)
plt.ylim(-2.5, 1.5)
plt.legend(['glass', 'wires', 'screw',
            'filament', 'light'])
plt.title('Light bulb for tutorial "Python after C++"')

ax = plt.gca()
ax.set_aspect(1)
ax.set_xticks([])
ax.set_yticks([])
plt.show()
```




```
import matplotlib.pyplot as plt
from numpy import pi, sin, exp, arange, vstack
from numpy.fft import fft

dt = 0.01
nt = 2**13
t = dt * arange(-nt//2, nt//2)
ns = nt//8
dw = pi / (ns*dt)
w = dw * arange(ns//2)

E = 0.9*exp(-(t/3)**8)*sin(20*t+2*t**2)

it1 = nt//2 - int(4//dt) - 1
it2 = nt//2 + int(4//dt) + 2
uw = int(40//dw)+1
```

```
WL = vstack([E[it-ns//2:it+ns//2]
             for it in range(it1, it2)])
WL *= WL[:, ::-1].conj()
WL = fft(WL)[:, :uw]
WL = abs(WL)
WL /= WL.max()

fig, (ax1, ax2) = plt.subplots(2, 1, height_ratios=[2,1])

p1 = ax1.pcolormesh(t[it1:it2], w[:uw], WL.T,
                    cmap='YlGnBu', shading='gouraud')
ax1.set_xlim(-4, 4)
ax1.set_ylim(0, 40)
ax1.set_ylabel('Frequency')
plt.colorbar(p1, ax=ax1, location='top')
ax1.text(-3.9, 35, 'Wigner domain')

ax2.plot(t, E, 'r')
ax2.set_xlim(-4, 4)
ax2.set_ylim(-1, 1)
ax2.set_xlabel('Time')
ax2.set_ylabel('Signal')
ax2.text(-3.9, 0.75, 'Time domain')

plt.show()
```



Приложение 3. Функции numpy

В этом приложении постараемся дать обзор функций numpy. Стараясь быть краткими, во-первых, перечислим небольшую часть функций и, во-вторых, опустим описание их аргументов и других особенностей, чтобы не лишать читателя необходимости ознакомиться с документацией (numpy.info(function) или онлайн). В numpy очень часто дублированы функции и методы массивов. Случаи, когда метод не имеет аналогичной функции (или она названа по-другому), будем обозначать явно, записывая его как атрибут массива x.

Сведения о массиве. Данные атрибуты являются полями.

x.ndim	ранг массива (длина кортежа shape)
x.dtype	тип элемента массива
x.itemsize	байт на элемент
x.size	полное количество элементов (произведение чисел shape)
x.nbytes	байт на весь массив (size * itemsize)
x.shape	кортеж — размерность массива
x.strides	кортеж — смещение между элементами по каждой оси в байтах
x.flags	прочие служебные атрибуты

Создание массива.

empty zeros ones full	массив заданного размера, неинициализованный или заполненный нулями, единицами, заданными значениями
eye diag	единичная матрица или матрица с заданной главной диагональю
array copy fromfile load loadtxt	массив на основе другого объекта, другого массива, файла, текстового файла
arange linspace logspace geomspace	массив, заполненный арифметической или геометрической прогрессией
concatenate stack block vstack hstack dstack column_stack row_stack	составить массив из нескольких массивов

Для записи массивов в файл используются x.tofile, save, savetxt.

Преобразование значений массива. Все эти функции, кроме real и imag, возвращают новый массив.

x.astype	преобразование типа
real imag conj conjugate absolute angle	комплексная арифметика angle — фаза
x.round around ceil fix floor rint trunc	округление
clip	приведение значений к указанному диапазону

Преобразование размерности массива. Как правило, эти функции не копируют данные, а только возвращают новый заголовок.

`x.T transpose`	транспонирует массив; `x.T` является полем
`swapaxes`	меняет местами оси массива (настраиваемое транспонирование)
`diagonal`	возвращает диагональ массива
`reshape`	массив с заданными размерностями, использующий те же данные
`x.flat`	1-мерный итератор по элементам массива
`squeeze`	убирает оси длины 1 из массива
`flip`	инвертирует заданные оси, для 1-мерного эквивалентен `x[::-1]`
`rot90`	поворот матрицы на 90°
`x.flatten`	1-мерная копия массива
`ravel`	1-мерный связный массив с теми же данными; копия делается только при необходимости
`x.base`	поле имеет значение `None`, если установлен флаг `OWNDATA`, в ином случае это массив, данные которого использует текущий заголовок

Анализ массива. Эти функции могут пробегать как по всему массиву, так и разбивать массив на срезы вдоль определенной оси (параметр `axis`), обрабатывать их независимо и собирать результаты в массив (обычно рангом на 1 меньше исходного). Некоторые имеют `nan`-версии, которые игнорируют элементы массива, равные `nan`.

`all any`	используются вместо встроенных `all` и `any`

`amax x.max nanmax`
`amin x.min nanmin`
`ptp`
	максимальное и
	минимальное значение массива,
	их разность

`argmax argmin`	индекс максимума и минимума, имеют `nan`-версию

`mean average`
`median`
`var std`
	среднее значение, среднее значение с весами, медиана, дисперсия и стандартное отклонение; кроме `average` имеют `nan`-версию

`sum prod`	сумма и произведение элементов, имеют `nan`-версию
`cumsum` `cumprod`	кумулятивные («нарастающим итогом») сумма и произведение, имеют `nan`-версию
`diff`	массив разностей соседних элементов исходного массива

`sort`
`x.sort`
`argsort`
	сортировка массива (функция создает копию, метод сортирует массив на месте);
	перестановка индексов, соответствующая сортировке

`nonzero`
`flatnonzero`
`argwhere`
	массив индексов, по которым в исходном массиве расположены ненулевые элементы (не имеют параметра `axis`; размер результата неизвестен заранее)

Алгебраические операции, в том числе сравнения, реализованы и как функции, и как перегруженные операторы. Как правило, используется операторная запись, кроме равенства.

`isclose(x, y)`	поэлементное равенство в пределах погрешности
`allclose(x, y)`	`True`, если массивы поэлементно равны в пределах погрешности; то же самое, что `isclose(x, y).all()`
`isfinite isinf isnan`	проверка `float`-массивов на исключения `inf` и `nan`

Математические функции применяются к массивам поэлементно. Функции двух аргументов укажем явно.

```
sign sqrt cbrt square hypot(x, y)
exp log log10 log2
sin cos tan arcsin arccos arctan arctan2(y, x)
sinh cosh tanh arcsinh arccosh arctanh
```

Суб-модуль `emath` содержит дубликаты функций, для которых бывает нужен комплексный результат при вещественном аргументе (`sqrt`, `log` и т. д.), чтобы автоматически обрабатывать такие случаи. Суб-модуль `fft` содержит функции, выполняющие быстрое преобразование Фурье.

Линейная алгебра.

`dot vdot einsum`	функции дополняют и обобщают матричное произведение `@` на случай многомерных массивов (`x.ndim` > 2)
`linalg.eig linalg.eigh`	собственные значения и вектора матрицы (`h`-версия для симметричной или эрмитовой матрицы)
`linalg.eigvals linalg.eigvalsh`	собственные значения матрицы; собственные вектора не вычисляются ради быстродействия
`linalg.det`	определитель матрицы
`linalg.inv`	обратная матрица
`trace`	след матрицы (сумма диагональных элементов)
`linalg.lstsq`	решение системы линейных уравнений методом наименьших квадратов

Псевдослучайные числа. `rng = np.random.default_rng()` инициализирует генератор псевдослучайных чисел (можно инициировать несколько независимых генераторов, а также контролировать стартовое значение генератора).

`rng.integers`	случайные целые числа
`rng.random`	равномерное распределение в диапазоне [0.0, 1.0)
`rng.normal`	нормально распределенные вещественные числа
`rng.choice(x)`	выбор случайного элемента массива
`rng.shuffle(x)` `rng.permuted(x)` `rng.permutation(x)`	случайные перестановки массива

Пользовательские универсальные функции. Класс `vectorize` и функция `frompyfunc` позволяют преобразовать скалярные функции (пользовательские или сторонние по отношению к `numpy`) в универсальные, то есть поддерживающие поэлементное применение, транслирование и т. д. по правилам `numpy`.

Приложение И. Декораторы

Декораторы Python не слишком важны для написания скриптов, но часто встречаются в больших программных продуктах, поэтому мы решили не включать их в основное содержание и описать только в приложении.

Декораторы модифицируют поведение функций. Пусть, для примера, мы хотим отслеживать вызовы некоторой функции `myfunc`:

```
def myfunc(x, y):
    return x * (y-1)
```

Разумеется, можно дописать вывод в тело функции. Чтобы этого не делать (а если функция предоставлена сторонним модулем, это и не удастся), можно написать обертку:

```
def myfunclog(x, y):
    print(f'Called myfunc({x}, {y}')
    return myfunc(x, y)
```

и вызывать ее всякий раз вместо `myfunc`. Чтобы не повторять эту процедуру для каждой функции, мы можем написать декоратор, функцию `logger`, которая принимает в качестве аргумента какую-либо функцию `func`, на ее основе формирует новую функцию `funclog`, которую возвращает в качестве значения:

```
def logger(func):
    def funclog(*pargs, **kvargs):
        print(f'Called {func.__name__}{pargs}{kvargs}')
        return func(*pargs, **kvargs)
    funclog.__name__ = func.__name__
    return funclog
myfunclog = logger(myfunc)
```

В отличие от предыдущей реализации, декоратор позволяет экранировать имена и назвать декорированную функцию так же, как исходную:

```
myfunc = logger(myfunc)
```

Особенности организации пространства имен в этом случае (так называемое замыкание пространства имен) таковы, что объект «старой» `myfunc` доступен по имени `func`, которое не является глобальным, а «замкнуто» в локальной области видимости функции `logger`.

Краткая (наиболее часто встречающаяся) запись применения декоратора такова:

```
@logger
def square(x):
    return x**2
```

Она эквивалентна объявлению функции `square` без декоратора и последующему присваиванию `square = logger(square)`.

У функции может быть более одного декоратора:

```
from time import perf_counter_ns
def timer(func):
    def f(*pargs, **kvargs):
        t = perf_counter_ns()
        res = func(*pargs, **kvargs)
        t = perf_counter_ns() - t
        return res, f'{t} ns'
    f.__name__ = func.__name__
    return f
```

```
@logger
@timer
def cube(x):
    return x**3
```

Однако следует иметь в виду, что порядок следования декораторов важен: первый является внешним. Увидеть в действии все приведенные выше функции можно, вызвав, например,

```
>>> print(square(8), cube(10), myfunc(5, 5))
Called square(8,){}
Called cube(10,){}
Called myfunc(5, 5){}
64 (1000, '2226 ns') 20
```

(Как мы видим, декоратор `timer` поменял возвращаемое значение функции `cube`: теперь это кортеж из числа и строки. А поменяв местами декораторы `timer` и `logger`, можно убедиться, что их порядок существенен.)

Помимо декораторов функций существуют декораторы классов и методов классов. Декораторы могут иметь аргументы, которые указываются в скобках: `@decorator(args)`. Из стандартных декораторов отметим декораторы кеширования функций, собранные в модуле `functools`, декораторы методов классов `staticmethod` и `classmethod`, декораторы классов модуля `dataclasses`, однако подробное их описание оставим документации.

Учебное пособие

ШИПИЛО Даниил Евгеньевич
КОНОВКО Андрей Андреевич
ЛУКАШЁВ Алексей Алексеевич
ПАНОВ Николай Андреевич

ЯЗЫК ПРОГРАММИРОВАНИЯ PYTHON.
СЕМЕСТР 3 — ПОСЛЕ C++

Подписано в печать 20.03.2024 г.
Формат А5. Объем 6,25 п.л. Тираж 100 экз.
Заказ №24

Физический факультет МГУ им. М.В. Ломоносова
119991, Москва, ГСП-1, Ленинские горы, д. 1, стр. 2

Отпечатано в отделе оперативной печати
физического факультета