Python

вводный курс (для любопытных)

Python один из старых языков программирования. За время своего существования язык постоянно развивался и становился всё более совершенным с каждой новой версией. Благодаря этому Python до сих пор занимает одно из первых мест среди наиболее популярных языков. В данном руководстве мы будем использовать версию *python3.12*. Надо иметь ввиду, что более ранние версии несовместимы с этой по многим аспектам. К сожалению, язык интерпретируемый и нет возможности создавать рабочие *exe*-файлы. Но хорошо то, что интерпретатор достаточно быстрый и работать с ним — одно удовольствие. Python сравнительно прост и лучше других языков подходит для новичков, только начинающих обучаться программированию (впрочем, это утверждение не бесспорно). Это моё руководство ориентировано на читателей, имеющих опыт программирования на современных языках и знающих, например, основы объектно-ориентированного и функционального стилей программирования. Как обычно, я буду придерживаться своего подхода к изложению материала и не буду тратить время на такую ерунду, как комментарии, подбор подходящих идентификаторов, правила форматирования и тому подобное. Полагаю, что всякий, кто хоть немного знаком с программированием, знает, что идентификаторы можно выбирать любые, кому что нравится, и незачем тут устанавливать какие-то правила. Поскольку Pyhton не ограничивает применение кириллицы, идентификаторы можно писать даже и на русском (а также и на других языках).

Python имеет интерактивный режим (*Repl*), который можно вызвать командной: **python**. На консоли появится сообщение о версии языка и приглашение в виде знака >>. После этого *Repl* можно использовать точно также, как это делается для большинства современных языков. Кроме *Repl* Python имеет свой встроенный редактор под названием *IDLE*. Редактор имеет своё меню и позволяет писать, редактировать и запускать программы. Кроме того в самом редакторе можно использовать тот же *Repl*. Освоить как *Repl*, так и *IDLE* очень просто и здесь я не буду на этом останавливаться. Далее я буду пользоваться исключительно только командной строкой, а редактор для написания программ читатель может выбрать любой, какой ему больше всего нравится (рекомендую не выбирать слишком

«крутые» редакторы вроде Visual Studio на которых вы потратите впустую уйму времени). Я предпочитаю использовать легковесный редактор *Geany*, который позволяет не только писать и редактировать, но также и запускать программы.

Базовые понятия

Рассмотрим пока простейшие конструкции языка с тем, чтобы можно было писать и выполнять простые программы, а затем переходить к более сложным задачам.

Getting started

Не изменяя традиции начнём с программы приветствия, которая на Python предельно простая:

print ("Привет, Мир!") → *Привет, Мир!*

Здесь и всюду далее стрелкой → я буду заменять слова: получим, будет выдано на консоли и так далее. При копировании программ для выполнения на своей машине эти стрелки и последующий текст надо удалять.

Текст программы надо написать на редакторе и поместить в файл, название которого имеет расширение **.py**. Пусть это будет файл **prog.py**. Дальше я программы всех примеров буду называть этим именем и не накапливать файлы на диске. Советую также поступать и читателям, поскольку небольшие программы примеров для выполнения проще скопировать из этого руководства, а не накапливать на диске кучу мусора и путаться с названиями файлов. Файл **prog.py** надо расположить там где удобно, в командной строке перейти в эту папку и выполнить команду:

python prog.py

Немедленно получим результат.

Наша программа содержит всего лишь один оператор **print**, который выводит на консоль указанный в скобках текст. Значит в программах на Python не требуется функция **main**, определяющая точку входа. Отметим ещё, что строки кода не требуется, хотя и допустимо, заканчивать точкой с запятой. Если на одной строке располагается несколько выражений (инструкций), тогда их надо разделять точкой с запятой.

Сразу отмечу, что строки комментариев на Python начинаются со знака #. Здесь я никогда не буду применять комментарии, так как

считаю, что в коротких примерах они лишние. Могут применяться комментарии документации, но о них поговорим позднее.

Как и любой другой язык, Python использует ключевые (зарезервированные) слова, которых, кстати, сравнительно немного. Список этих слов можно получить, выполнив такую программу:

```
import keyword
print(keyword.kwlist)
```

Этот текст также надо написать на редакторе (как я уже сказал, в файле **prog.py**, просто изменив текст предыдущего примера) и выполнить в командной строке. Получим такой результат:

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

Если читатель слабо знает английский, будет полезно эти слова перевести и по-возможности запомнить, все они нам потребуются. В конце изучения курса можно вернуться к этому списку. Если окажется, что какие-то из этих слов вам ни о чём не говорят, это будет означать, что что-то вы пропустили или нетвёрдо усвоили. Список этот полезен ещё и тем, что все эти слова нельзя использовать в качестве идентификаторов для переменных, функций, классов и так далее.

Немного усложним программу приветствия:

```
x = input("Как вас зовут? ")
print("Привет, {0}!".format(x))
```

Теперь возможен после запуска программы, например, такой диалог:

Как вас зовут? Борис Уваров
Привет, Борис Уваров!

Оператор **input** позволяет вводить текст с клавиатуры. При этом предварительно он выводит тот текст, который оператору передается в скобках и далее программа будет ждать, пока мы не напечатаем что-то на клавиатуре, закончив ввод клавишей *enter*. При этом введённый текст станет значением переменной **x**. Идентификаторы переменных на Python должны начинаться с буквы (латиница, кириллица, греческий алфавит и т. д. и в любом регистре) или с нижней черты (_). Функция **format** обеспечивает форматированный вывод. Технику вывода на консоль будем осваивать по ходу дела. В данном примере оператор **print** выведет заданный текст в кавычках, поставив на месте нуля в фигурных скобках значение переменной **x**.

Обычно такую подстановку данных называют *интерполяцией*, позже рассмотрим более подробно. Python позволяет вместо двойных кавычек использовать одинарные.

Выводимый оператором **input** текст можно сохранять в переменной:

```
x = 'Как Вас зовут? '
y = input(x)
print('Здравствуйте, {0}'.format(y)) →
```

Как Вас зовут? Борис
Здравствуйте, Борис

Полученное с помощью оператора **input** значение всегда будет текстом (имеет тип **str**). Если нам надо ввести с клавиатуры например целое число, то мы должны полученное текстовое значение конвертировать к типу **int** с помощью стандартной функции **int()**:

```
x = input('Введите число: ')
y = int(x)
print(y)
```

Получим такой диалог:

Введите число: 123
123

О конвертировании типов смотрите далее.

Базовые (встроенные) типы

Python является языком с динамической системой типов и контролирует типы на этапе выполнения программы (runtime), а не при компиляции. Поэтому тип переменных при их инициализации можно не указывать, хотя это и допустимо. Система базовых типов на Python довольно простая.

Числа (*Numbers*)

Числа представлены всего тремя типами:

Целые числа не ограниченные по величине имеют название типа **int** (в других языках такой тип обычно называется *BigInt* или *Long)*. Переменная на Python может быть объявлена одновременно с инициализацией, например:

```
x = 12345
print(x) → 12345
```

print (type(x)) → *<class 'int'>*

Переменная **x** автоматически получит тип **int**. Функция **type** позволяет узнать тип переменной (на слово **class** пока не обращайте внимания). При объявлении переменной тип можно указать и явно через двоеточие после идентификатора:

x: int = 12345
print(x) → *12345*

Но практически в данном случае это ничего не даёт, поскольку дальше мы можем этой переменной присвоить значение другого типа, например:

x = "Hello"
print (type(x)) → *<class 'str'>*

И после этого **x** получит тип **str**.

Числа с плавающей запятой (это название по традиции, на самом деле употребляется точка) имеют тип **float**:

x = 27.975
print(x) → *27.975*
print (type(x)) → *<class 'float'>*

Можно применять «научную нотацию»:

y: float = 0.27975e2
print(y) → *27.975*

Комплексные числа имеют тип **complex** при таком синтаксисе:

x = 0.5 + 7j
print(x) → *(0.5+7j)*
print(x.real) → *0.5*
print(x.imag) → *7.0*
print (type(x)) → *<class 'complex'>*

Функция (метод) **real** выделяет действительную часть, а **imag** – мнимую. Обратите внимание: мнимую часть мы написали, как целое число **7**, а метод **imag** вернул его как число с плавающей запятой.

Математические функции для комплексных чисел находятся в модуле **cmath**, который надо предварительно импортировать (подробно о модулях поговорим позднее):

import cmath
x: complex = cmath.sqrt(-5.0)
print(x) → *2.23606797749979j*

По количеству знаков после десятичной точки можно судить о точности вычислений. Тип **float** на Python соответствует типу **double** на других языках (числа двойной точности).

Имеется обычный набор математических операторов: **+, -, *,** /. Оператор деления **%** возвращает остаток результата:

print(7 % 3) → *1*

print(7.8 % 3) → *1.8*

Оператор деления // возвращает целую часть результата:

print(7 // 3) → *2*

print(7.8 // 3) → *2.0*

Двумя знаками ****** обозначается операция возведения в степень:

print(2 ** 3) → *8*

print(1.7 ** 2.5) → *3.7680989902071307*

Python сам определяет тип числа из контекста и нет необходимости ставить десятичную точку при целых числах:

print(7 / 3) → *2.3333333333333335*

print(9 / 3) → *3.0*

Значит, при делении тип чисел всегда **float**.

Для арифметических операций допустима краткая форма:

```
x = 3
x *= 3
print(x) → 9
```

Здесь **x *= 3** то же самое, что и **x = x * 3**.

Математические функции находится в модуле **math**:

```
import math
x = math.sin(1.5)
print(x) → 0.9974949866040544
```

При этом надо ещё и указывать название модуля при вызове функций (**math.sin(1.5)**), что вообще-то выглядит довольно глупо. От этого можно избавиться, применив оператор **from** (русское «из»):

```
from math import sin
x = sin(1.5)
print(x)→ 0.9974949866040544
```

В модуле **random** имеется одноимённая функция, которая возвращает случайное число типа **float** в диапазоне **0..1**.

```
import random
x = random.random()
print(x) → 0.5825471002378675
```

При повторных вызовах будем получать всё новые случайные числа.

Кроме функции **random** есть ещё функция **randint**, позволяющая получать случайные целые числа из заданного диапазона:

```
import random
```

x = random.randint(1, 10)
print(x) → *7*
y = random.randint(1, 100)
print(y) → *83*

Функция **choice** из этого же модуля позволяет выбирать случайным образом элементы в коллекциях, или буквы в текстах:

import random
x = random.choice([1, 2, 3, 4])
y = random.choice('Тучи над городом встали')
print(x) → *3*
print(y) → *м*

Функция **floor** из модуля **math** возвращает целую часть числа изменяя его всегда в сторону уменьшения:

import math
x = math.floor(2.5)
print(x) → *2*
y = math.floor(-2.5)
print(y) → *-3*

Функция **trunc** делает то же самое, но изменяет число всегда по направлению к нулю:

z = math.trunc(2.5)
print(z) → *2*
w = math.trunc(-2.5)
print(w) → *-2*

Функция **round** позволяет округлять числа:

x, y, z = round(2.567), round(2.467), round(2.567, 2)
print(x, y, z) → *3 2 2.57*

Второй (необязательный) аргумент функции **round** задаёт число знаков после запятой. Обратите внимание на применённый здесь синтаксис групповой инициализации.

В модуле **fractions** есть функция **Fraction**, которая позволяет работать с рациональными дробями. Например:

from fractions import Fraction
x = Fraction(3, 7)
print(x) →*3/7*
y = Fraction(9, 6)
print(y) →*3/2*
print(x/y) →*2/7*
print(x2)** → *9/49*
print(type(x)) → *<class 'fractions.Fraction'>*

Как видим, если есть возможность, дробь сокращается при её создании, и над дробями можно выполнять арифметические операции. При использовании в математических функциях дроби трансформируются к типу **float**:

import math
from fractions import Fraction
x = Fraction(3, 7)
print(math.sin(x)) → *0.415571854993052*

Строки (*Strings*)

Строки это набор символов в двойных или одинарных кавычках. Тип строковых переменных называется **str**:

x = 'Маша'
print(type(x)) → *<class 'str'>*

Отдельные символы в кавычках имеют тип **chr**:

x: chr = 's'
print(type(x)) → *<class 'str'>*

Booleans

Логические (булевы) переменные могут иметь всего два значения: **True** и **False**, их тип называется **bool**:

x = True
print(type(x)) → *<class 'bool'>*

Переменные этого типа могут участвовать в логических операциях **and**, **or** или **not**:

x = True; y = False
print(x or y) → *True*
print(x and y) → *False*
print(not x) → *False*

Встроенная функция **isinstance**, возвращающая логическое значение, также позволяет контролировать тип:

print(isinstance(2.08, float)) → *True*
print(isinstance('Катя', float)) → *False*

В логических выражениях значения **True** и **False** можно заменять цифрами **1** и **0**:

print(1 and 0) → *0*
print(not 1) → *False*

Результат может быть как в форме слов, так и цифр.

Логические значения возвращаются операторами сравнения: **==, >, >=, <, <=, !=**:

print(7 > 2) → *True*

print('Люда' == 'Петя') → *False*

Переменные могут принимать специальное значение **None**, которое означает отсутствие значения.

x = print(7) → *7*

print(x) → *None*

Оператор **print** только выводит результат на консоль и ничего не возвращает, иначе говоря, он возвращает **None**.

Проверка переменной на **None** может выполняться с помощью операции сравнения на равенство (==) или с применением оператора **is**:

```
x = None
print(x is None) → True
print(x == None) → True
```

Дальше мы увидим, что значение **None** полезно во многих случаях.

Преобразование типов

Конкретный тип можно преобразовать (конвертировать) в другой тип. Функции для конвертирования типов имеют те же названия, что и названия типов. Например, функция для преобразования в тип **int** называется **int**(), в тип **float** – называется **floa**t() и так далее.

```
x = '42.78'
y = float(x)
print(y) → 42.78
z = int(y)
print(z) → 42
```

Здесь сначала текстовая переменная преобразуется к типу **float**, а затем переменная типа **float** преобразуется к типу **int**. При преобразовании типа **float** к типу **int** число не округляется.

Есть подобные функции и для преобразования одних видов коллекций в другие:

```
a = 'hello'
print(list(a)) → ['h', 'e', 'l', 'l', 'o']
print(set(a)) → {'o', 'e', 'l', 'h'}
print(tuple(a)) → ('h', 'e', 'l', 'l', 'o')
```

Здесь текстовая переменная преобразуется в список, в множество и в кортеж (смотрите далее).

Коллекции

Списки (List)

В Python различаются упорядоченные и неупорядоченные коллекции. Упорядоченными коллекциями являются списки, которые представляют элементы любых типов в квадратных скобках:

s = [1.3, 7, True, 'Лариса']
print(s) → *[1.3, 7, True, 'Лариса']*

Если нужен пустой список, применяются пустые скобки:

s = []

Посмотрим, какой тип имеют списки:

s = [1.3, 7, True, 'Лариса']
print(type(s)) → *<class 'list'>*

Значит списки имеют тип **list**.

Элементы списков можно извлекать по индексам (начинаются с нуля):

s = [1.3, 7, True, 'Лариса']
print(s[3]) → *Лариса*

Допустимы отрицательные индексы:

print(s[-2]) → *True*

Извлекать элементы списка можно и так:

m = [3, 8, 4, 5]
a, b, c, d = m
print(c) → *4*

При этом количество элементов в списке и количество переменных в левой части от знака равенства должны быть одинаковыми. Но если последнюю переменную снабдить звёздочкой впереди, то оставшиеся элементы будут объединены в список:

m = [3, 8, 4, 5]
a, *b = m
print(a, b) → *3 [8, 4, 5]*

Звёздочку можно использовать и так:

m = [3, 8, 4, 5,9]
a, *b, c = m
print(a, b, c) → *3 [8, 4, 5] 9*

Списки позволяют получать срезы (часть списка):

s = [1, 2, 3, 4, 5, 6, 7, 8, 9]
s1 = s[4:7]
print(s1) → *[5, 6, 7]*

Списки можно изменять (способность изменяться называется *mutable*):

x = [1.3, 7, True, 'Лариса']
x[2] = 77
print(x) → *[1.3, 7, 77, 'Лариса']*

Функции **append**, **insert**, **remove**, **index**, **len** и **reverse** позволяют работать с элементами списков. О назначении этих функций можно судить по их названиям. Достаточно показать их действие на примерах без лишних комментариев:

s = [1.3, 7, True, 'Лариса']
s.append('Маша')
print(s) →*[1.3, 7, True, 'Лариса', 'Маша']*
s.insert(2,'Катя')
print(s) → *[1.3, 7, 'Катя', True, 'Лариса', 'Маша']*
s.remove(7)
print(s) → *[1.3, 'Катя', True, 'Лариса', 'Маша']*
print(s.index(True)) → *2*
print(len(s)) → *5*
s.reverse()
print(s) → *['Маша', 'Лариса', True, 'Катя', 1.3]*

Для реверса можно также применить синтаксис (пока несколько загадочный), показанный в следующем примере:

s = [1.3, 7, True, 'Лариса']
s1 = s[::-1]
print(s1) → *['Лариса', True, 7, 1.3]*
print(s) → *[1.3, 7, True, 'Лариса']*

При этом исходный список не изменяется, а создаётся новый список. Применение этой техники обсудим позднее.

Функция **sum** суммирует числовые элементы списка:

s = [2, 4.3, 1.2, 7]
print(sum(s))→ *14.5*

Функция **max** возвращает максимальный, а функция **min —** минимальный элемент списка:

s = [2, 4.3, 1.2, 7]
print(max(s)) → *7*
print(min(s)) → *1.2*

Знак сложения (+) служит знаком конкатенации:

s1 = [1.3, 7, True, 'Лариса']
s2 = [5, 6, 7]
s = s1 + s2
print(s) → *[1.3, 7, True, 'Лариса', 5, 6, 7]*

А знак умножения (*) позволяет повторять элементы в списке:

s = [1, 2, 3]
s1 = s * 3
print(s1) → *[1, 2, 3, 1, 2, 3, 1, 2, 3]*

Тип каждого элемента списка можно узнать после его извлечения:

s = [1.3, 7, True, 'Лариса']
print(type(s[1])) → *<class 'int'>*

Функция **count** определяет, сколько раз в списке встречается заданный элемент:

s = [7, 0.3, 7, 'hello', 2, 7, 3, 7, 4]
print(s.count(7)) → *4*

Элементами списка могут быть списки:

s = [1, 2, 3, [0.3, 1.2, 77], 11, 22]
print(s[3]) → *[0.3, 1.2, 77]*

Функция **pop** возвращает последний элемент и удаляет этот элемент из списка:

s = [1.3, 7, True, 'Лариса']
print(s.pop()) → *Лариса*
print(s) → *[1.3, 7, True]*

Эта функция полезна при организации стека.

Функция **sort** позволяет сортировать списки:

s = [9,3,6,1,8,4]
list.sort(s)
print(s) → *[1, 3, 4, 6, 8, 9]*

Функция **sort** принимает ещё и атрибут **reverse**, если ему присвоить значение **True** (по умолчанию он равен **False**), то сортировка будет выполнена по убыванию:

s = [9,3,6,1,8,4]
list.sort(s, reverse = True)
print(s) → *[9, 8, 6, 4, 3, 1]*

Цикл **for** (о циклах далее) позволяет извлекать элементы поочерёдно. При этом над элементами можно выполнять какие-нибудь действия:

s = [3, 4, 5, 6, 7]
for x in s: print(x * x,'', end = '') → *9 16 25 36 49*

Здесь использовано следующее свойство оператора **print**: если в конце поставить выражение **end = ''**, то оператор **print** выводит данные без перевода строки, который заменяется тем, что стоит внутри кавычек. При пустых кавычках вместо перевода строки ничего не выводится.

Python позволяет использовать ранги (или диапазоны), которые можно считать разновидностью списков:

r = range(7)
print(r) → *range(0, 7)*
print(type(r)) → *<class 'range'>*
print(r[2]) → *2*

Значит, можно извлекать числа из ранга по индексу.

Ранги удобно применять для инициализации списков:

s = list(range(5, 10))
print(s) → **[5, 6, 7, 8, 9]**

Значит верхняя граница не входит в ранг, и, соответственно, в список.

Функция **range** возвращает ранг, а функция **list** создаёт список. Ранг может быть и с заданным шагом:

s = list(range(0, 15, 3))
print(s) → *[0, 3, 6, 9, 12]*

Можно и по убыванию:

s = list(range(18, 3, -3))
print(s) → *[18, 15, 12, 9, 6]*

Для рангов можно применять функцию **sum**:

x = sum(range(4, 7))
print(x) → 15

С помощью цикла **for** можно создавать самые разные списки. Например, создадим список с элементами-списками:

s = [[x ** 2, x ** 3] for x in range(4)]
print(s) → *[[0, 0], [1, 1], [4, 8], [9, 27]]*

Или ещё более сложный пример:

s = [[x, x / 2, x * 2] for x in range(-6, 7, 2) if x > 0]
print(s) → *[[2, 1.0, 4], [4, 2.0, 8], [6, 3.0, 12]]*

Здесь использован условный оператор **if**, смотрите далее.

Кроме обычных срезов, есть возможность делать разные выборки из списков с помощью необязательного третьего индекса (в документации этот приём называют словом **stride** «шаг»). Покажу на примере:

m = list(range(0,10))

print(m) → *[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]*
m1 = m[1:10:2]
print(m1) → *[1, 3, 5, 7, 9]*
m2 = m[3:8:2]
print(m2) → *[3, 5, 7]*

Значит, третий индекс на самом деле задаёт шаг для выборки. Границы выборки можно не указывать явно, это будет равнозначно границам всего списка:
m1 = m[::2]
print(m1) → *[0, 2, 4, 6, 8]*
Если задать значение шага отрицательным, перебор элементов будет в обратном порядке, что позволяет выполнить реверс:
m3 = m[9:0:-1]
print(m3) → *[9, 8, 7, 6, 5, 4, 3, 2, 1]*

Нулевой элемент не попал в выборку потому, что ранг не включает последний элемент. Чтобы нулевой элемент присутствовал, надо не указывать нижнюю границу: **m3 = m[9::-1].** Можно опустить также и верхнюю и тогда получим тот способ реверса, который мы уже применяли :
m3 = m[::-1]
print(m3) → *[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]*

В дальнейшем мы рассмотрим ещё много приёмов работы со списками.

При работе с функциями, изменяющими списки «на месте», надо быть внимательным, чтобы не допустить ошибки. Например:
m = [1, 2]
s = m
m.append(3)
print(m) → *[1, 2, 3]*
print(s) → *[1, 2, 3]*

Здесь изменение списка **m** приводит к изменению списка **s**, так как в этом случае идентификаторы **m** и **s** ссылаются на один и тот же список. Такое невозможно для неизменяемых коллекций, например для кортежей.

Часто похожие операции ведут себя по-другому, например:
m = [1, 2]
s = m
m = m + [3]
print(m) → *[1, 2, 3]*
print(s) → *[1, 2]*

И это всегда надо иметь ввиду. Иногда предусмотрена некоторая защита от возможных ошибок. Например, если мы напишем:
y = y.sort()

То получим результат, равный значению **None**, поскольку функция **sort** изменяет список на месте. По значению **None** ошибку легко обнаружить.

Строки

Строковые (или текстовые) переменные похожи на списки. Практически это тоже списки с элементами в виде букв (или каких-то других знаков). В частности, символы можно извлекать по индексу:
x = 'Москва'
print(x[3]) → *к*

Как и для списков, отдельные буквы можно извлекать с помощью группового присваивания:
x = 'Москва'
a, b, c, d, e, f = x
print(b, d, e) → *о к в*

К строкам применимы некоторые из функций, которые мы применяли для списков, например:
s = "мама мыла раму"
print(s.index("ы")) → *6*
print(len(s)) → *14*
s1 = s[::-1]
print(s1) → *умар алым амам*
print(type(s[3])) → *<class 'str'>*
print(s.count("м")) → *4*

Для строк можно получать срезы:
s = "Ленинград"
s1 = s[3:6] → *инг*
print(s1)
s2 = s[4:] → *нград*
print(s2)
s3 = s[:5]
print(s3) → *Ленин*

Знак умножения (*) позволяет повторять строку:
s = 3 * "Петя"
print(s) → *ПетяПетяПетя*

Как и для списков, знак **+** служит знаком конкатенации:

s1 = 'мама '
s2 = 'мыла раму'
s = s1 + s2 → *мама мыла раму*

В Python строки в двойных и в одинарных кавычках мало отличаются друг от друга, но разница всё же есть. Например, внутри текста в двойных кавычках могут быть одинарные кавычки, в английском они являются апострофом. Справедливо и обратное:
print("abc'defg") → *abc'defg*
print('abc"defg') → *abc"defg*

Строки в Python неизменяемы (immutable) и мы не можем, например, в строке изменить букву:
s = "Масква"
s[1] = 'о' → *ошибка*
print(s)

Исправить здесь букву **а** можно как-нибудь так:
s = "Масква"
s1 = s[0] + 'о' + s[2:]
print(s1) → *Москва*

То-есть, мы не изменяем строку, а создаём новую.

Функция **split** преобразует строку в список:
s = 'У лукоморья дуб зелёный'
print(s.split()) → *['У', 'лукоморья', 'дуб', 'зелёный']*

Функция **split** может принимать аргумент, который определят по какому символу разбивать текст на элементы. Например нашу строку можно разбить по букв «**о**»:
s = 'У лукоморья дуб зелёный'
print(s.split("о")) → *['У лук', 'м', 'рья дуб зелёный']*

Сам этот символ не входит в слова-элементы. При отсутствии аргумента разбивка выполняется по пробелам, но его можно указать и явно:
s.split(" ")

Функция **list** трансформирует текст в список, разбивая его на символы:
s = 'У лукоморья дуб зелёный'
print(list(s)) →
['У', ' ', 'л', 'у', 'к', 'о', 'м', 'о', 'р', 'ь', 'я', ' ', 'д', 'у', 'б', ' ', 'з', 'е', 'л', 'ё', 'н', 'ы', 'й']

Поскольку в списке можно изменять элементы, то изменить букву в тексте можно так:
s = 'У лукоморья дуб зелёный'

s1 = list(s)
s1[2] = 'Л'
print(''.join(s1)) → *У Лукоморья дуб зелёный*

Функция **join** выполняет обратную трансформацию списка в текст. Эту функцию надо вызвать для пустого разделителя ''. Если задать какой-нибудь символ, то он будет разделять все буквы текста:
print(':'.join(s1)) → *У: :Л:у:к:о:м:о:р:ь:я: :д:у:б: :з:е:л:ё:н:ы:й*

Функция **upper** переводит все буквы текста в верхний регистр:
s = 'abc'
print(s.upper()) → *ABC*

Функция **find** возвращает индекс первой буквы заданного текста в строке (если этот текст там есть, а если нет, то возвращается **-1**), а функция **replace** позволяет заменить часть строки на заданный текст:
s = 'подари мне платок'
print(s.find('мне')) → *7*
s1 = s.replace('платок', 'букет')
print(s1) → *подари мне букет*

При замене текста создаётся новая строка, а исходная строка не изменяется.

Интерполяция

Сочетание в строках обратного слеша с буквой образует так называемую управляющую последовательность. Так знаки **\n** воспринимаются, как перевод строки. Есть и другие подобные комбинации. Если перед строкой поставить букву **r**, то управляющие последовательности отменяются и вся строка выводится «как есть»:
print('Привет \nМир') → *Привет*
Мир
print(r'Привет \nМир') → *Привет \nМир*

Вставку каких-либо данных в строку при выводе обычно называют *интерполяцией*. На Python такая интерполяция выполняется с помощью функции **format**. Для этого внутри текста надо вставить пустые фигурные скобки, а затем применить к тексту функцию **format** в точечной нотации, передав ей вставляемые данные. Эти данные будут вставлены на место фигурных скобок. Например:
v = 7.2
c = 2.34
print("Стоимость товара {} рублей".format(v * c)) →

Стоимость товара 16.848 рублей

Есть также вариант интерполяции без явного вызова функции **format**:

x = 2; y = 3
print('цена = %s, количество = %s, стоимость товара = %s' %(x, y, x * y)) → *цена = 2, количество = 3, стоимость товара = 6*

Здесь значения вставляются в текст на место знаков **%s**. Вставляемые значения помещаются в кортеж перед которым должен быть знак **%**.

А это ещё один вариант интерполяции с явным вызовом функции **format**:

x = 2; y = 3
print('цена = {0}, количество = {1}, стоимость товара = {2}'.format(x, y, x * y)) → *цена = 2, количество = 3, стоимость товара = 6*

Цифры в фигурных скобках указывают, какое значение из прилагаемого списка будет поставлено на это место.

x **= 2; y = 3**
print('цена = {2}, количество = {1}, стоимость товара = {0}'.format(x, y, x * y)) → *цена = 6, количество = 3, стоимость товара = 2*

Есть также возможность разбить с помощью запятых длинное число на тройки цифр и задать количество знаков после запятой с округлением. На следующем примере показан синтаксис этих операций:

x = 2969997.2567
print('{:,.2f}'.format(x)) → *2,969,997.26*

Можно также задать длину поля для выводимого значения:

x = 3.14159; y = -42
print('%.3f, %+9d' % (x, y)) → *3.142, -42*

При знаке плюс значение прижато к правому краю выделенного поля, а при знаке минус будет прижато к левому краю:

x = 3.14159; y = -42
print('%.3f, %-9d конец' % (x, y)) → *3.142, -42 конец*

В формате для чисел с плавающей запятой применяется буква **f**, для целых чисел — буква **d**, для строк — буква **s**. Для типа *bool* при формате **s** выводится слово **True** или **False**, а при формате **d** — цифра **1** или **0**.

Оператор **print** после вывода строки выполняет перевод на новую строку. Специальный оператор **end** позволяет не выполнять этот

перевод. Слово **end** надо поставить после выводимого текста через запятую, а после знака равенства можно указать, что надо вставить в конце строки. Это может быть пробел, или запятая, или произвольный текст. Например, вычислим числа Фибоначчи и выведем не столбиком, а в строке:

```
a, b = 0, 1
while a < 100:
    print(a, end = ', ')
    a, b = b, a+b
```
→ *0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,*

Здесь мы использовали цикл **while**, а для вычисления чисел Фибоначчи потребовалось всего лишь групповое присваивание.

Кортежи (tuple)

Кортежи (*tuple*) относятся к упорядоченным коллекциям и во многом похожи на списки. Они представляют набор элементов любого типа в круглых скобках (скобки можно не применять, но это ухудшает читабельность кода):

```
x= ('a', 1, 'python', (1, 2))
print(x)
```
→ *('a', 1, 'python', (1, 2))*

print(x[2]) → *python*

print(type(x)) → *<class 'tuple'>*

Элементы кортежей можно извлекать по индексам. Кортежи являются неизменяемыми (*immutable*), поэтому нельзя, например, присвоить элементу новое значение.

При инициализации кортежа с одним элементом после этого элемента надо поставить запятую:

```
t = ("a",)
print(t)
```
→ *('a',)*

print(type(t)) → *<class 'tuple'>*

Иначе будет создана, например, строка:

```
t = ("a")
print(t)
```
→ *a*

print(type(t)) → *<class 'str'>*

Для создания кортежей есть также функция **tuple**, которая возвращает кортеж для любого своего аргумента:

```
t = tuple('a')
print(t)
```
→ *('a',)*

print(type(t)) → *<class 'tuple'>*

Если единственный аргумент представлен строкой, списком или кортежем, **tuple** создаёт кортеж с элементами этих коллекций:

t = tuple('Воронеж')
print(t) → *('В', 'о', 'р', 'о', 'н', 'е', 'ж')*
t = tuple([1, 3, 'abc', True])
print(t) → *(1, 3, 'abc', True)*

Кортежи позволяют делать срезы с теми же возможностями, как и для списков:

t = (3, 4, 5, 6, 7, 8, 9)
print(t[1:4]) → *(4, 5, 6)*
print(t[3:]) → *(6, 7, 8, 9)*

Хотя кортежи и неизменяемые, можно замещать один кортеж другим:

t = (1, 2, 3, 4, 5)
t = ('a', 'b', 'c') + t[2:]
print(t) → *('a', 'b', 'c', 3, 4, 5)*

Кортежи, списки и строки можно сравнивать (операторы ==, >, >=, <, <=, !=). Сравнение выполняется так: сравниваются первые элементы, если они равны, сравниваются следующие элементы и так далее до тех пор пока не встретятся неравные элементы. Результат сравнения последних возвращается, как результат операции.

t = (1, 2, 3, 4, 5)
t1 = (1, 2, 7, 0)
s = [1, 2, 3, 9, 4, 8]
s1 = [1, 2, 3, 4]
print(t1 > t) → *True*
print(s < s1) → *False*

Оставшиеся элементы не принимаются во внимание, а размеры сравниваемых коллекций не обязательно должны быть равны. В следующем примере используется возможность сравнения кортежей для сортировки слов в строке по длине слов:

```
s = 'у лукоморья дуб зелёный'
w = s.split()
t = list()
for x in w: t.append((len(x), x))
t.sort(reverse=True)
r = list()
for _, w in t: r.append(w)
print(r) → ['лукоморья', 'зелёный', 'дуб', 'у']
```

Здесь сначала с помощью функции **split** преобразуем текст **s** в список **w**, элементы которого представлены словами текста. Затем в цикле создаём список **t** с элементами в виде пар (кортежей), первый элемент которых содержит длину слов, а второй — сами слова. Сортируем список **t** по первым элементам кортежей (по длине слов) и выполняем его реверс. Наконец в цикле создаём список **r**, помещая в него вторые элементы пар (слова). Поскольку первые элементы пар (длины слов) нам не нужны, заменяем их идентификатор на нижнюю черту (знакозаменитель).

Множества (set)

К неупорядоченным коллекциям относятся множества (**set**), представляющие набор элементов любых типов в фигурных скобках:
x = {7, 2, 'a', 2, 5, 2, 1}
print(x) → *{1, 2, 5, 'a', 7}*
print(type(x)) → *<class 'set'>*

Множества содержат только оригинальные элементы, дублирующие элементы удаляются. Порядок расположения элементов в множестве может меняться при операции с ними. В примере элементы оказались отсортированными при создании множества.
m = {2, 5, 7, 2, 3, 2, 9, 2}
print(m) → *{2, 3, 5, 7, 9}*

Множества неизменяемы, элементы множеств нельзя извлекать по индексам. Иногда их называют наборами. Наборы имеют очень быстрое время поиска и применение их особенно эффективно, когда есть большая коллекция вещей и эти вещи отыскиваются по их имени. Множества позволяют легко устранять дублирующиеся элементы в списке. Для этого достаточно преобразовать список в множество с помощью функции **set**, а затем это множество преобразовать обратно в список с помощью функции **list**:
m = [2, 3, 7, 3, 9, 3]; s = set(m); m = list(s)
print(m) → *[9, 2, 3, 7]*

Конечно, при этом мы теряем первоначальный порядок элементов. Можно написать и так:
print(list(set([2, 3, 7, 3, 9, 3]))) → *[9, 2, 3, 7]*

Приведу ещё несколько примеров операций над множествами:
print({1, 2, 3, 4, 5}.union({3, 4, 5, 6})) → *{1, 2, 3, 4, 5, 6}*
print({1, 2, 3, 4, 5} | {3, 4, 5, 6}) → *{1, 2, 3, 4, 5, 6}*

print({1, 2, 3, 4}.difference({2, 3, 5})) → *{1, 4}*
print({1, 2, 3, 4} - {2, 3, 5}) → *{1, 4}*
print({1, 2, 3, 4}.symmetric_difference({2, 3, 5})) → *{1, 4, 5}*
print({1, 2, 3, 4} ^ {2, 3, 5}) → *{1, 4, 5}*
print({1, 2}.issuperset({1, 2, 3})) → *False*
print({1, 2} >= {1, 2, 3}) → *False*
print({1, 2}.issubset({1, 2, 3})) → *True*
print({1, 2} <= {1, 2, 3}) → *True*
print({1, 2}.isdisjoint({3, 4})) → *True*
print({1, 2}.isdisjoint({1, 4})) → *False*

(*disjoint*: дизъюнктный - не иметь общих элементов)

В модуле **collections** есть класс **Counter**, преобразующий множество в хеш (смотрите далее):

```
from collections import Counter
h = Counter(['a','b','b','c'])
```
print(h) → *Counter({'b': 2, 'a': 1, 'c': 1})*

Здесь элементы служат ключами, а значения задают число дублирующихся элементов.

Хеш (hash)

Для этого вида коллекций применяют разные названия: ассоциированный список, словарь (*dictonary*), мап (*map* – отображение), хеш (*hash*). Вообще-то **hash** переводится, как мусор, но будем использовать этот термин, как более краткий и легко запоминающийся (слово **map** нам потребуется для других целей). Обычно пишут «хэш», но мне кажется правильнее «хеш», впрочем это несущественно.

Хеш представляет сбой коллекцию из пар ключ (**key)**-значение (**value**). Синтаксически элементы хеша записываются в фигурных скобках, пары разделяются запятыми, значения от ключей отделяются двоеточием:

h = { 'Люда' : 25, 'Лена' : 19, 'Катя' : 21 }
print(h) →*{'Люда': 25, 'Лена': 19, 'Катя': 21}*
print(type(h)) → *<class 'dict'>*

Здесь имена представляют ключи, возраст — значения. Как видим, тип хешей называется **dict**.

Хеш во многом похож на список, в котором роль индексов выполняют ключи:

h = { 'Люда' : 25, 'Лена' : 19, 'Катя' : 21 }
print(h['Лена']) → *19*

Вообще-то элементы хеша неупорядоченные и их расположение может изменяться при манипуляциях над хешем. Ключи и значения могут быть представлены значениями разных типов без ограничений:

h = { 'abc' : 12.5, 123: 'hello', 'x' : False }
print(h) → *{'abc': 12.5, 123: 'hello', 'x': False}*

Хеши изменяемы (mutable) и значения можно изменять «на месте»:

h = { 'abc' : 12.5, 123: 'hello', 'x' : False }
h['abc'] *= 2
print(h) → *{'abc': 25.0, 123: 'hello', 'x': False}*

В хеш можно добавлять новые пары:

h = {}
h['имя'] = 'Пётр'
h['возраст'] = 35
h['профессия'] = 'царь'
print(h) → *{'имя': 'Пётр', 'возраст': 35, 'профессия': 'царь'}*

В качестве значений в хеше могут применяться коллекции всех видов и, в частности, хеши тоже. Например:

h = {'гражданин': {'имя': 'Борис', 'фамилия': 'Уваров'},'дети': ['Виктор', 'Наталья'], 'возраст': 40.5}
print(h) → *{'гражданин': {'имя': 'Борис', 'фамилия': 'Уваров'}, 'дети': ['Виктор', 'Наталья'], 'возраст': 40.5}*
print(h['дети']) → *['Виктор', 'Наталья']*
print(h['гражданин']['фамилия']) → *Уваров*
h['дети'].append('Мария')
print(h) → *{'гражданин': {'имя': 'Борис', 'фамилия': 'Уваров'}, 'дети': ['Виктор', 'Наталья', 'Мария'], 'возраст': 40.5}*

Даже ключи могут быть представлены кортежами.

h = { ('Борис', 'Уваров') : 45, ('Виктор', 'Борисов') : 50 }
print(h) → *{('Борис', 'Уваров'): 45, ('Виктор', 'Борисов'): 50}*

Оператор **in** позволяет проверить наличие заданного ключа в хеше:

h = { 'a' :1, 'b' : 2, 'c' : 3 }
print('b' in h) → *True*
print(not 'f' in h) → *True*

Функция **keys** выделяет все ключи в хеше, а функция **list** позволяет получить список, в котором элементы представлены ключами хеша:

h = { 'a' :1, 'b' : 2, 'c' : 3 }
print(h.keys()) → *dict_keys(['a', 'b', 'c'])*
s = list(h.keys())

print(s) → *['a', 'b', 'c']*

Соответственно, функция **values** позволяет получить список значений:

h = { 'a' :1, 'b' : 2, 'c' : 3 }
print(h.values()) → *dict_values([1, 2, 3])*
s = list(h.values())
print(s) → *[1, 2, 3]*

Если вызвать значение по несуществующему ключу, получим ошибку. Но в модуле **defaultdict** есть функция **defaultdict**, которая позволяет задать значение по умолчанию, которое будет получено по любому несуществующему ключу. Тут применяется такой синтаксис:

from collections import defaultdict
h = {1:'aa', 2:'bb', 3:'cc'}
h = defaultdict(lambda: 'Hello')
print(h[5]) → *Hello*

Хеш **h** не имеет ключа со значением **5**, но ошибка не зафиксирована и получено значение по умолчанию.

Вызов функции **defaultdict** всегда должен находиться после объявления хеша. Слово **lambda** объявляет безымянную функцию, о которых смотрите далее.

Массивы (array)

Кроме списков Python позволяет применять массивы. В основном массивы похожи на списки, а главное их отличие в том, что элементы массивов должны быть строго одного типа. Чтобы использовать массив надо импортировать модуль **array**. Это связано с тем, что массив в Python не является фундаментальным типом данных. Команда импорта имеет такой вид:

from array import *

Теперь можно объявить массив и это объявление имеет оригинальный синтаксис:

имя = array (знак типа, [элементы])

Здесь **имя** – идентификатор массива, а «знак типа» - условное обозначение типа элементов. Приняты такие обозначения:

b – знаковое целое число (размером 1 байт)
B – целое число без знака (1 байт)
c – символ (1 байт)
u – символ unikode (2 байта)

i - знаковое целое (2 байта)
I – целое без знака (2 байта)
w – символ unikode (4 байта)
l — знаковое целое (4 байта)
L - целое без знака (4 байта)
f – число с плавающей запятой (4 байта)
d – число с плавающей запятой (8 байт)

Эти условные обозначения типа должны указываться в кавычках. Таким образом, массив позволяет управлять и строго контролировать тип его элементов. Покажем пример объявления массива и извлечение элемента по индексу.

```
from array import *
m = array('d', [1,2,3,4]) → 1.0 2.0 3.0 4.0
for i in m: print(i, end = ' ')
print('\n', m[2]) → 3.0
```

Массивы mutable и как и для списков могут применяться следующие функции для их изменения: *append, insert, remove, pop, index, reverse:*

```
from array import *
m = array('d', [1,2,3,4])
m.append(6); print(m) →array('d', [1.0, 2.0, 3.0, 4.0, 6.0])
m.insert(2,7); print(m) →array('d', [1.0, 2.0, 7.0, 3.0, 4.0, 6.0])
m.remove(3); print(m) →array('d', [1.0, 2.0, 7.0, 4.0, 6.0])
m.pop(); print(m) → array('d', [1.0, 2.0, 7.0, 4.0])
print(m.index(7)) → 2
m.reverse(); print(m) →array('d', [4.0, 7.0, 2.0, 1.0])
```

Как видим, команда **print(m)** выводит массив **m** в том же виде, в каком он объявляется.

Метод **count** определяет число вхождений заданного элемента в массив, а методы **tostring** и **tolist** конвертируют массив в строку и список, соответственно:

```
from array import *
m = array('d', [1,2,3,4])
s = m.tolist()
print(s) → [1.0, 2.0, 3.0, 4.0]
```

Функции

Функции в Python объявляются со словом **def**, после которого располагается идентификатор (имя) функции и список аргументов в круглых скобках. При отсутствии аргументов скобки обязательны. После списка аргументов ставится двоеточие, после которого располагается тело функции (в документации называется блоком) . Блок может быть в той же строке, если он там помещается целиком:

def f(x, y): z = x + y; z = z + 1; return z
print(f(3, 9)) → *13*

Чаще всего тело надо начинать с новой строки с обязательным отступом. Величина отступа может быть любой, обычно его принимают равным четырём пробелам. Следующие строки надо располагать точно с тем же отступом, при этом надо использовать только пробелы или только клавишу *Tab*:

```
def f(x, y):
    z = x + y
    z = z + 1
    return z
print(f(3, 9)) → 13
```

Все строки кода расположенные далее с отступом входят в блок, который заканчивается там, где появляется строка без отступа. Таким образом, форматирование в Python влияет на функциональность, а фигурные скобки, как это принято во многих языках, не применяются. Лично мне такая практика совсем не нравится, но тут уж ничего не поделаешь. Часто могут быть ошибки, например нельзя сделать так:

```
def f(x, y): z = x + y;
    z = z + 1; return z
print(f(3, 9))
```

Но можно, например, так

```
def f(x, y):
    z = x + y; z = z + 1;
    return z
print(f(3, 9))
```

Большинство редакторов (IDE) форматирование поддерживают автоматически.

Результат функцией возвращается с помощью оператора **return**, без которого получим *None*:

def f(x, y): z = x + y
print(f(3, 9)) → *None*

Тип аргументов функции обычно выводится из тех значений, которые будут переданы функции, но тип можно указать и явно:

def f(x: int, y: int): return x + y
print(f(3, 9)) → *12*
print(f('Hello, ', 'world!')) → *Hello, world!*

В этом примере объявление типа аргументов ничего не даёт, поскольку при задании значений другого типа (у нас это тип **str**) тип аргументов автоматически меняется. Тип **str** здесь допустим потому, что оператор + для этого типа тоже имеет смысл и является знаком конкатенации (объединяет две строки в одну).

Можно задать тип возвращаемого функцией результата, для чего применяется стрелка после списка аргументов:

def f(x: int, y: int) -> int: return x + y

При этом можно указать любой тип, например можно написать так:

def f(x: int, y: int) -> str: return x + y
print(f(3, 9) + "aa")

Однако, на самом деле тип результата всё равно будет **int** и приведённый код не будет работать.

Как и все другие сущности (terms) в Python, функции также имеют тип, который называется *function*:

def f(x): return x2**
print(f(5)) → *25*
print(type(f)) → *<class 'function'>*

Python имеет много встроенных (стандартных) функций и модулей. Можно получить их список, выполнив команду:

print(dir(__builtins__))

Получим огромный список функций. Функции с идентификатором, ограниченным двумя знаками подчёркивания, играют особую роль и называются атрибутами. Мы познакомимся с ними позднее.

Информацию о конкретной функции можно получить командной:

print(help(max))→ сведения о функции **max**

Математические функции находятся в модуле **math**, который при использовании функций надо импортировать:

import math
print(math.sin(1.5)) → *0.9974949866040544*

При этом название модуля надо указывать ещё и при вызове функции. Полный список функций в модуле можно получить командной:

import math

print(dir(math))

Python имеет большой список математических функций:
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'sumprod', 'tan', 'tanh', 'tau', 'trunc', 'ulp']

Функции *__doc__', '__loader__', '__name__', '__package__', '__spec__* из этого списка вообще-то не являются математическими, функции такого вида рассмотрим позднее.

Можно импортировать не весь модуль, а только нужную функцию. Для этого применяется слово **from**:

from math import sin
print(sin(1.5)) → *0.9974949866040544*

В этом варианте не надо указывать название модуля перед именем функции при её вызове.

Аргументы функций могут быть заданы «по умолчанию» (все, или только часть из них). Если не все аргументы по умолчанию, то те, что по умолчанию должны быть в конце списка. Аргументы по умолчанию могут быть изменены при вызове функции (тоже все, или часть из них). Например:

```
def f(x = 2, y = 3):
    a = x**2
    b = x + y
    return b - a
```
print(f()) → *1*
print(f(3)) → *-3*
print(f(1, 2)) → *2*

При вводе значений аргументов при вызове функции они всегда могут задаваться именованными:

```
def f(x, y):
    a = x**2
    b = x + y
    return b - a
```
print(f(y = 3, x = 2)) → *1*

Именованные аргументы могут задаваться в произвольном порядке, что особенно удобно при длинном списке аргументов.

Аргументы по умолчанию также могут задаваться, как именованные, что позволяет создавать разные комбинации значений аргументов:

```
def f(x = 2, y = 1):
    a = x**2
    b = x + y
    return b - a
print(f(y = 3)) → 1
```

Количество аргументов функции может быть не фиксированным заранее. Для этого надо перед идентификатором последнего аргумента поставить звёздочку и тогда можно передавать функции произвольное количество значений. Все «лишние» значения будут объединены в кортеж и присвоены последнему аргументу:

```
def f(x, y, *z):
    print(x, ',', y) →2 , 3
    print(z) → (4, 5, 6)
f(2, 3, 4, 5, 6)
```

Фактически это то же самое, как если бы мы передали последнему аргументу в списке значение в виде кортежа:

```
def f(x, y, z):
    print(x, ',', y) →2 , 3
    print(z) →(4, 5, 6)
f(2, 3, (4, 5, 6))
```

Последний аргумент может также иметь две звёздочки. В этом случае надо значения передавать, как именованные аргументы, присваивая им новые имена при вводе. Тогда все «лишние» значения будут представлены, как хеш:

```
def f(x, **y):
    print(x) → 2
    for i in y: print(i,', ', end='') →a , b , c ,
    print(y) →{'a': 3, 'b': 4, 'c': 5}
f(2, a=3, b=4, c=5)
```

В теле функции могут быть вложенные функции:

```
def g(t): return t**2
def f(x):
    y = g(x) + 3; return y
print(f(5)) → 28
```

Здесь аргументы функций **g** и **f** имеют разные идентификаторы, но это необязательно, идентификаторы могут быть одинаковые, например, **g(x)** и **f(x)**. Конфликта имён не происходит, так как это

разные переменные. Всегда лучше иметь как можно меньше разных идентификаторов в программе.

Функция всегда должна быть объявлена раньше того выражения, где эта функция вызывается. В примере выше можно было сделать так:

```
def f(x):
    y = g(x) + 3; return y
def g(x): return x**2
print(f(5))
```

Но нельзя объявить функцию **g** после оператора **print**, при выполнении которого произойдёт вызов функции **g**:

```
def f(x):
    y = g(x) + 3; return y
print(f(5))
def g(x): return x**2
```

В таком варианте интерпретатор выдаст ошибку.

Переменные, объявленные вне функции, являются по отношению к ней глобальными и доступны в теле функции. Объявленные в теле функции переменные, являются локальными и за пределами функции не видны. Локальная переменная может иметь тот же идентификатор, что и глобальная, но в таком случае локальная переменная временно маскирует глобальную и тогда глобальная переменная в теле функции недоступна. Например:

```
x = 5
def f(y):
    x = y**2
    return x
print(f(2)) → 4
print(x) → 5
```

Здесь нельзя в теле функции написать, например, такую строку:

```
x = y**2 + x
```

Пожалуй, в подобных ситуациях лучше для глобальных и локальных переменных применять разные идентификаторы для устранения подобных ошибок, а также для улучшения читабельности. Применение одного и того же идентификатора для глобальных и локальных переменных часто чревато ошибками. Например, следующий код работает нормально:

```
x = 10
def f(): print(x)
f() → 10
```

Но если мы запрограммируем так:

```
x = 10
def f():
    print(x)
    x += 1
f()
```

то получим ошибку о том, что здесь попытка использовать переменную **x** (в строке **print(x)**) раньше, чем она была определена. То-есть, переменная **x** не считается глобальной даже там, где она использована раньше, чем определена локальная переменная с тем же именем. Имеется модификатор **global** позволяющий принудительно объявить переменную глобальной. При этом она остаётся глобальной, даже если мы объявим её в теле функции заново:

```
x = 10
def f():
    global x
    print(x) → 10
    x = 7
    print(x) → 7
f()
print(x) → 7
```

Как видим, глобальная **x** была изменена в теле функции.

Есть также модификатор **nonlocal**, который применяют для вложенных функций:

```
def f():
    x = 10
    def g():
        nonlocal x
        print(x) → 10
        x += 1
    g()
    print(x) → 11
f()
```

Python также позволяет применять безымянные функции. Часто их называют лямбда-выражениями, иногда closure (замыкание). Делать функцию безымянной удобно, когда она вызывается только один раз и нет смысла присваивать ей имя (идентификатор). Чаще в качестве безымянных применяют простые функции с телом из одного выражения. От обычных функций безымянные отличаются тем, что объявляются с ключевым словом **lambda**, а аргументы не

заключаются в скобки. Кроме того, не используется оператор **return** для возвращения результата. Для вызова безымянной функции надо предать ей значения аргументов в круглых скобках, например:

```
r = (lambda x: x**2)(3)
print(r) → 9
```

Аргументов может быть больше одного:

```
r = (lambda x, y: x**2+y**2)(3,4)
print(r) → 25
```

Тело безымянной функции может быть представлено несколькими выражениями, разделёнными запятыми и заключёнными в круглые скобки. В этом случае результат будет представлен в виде кортежа, элементы которого равны результатам вычисления всех выражений:

```
r = (lambda x, y: (a := x**2, b := y**2, a + b))(3,4)
print(r) → (9, 16, 25)
```

Выражения в теле функции могут записываться на разных строках, а отступы в этом случае не имеют значения:

```
r = (lambda x, y:
(a := x**2,
b := y**2,
a + b))(3,4)
print(r)→ (9, 16, 25)
```

Если применить квадратные скобки, вместо кортежа получим список:

```
r = (lambda x, y: [a := x**2, b := y**2, a + b])(3,4)
print(r) → [9, 16, 25]
```

При фигурных скобках будем иметь множество.

Безымянной функцией можно инициировать переменную, которая станет обычной функцией:

```
f = lambda x, y: x**2+y**2
print(f(3, 4)) → 25
```

Обычная функция может возвращать безымянную функцию в качестве результата:

```
def f(a): return lambda x: x ** a
g = f(3)
print(g(4)) → 64
```

Передавая функции **f** значение аргумента **a**, получаем функцию **g** с одним аргументом **x**.

Можно, конечно, и так:

```
def f(): return lambda x, a: x ** a
g = f()
```

print(g(4, 3)) → *64*

Чтобы не путаться с порядком следования аргументов, их можно сделать именованными:

```
def f(): return lambda x, a: x ** a
g = f()
print(g(a = 3, x = 4)) → 64
```

Количество аргументов у функции называется её арностью. С помощью безымянной функции можно выполнять операцию понижения арности. Такая операция называется каррированием. Например, функцию двух переменных (арность равна 2) можно каррировать до функции одного переменного (арность равна 1):

```
def f(x, y): return x + y
g = lambda t: f(3, t)
print(g(4)) → 7
```

Задав значение одного аргумента функции **f** мы получили функцию **g** имеющую один аргумент. Для ясности здесь также лучше применять именованные аргументы:

```
def f(x, y): return x + y
g = lambda t: f(x = 3, y = t)
print(g(t = 4)) → 7
```

Вовсе не обязательно вводить новый идентификатор **t**:

```
def f(x, y): return x + y
g = lambda y: f(3, y)
print(g(y = 4)) → 7
```

Применяя последовательно цепочку безымянных функций можно арность больше двух также понизить до единицы.

Рекурсивные функции на Python создаются очень просто. Запрограммируем, например, вычисление факториала:

```
def f(n, r = 1):
    if n <= 1: return r
    else: return f(n-1, r * n)
print(f(5)) → 120
```

(В модуле *math* есть стандартная функция *factorial* для вычисления факториала).

Вот так можно запрограммировать вычисление суммы элементов списка:

```
def f(m):
    if not m: return 0
    else: return m[0] + f(m[1:])
print(f([1, 2, 3, 4, 5])) → 15
```

Здесь мы воспользовались тем, что пустой список в условных выражениях воспринимается, как **False** и значит выражение **not m** при **m = []** даст **True**. При рекурсивном вызове функции здесь использован срез списка без первого (нулевого) элемента.

Более сложный случай для списка произвольной структуры:

```
def f(L):
    s = 0
    for x in L:
        if not isinstance(x, list): s += x
        else: s += f(x)
    return s
L = [1, [2, [3, 4], 5], 6, [7, 8]]
print(f(L)) → 36
```

Если элемент списка не является списком, то он добавляется к сумме, а если это список, то происходит рекурсивный вызов функции **f** для этого элемента-списка.

Ветвления и циклы

Python использует обычный условный оператор **if-else**, некоторое отличие есть только в синтаксисе:

```
def f(x):
    if x < 7: print("x < 7")
    else: print("x >= 7")
f(3) → x < 7
f(9) → x >= 7
```

Ветви **if** и **else** должны располагаться на разных строках. После условного выражения и после слова **else** надо ставить двоеточие. Если выражения после условия и после слова **else** записываются на новых строках, то их надо располагать с отступом:

```
def f(x):
    if x < 7:
        print("x < 7")
    else:
        print("x >= 7")
```

На месте отдельных выражений могут быть блоки кода:

```
def f(x):
    if x < 7: y = 7 - x; print(y)
    else: y = x - 7; print(y)
```

f(3) → *4*
f(9) → *2*

Блоки могут возвращать значения с помощью оператора **return**, при этом тип возвращаемых значений не обязательно одинаковый. Значит, условный оператор может возвращать результат, тип которого зависит от исходных данных:

```
def f(x, y):
        if x < y: return y - x
        else: return "x > y"
a = f(3, 5)
b = f(9, 4)
print(a) → 2
print(b) → x > y
```

Здесь **a** имеет тип **int**, а **b** – тип **string**.

Ветвь **else** может отсутствовать и тогда, если условие не выполняется, условный оператор возвращает значение **None**. Это единственное значение для типа **NoneType**:

```
def f(x):
        if x < 7: y = 7 - x; return y
print(f(3)) → 4
print(f(9)) →None
print(type(f(9))) → <class 'NoneType'>
```

Условный оператор может иметь больше двух ветвей. В этом случае для промежуточной ветви применяется слово **elif**:

```
def f(x,y):
   if x < y: print('x меньше y')
   elif x > y: print('x больше y')
   else: print('x равно y ')
f(3, 5) →x меньше y
f(5, 3) →x больше y
f(3, 3) →x равно y
```

Можно также применять вложенные условные операторы, если это больше нравится:

```
def f(x,y):
   if x == y: print('x равно y')
   else:
      if x < y: print('x меньше y')
      else: print('x больше y')
f(3, 5) →x меньше y
f(5, 3) →x больше y
```

f(3, 3) → *x равно y*

Есть ещё одна разновидность оператора **if-else**, когда выражение стоит перед словом **if**:

```
def f(x, y):
    z = x - y if x > y else y - x
    return z
```

print(f(7, 3)) → *4*
print(f(2, 7)) → *5*

Здесь не требуется ставить двоеточие после слова **else**. Можно было не применять локальную переменную **z**, а сделать так:

```
def f(x, y):
    return x - y if x > y else y - x
```

print(f(7, 3)) → *4*
print(f(2, 7)) → *5*

Ветвь **else** в этом случае необходима и такой вариант недопустим:

```
z = x - y if x > y
```

Или надо делать так:

```
z = x - y if x > y else None
```

Оператор **if-else** можно и вообще не применять, а использовать только логические операторы **and** и **or**:

```
def f(x, y):
    return ((x<y) and 'меньше') or 'больше, равно'
```

print(f(3, 7)) → *меньше*

Для работы с элементами коллекций Python имеет обычный для современных языков цикл **for** с использованием ключевого слова **in**:

```
m = ['Люда', 'Люба', 'Катя', 'Лена']
for x in m:
  print("С Новым годом, {}!".format(x))
```

Получим:

С Новым годом, Люда!
С Новым годом, Люба!
С Новым годом, Катя!
С Новым годом, Лена!

Значит, после идентификатора коллекции должно быть двоеточие, после которого располагается тело цикла.

В цикле **for** можно использовать ранги:

for x in range(1, 7, 2): print(x) → *1 3 5*

Если кроме значений элементов коллекции надо получить ещё и их индексы, надо применить функцию **enumerate**, которая

генерирует кортежи (пары) из индекса и значения. Затем кортежи циклом for распаковываются в индексы (**i**) и значения (**x**):

```
m = ['one', 'two', 'three', 'four']
for i, x in enumerate(m):print(i, '::', x)
```

Получим на консоли:

0 :: one
1 :: two
2 :: three
3 :: four

Обратите внимание на список значений, переданных оператору **print**. В этом списке содержатся идентификаторы переменных и текстовая константа «вперемжку». Оператор **print** всё приводит к типу **string** автоматически без применения функции **format** для интерполяции.

Подобным образом оператор цикла **for-in** работает со списками, рангами, кортежами, множествами и строками. Применение оператора **for-in** для хешей просто покажу на примерах.

В таком варианте будут извлечены ключи:

```
    h = { 'Люда' : 25, 'Лена' : 19, 'Катя' : 21 }
for x in h: print(x,'', end = '') → Люда Лена Катя
```

А в таком — значения:

```
h = { 'Люда' : 25, 'Лена' : 19, 'Катя' : 21 }
for i in h: print(h[i],'', end = '') → 25 19 21
```

Можно, конечно, вывести на консоль ключи и значения парами:

```
h = { 'Люда' : 25, 'Лена' : 19, 'Катя' : 21 }
for i in h: print(i, h[i])
```

Получим:

Люда 25
Лена 19
Катя 21

При создании списка перед оператором **for** может быть выражение, вычисляющее элементы списка:

```
m = [x**2 for x in range(3, 7)]
print(m) → [9, 16, 25, 36]
```

Циклы **for** могут быть вложенными:

```
m = [x + y for x in 'abc' for y in 'lmn']
print(m) → ['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Значит, выражение перед оператором **for** может использовать переменные цикла как основного, так и вложенных операторов.

Оператор цикла **while** имеет следующий синтаксис:

```
x = 0
while x < 5:
    print(x * x, end = ", ") →0, 1, 4, 9, 16,
    x += 1
```

Тело цикла выполняется повторно до тех пор, пока условие возвращает значение **true**. Если вместо условного выражения поставить значение **true**, цикл будет бесконечным.

Оператор **break** позволяет прервать цикл на любой итерации. Например, требуется найти число, которое нацело делится на **9** и при делении на **7** даёт целую часть, равную **25**:

```
x = 0
while x < 300:
    if x % 9 == 0 and x // 7 == 25 : print(x); break
    x +=1
```

Получим: *180* .

Здесь при нахождении нужного числа выходим из цикла, так как дальнейшие итерации не имеют смысла.

Оператор **continue** прерывает выполнение очередной итерации и выполняет возврат на начало цикла. Например, исключим из ранга **0..20** числа, нацело делящиеся на **2**, **3** и **7**:

```
for x in range(0, 20):
    if x % 2 == 0 or x % 3 == 0 or x % 7 == 0: continue
    print(x, end = ", ") → 1, 5, 11, 13, 17, 19,
```

Операторы **break** и **continue** применяются всегда в теле циклов **while** или **for**. Если эти операторы находятся во вложенном цикле, то выход происходит только из внутреннего цикла, а внешний цикл будет продолжаться. Операторы для выхода сразу из вложенного и внешнего цикла отсутствуют.

Если цикл находится в теле функции, выход из цикла может быть с помощью оператора **return**:

```
def f(m):
    for x in m:
        if (x % 2 != 0): return x
    return 10
print(f(range(4, 10))) → 5
```

Здесь происходит выход из цикла и из функции при получении первого нечётного числа из ранга **4..10**. Операция **return 10** будет выполнена, если только функции **f** передать список, в котором нет нечётных чисел. Если в теле функции есть вложенные циклы, то

выход из внутреннего цикла по оператору **return** также приводит и к выходу из функции.

Иногда бесконечный цикл бывает полезен, например:

```
while True:
    x = input('Введите текст:')
    if x == 'stop': break
    try: print(float(x) ** 2)
    except: print('Ошибка, попробуйте снова!')
print('Пока!')
```

Здесь использованы операторы **try - except**, позволяющие перехватывать ошибки (исключения). Если в блоке оператора **try** будет зафиксирована ошибка (исключение), то будет выполнен блок оператора **except**, в котором эту ошибку можно обработать и продолжить выполнение программы. При отсутствии ошибки оператор **except** просто игнорируется. В нашем примере исключение будет иметь место, если переменная **x** получит значение, которое не может быть преобразовано функцией **float**, например, если мы вместо числа введём с клавиатуры слово. Наш бесконечный цикл может быть прерван вводом слова **stop**. Подробнее об исключениях поговорим позднее.

Итераторы

Современные языки программирования используют для работы с коллекциями специальные функции, называемые итераторами. Итераторы автоматически создают цикл, в котором можно выполнять нужные операции с элементами коллекций. Язык Python также имеет несколько итераторов.

Итератор **zip** это функция, принимающая одну или более коллекций и возвращающая некий объект типа **zip**. Например, можно итератору **zip** передать два списка:

```
m1 = [1,2,3,4]
m2 = [5,6,7,8]
w = zip(m1, m2)
print(type(w)) → <class 'zip'>
print(list(w)) → [(1, 5), (2, 6), (3, 7), (4, 8)]
```

Сам объект **w** (типа **zip**) играет вспомогательную роль, но если его передать функции **list**, то получим список, членами которого будут кортежи — пары, составленные из элементов исходных списков.

Такие списки удобны для работы с элементами исходных коллекций. Например можно их попарно суммировать в цикле **for** :

m1 = [1,2,3,4]
m2 = [5,6,7,8]
m3 = list(zip(m1, m2))
print(m3) → *[(1, 5), (2, 6), (3, 7), (4, 8)]*
for (x, y) in m3: print(x, y, '--', x+y)

Получим:

1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 — 12

Вместо функции **list** можно использовать функции **tuple** или **set** и тогда результат будет представлен кортежем или множеством соответственно.

Если итератору **zip** передать один список, то получим :

m = [1,2,3,4]
m1 = list(zip(m))
print(m1) → *[(1,), (2,), (3,), (4,)]*
print(type(m1[0])) → *<class 'tuple'>*

Здесь элементы списка **m1** тоже кортежи, имеющие один элемент.

Итератор **zip** может принимать также три или более коллекций, соответственно членами итогового списка будут кортежи с тремя и более членами. Аргументами итератора **zip** могут быть и коллекции других типов. Например, это могут быть кортежи:

m1 = (1,2,3,4)
m2 = (5,6,7,8)
m3 = list(zip(m1, m2))
print(m3) → *[(1, 5), (2, 6), (3, 7), (4, 8)]*

Но если итератору передавать множества, то может быть изменён порядок расположения элементов:

m1 = {1,2,3,4}
m2 = {5,6,7,8}
m3 = list(zip(m1, m2))
print(m3) → *[(1, 8), (2, 5), (3, 6), (4, 7)*

Если использовать хеши, в результирующем списке останутся только ключи:

m1 = {'aa' : 1,'bb' : 2,'cc' : 3}
m2 = {'dd' : 4,'ee' : 5,'ff' : 6}
m3 = list(zip(m1, m2))

print(m3) → *[('aa', 'dd'), ('bb', 'ee'), ('cc', 'ff')]*

С помощью цикла **for** и итератора **zip** легко сформировать хеш из двух заданных списков:

```
m1 = ['aa', 'bb','cc']
m2 = [1, 2, 3]
m3 =  {k: v for (k, v) in zip(m1, m2)}
```
print(m3) → *{'aa': 1, 'bb': 2, 'cc': 3}*

Можно также применить функцию **dict**:

```
m1 = ['spam', 'eggs', 'toast']
m2 = [1, 3, 5]
h = dict(zip(m1, m2))
```
print(h) → *{'spam': 1, 'eggs': 3, 'toast': 5}*

Итератору **zip** можно предавать коллекции разной длины, при этом лишние элементы более длинных коллекций будут игнорироваться.

Объект типа **zip** может, кроме того, быть передан функции **next**, которая при повторных вызовах будет возвращать элементы итоговой коллекции, например:

```
m1 = [1,2,3,4]
m2 = [5,6,7,8]
w = zip(m1, m2)
```
print(next(w)) →*(1, 5)*
print(next(w)) →*(2, 6)*
print(next(w)) → *(3, 7)*

Итератор **map** принимает одну коллекцию и функцию, с помощью которой над элементами исходной коллекции могут быть выполнены нужные операции. Этот итератор тоже возвращает объект, который теперь будет иметь тип **map**, и с помощью функций **list**, **tuple**, или **set** можно получить коллекцию нужного вида. Например:

```
import math
m = [4, 9, 16,25]
s = map(math.sqrt, m)
```
print(list(s)) → *[2.0, 3.0, 4.0, 5.0]*

Ещё один пример:

```
m = map(len, ['Ленинград', 'Сталинград', 'Севастополь', 'Одесса'])
```
print(list(m)) → *[9, 10, 11, 6]*

Здесь функция **len** определяет длину слов, представляющих элементы списка.

Вместо стандартной могут применяться собственные функции:

```
m = [2,3,4,5]
def f(x): return x**2
```

s = map(f, m)
print(list(s)) → *[4, 9, 16, 25]*

Очень удобно передавать итератору безымянную функцию:

m = [2,3,4,5]
s = map(lambda x: x2, m)**
print(list(s)) → *[4, 9, 16, 25]*

Объект типа **map** также может использоваться функцией **nex**t:

m = [2,3,4,5]
def f(x): return x2**
s = map(f, m)
print(next(s)) → *4*
print(next(s)) → *9*
print(next(s)) → *16*

Ещё один итератор **iter** также принимает коллекцию и при передаче ему списка возвращает объект типа **list_iterator**. При передаче его функции **next** будем получать элементы исходного списка:

m = [1, 2, 3]
i = iter(m)
print(type(i)) → *<class 'list_iterator'>*
print(next(i)) → *1*
print(next(i)) → *2*

Если итератору **iter** передать кортеж, получим объект типа **tuple_iterator**:

m = (1, 2, 3)
i = iter(m)
print(type(i)) → *<class 'tuple_iterator'>*
print(next(i)) → *1*
print(next(i)) → *2*

При использовании множества объект будет иметь тип **set_iterator**, а для хеша получим тип **dict_keyiterator**. Слово **key** здесь означает, что функция **next** в данном случае будет возвращать только ключи.

Итератор **filter** позволяет выбирать из коллекции элементы, удовлетворяющие заданному условию, например:

m = [3, 8, 4, 5,9]
s = filter(lambda t: t > 5, m)
print(type(s)) →*<class 'filter'>*
print(list(s)) → *[8, 9]*

Объектно-ориентированное программирование

Python позволяет программировать в объектно-ориентированном стиле, но при этом применяется своеобразный подход. С техническими деталями и синтаксисом лучше всего познакомиться на примерах.

Создадим для начала пустой класс, например, под названием *Prob*:

class Prob: pass
p = Prob()
print(type(p)) → *<class '__main__.Prob'>*
p.x = 'Hello'
print(p.x) → *Hello*
p.f = lambda x: x2**
print(p.f(3)) → *9*
def g(x, y): return x + y
p.f = g
print(p.f(2, 3)) → *5*

По традиции название класса обычно пишут с заглавной буквы, но здесь это необязательно. Python вообще, за некоторыми исключениями, безразличен в отношении регистра первой буквы в идентификаторах. После названия класса ставится двоеточие, за которым располагается тело класса с соблюдением отступов точно также, как и для функций. Если тело состоит из одного выражения, то его можно поместить в той же строке, где и название. В нашем примере тело класса представлено одним оператором **pass**, который ничего не делает. Этот оператор бывает полезен при отладке программ. Оператор **pass** приходится применять потому, что в теле класса обязательно должно быть хоть что-нибудь. После названия класса пустые скобки необязательны, хотя и допустимы.

Хотя класс и пустой, но он позволяет создавать экземпляры (объекты) по обычному правилу. Пустые скобки тут лучше применять, так как могут быть нюансы. Здесь мы создаём экземпляр класса **p**. Как видим, его тип **__main__.Prob**. Здесь**__main__** это встроенный атрибут, о которых поговорим позже. Можно считать, что здесь атрибут указывает на то, что тип **Prob** пользовательский. Пока всё, как обычно, но дальше мы встречаем нечто неожиданное. Оказывается, экземпляр класса позволяет создавать переменные

экземпляра (поля) (**x**) и методы экземпляра (свойства) (**f**), которые можно изменять.

Попробуем объявить переменную и функцию в теле класса

```
class Prob:
        x = 22
        def f(y): return y*y
p = Prob()
print(Prob.x) → 22
p.x = 33
print(p.x) → 33
print(Prob.f(3)) → 9
```

Переменная **x** доступна по чтению и по записи как по экземпляру, так и по классу, а функция **f** доступна только по классу. Получается, что, как это обычно принято, **x** можно считать одновременно переменной экземпляра и переменной класса, а **f** это метод класса, доступный только по названию класса. Полноценную переменную экземпляра можно создать только с применением методов доступа (или через конструктор), например:

```
class Prob:
        def f(self, x): self.y = x
        def g(self): return self.y
p = Prob()
p.f(123)
print(p.g()) → 123
```

Здесь **self** это атрибут. На F# применяется нечто подобное и там это называется собственный идентификатор. Функция **f** обеспечивает доступ к переменной экземпляра **y** по записи, а **g** — по чтению. Слово **self** (свой) применяется только по традиции, обозначать атрибут можно как угодно. Если вставить строку: **print(self)** , то будет выведено:*<__main__.Prob object at 0x00000255B6185D00>*. Значит атрибут это ссылка на объект класса. Обратите внимание на синтаксис: в списке аргументов атрибут отделяется от переменной запятой, а в теле функции — точкой. Дальше для краткости я буду обозначать атрибут одной буквой, чаще всего **s**.

Если функция экземпляра не имеет аргументов, то в списке аргументов указывается один только атрибут:

```
def g(s): return s.y
```

Для доступа по записи необязательно вводить новый идентификатор, можно воспользоваться тем же; конфликта имён не будет:

```
class Prob:
        def f(s, x): s.x = x
        def g(t): print(t.x)
p = Prob()
r = Prob()
p.f("Солнце")
r.f(3.14159)
p.g() → Солнце
r.g() → 3.14159
```

В экземплярах класса можно задавать разные значения любых типов переменной экземпляра.

Как мы уже знаем, кроме того, можно для экземпляра создавать новые переменные, например мы можем добавить:

```
p.y = 777
```

Таким образом, если в классе объявить метод, имеющий аргумент с атрибутом, то этот метод будет доступен только по имени экземпляра и недоступен по имени класса. Иначе говоря, функция с атрибутом является методом экземпляра, а без атрибута — методом класса.

При числе аргументов метода больше одного атрибут должен находиться на первом месте в списке:

```
class Prob:
        def f(s, x, y): return x + y
p = Prob()
print(p.f(2, 3)) → 5
```

Аргументы класса в Python можно применить только с помощью встроенной функции **__init__**, которая фактически выполняет роль конструктора класса. Функции с идентификаторами, которые начинаются и заканчиваются двумя знаками подчёркивания, обладают особыми свойствами. В документации их тоже называют атрибутами, но чтобы не путаться в терминах, будем дальше их называть функциями-атрибутами. Несмотря на то, что это встроенные функции, их надо применять со служебным словом **def**. Рассмотрим их немного позже.

Посмотрим на пример применения конструктора:

```
class Prob:
        def __init__(s, x, y):
                s.a = x
                s.b = y
p = Prob('Люся', 17)
```

print(p.a, p.b) → *Люся 17*

В строках **s.a = x** и **s.b = y** создаются переменные экземпляра класса **a** и **b** (привязываются к экземпляру класса через атрибут **s**). Похожий способ применения аргументов класса используется и в других языках программирования, только вместо атрибута обычно применяют служебное слово **this**. При создании экземпляра класса функция-атрибут **__init__** вызывается автоматически и ей передаются аргументы класса.

Нет никакой необходимости вводить новые идентификаторы для переменных экземпляра, лучше всегда применять те же:

```
class prob:
    def __init__(s, x, y):
        s.x = x
        s.y = y
p = prob('Люся', 17)
print(p.x, p.y)
```

→ *Люся 17*

Аргументы класса доступны в теле класса с обязательным использованием атрибута:

```
class Prob:
    def __init__(s, x, y):
        s.x = x
        s.y = y
    def f(s, z): return z * (s.x + s.y)
p = Prob(2, 3)
print(p.f(5))
```

→ *25*

Давайте попробуем что-нибудь похожее на фрагмент из реальности:

```
class Product:
    def __init__(s, denomination, price):
        s.denomination = denomination
        s.price = price
    def cost(s, weight): return weight * s.price
p = Product('картошки', 3)
print('стоимость', p.denomination, 'равна', p.cost(10),'рублям')
r = Product('репы', 5)
print('стоимость', r.denomination, 'равна', r.cost(5),'рублям')
```

Получим на консоли такой результат:

стоимость картошки равна 30 рублям
стоимость репы равна 25 рублям

Не слишком элегантно, но в общем-то вполне приемлемо.

Далее пример использования переменной класса:

```
class Prob:
        x = 5
        def __init__(s, y): s.y = y
        def f(s, z): return z * (Prob.x + s.y)
p = Prob(3)
print(p.f(2)) → 16
Prob.x = 7
p = Prob(3)
print(p.f(2)) → 20
```

Фактически равнозначно объявление переменной класса в теле класса или вне класса:

```
class Prob:
        def __init__(s, y): s.y = y
        def f(s, z): return z * (Prob.x + s.y)
p = Prob(3)
Prob.x = 5
print(p.f(2))→ 16
Prob.x = 7
p = Prob(3)
print(p.f(2)) → 20
```

Обратите внимание: мы создали экземпляр класса **p** когда переменная класса **x** ещё не была объявлена, хотя она использована в теле метода **f** экземпляра класса.

Наследование классов в Python также реализуется своеобразно. Посмотрим на примере в котором объявлены два класса: **Rect** (прямоугольник) и **Sq** (квадрат). Поскольку квадрат это частный вариант прямоугольника, можно не создавать новых методов для квадрата, а использовать методы для прямоугольника. Для этого надо сделать класс **Sq** дочерним по отношению к классу **Rect**. На Python это будет выглядеть приблизительно так:

```
class Rect():
        def __init__(s, x, y):
                s.x = x
                s.y = y
        def a(s): return s.x * s.y
        def p(s): return 2 * (s.x + s.y)
r = Rect(3, 2)
print(r.a()) → 6
print(r.p()) → 10
```

```
class Sq(Rect):
        def __init__(s, x):
                super().__init__(x, x)
r = Sq(5)
print(r.a()) → 25
print(r.p()) → 20
```

Значит, чтобы класс **Sq** наследовал классу **Rect** надо название родительского класса указать в круглых скобках после названия дочернего класса, подобно аргументу для функции. Затем в теле функции-конструктора дочернего класса надо вызвать функцию-конструктор родительского класса, применив слово **super** (пустые скобки обязательны) и передать этому конструктору аргументы (в данном случае длины сторон прямоугольника, которые для квадрата одинаковы). После этого в экземплярах дочернего класса будут доступны все методы и свойства родительского класса (в нашем случае методы для вычисления площади **a** и периметра **p** прямоугольника). Обратите внимание: при передаче аргументов конструктору родительского класса не надо указывать их атрибуты (**__init__(x, x)** вместо **__init__(s.x, s.x)**).

Если аргументов у родительского класса нет, и явный вызов его конструктора не требуется, то наследование выполняется совсем просто:

```
class A:
        def f(s, x): s.x = x
        def g(s): print(s.x)
class B(A): pass
p = B()
p.f(77)
p.g() → 77
```

Класс может наследовать нескольким родительским классам:

```
class A:
        def f(s,x): s.x = x; return 2 * s.x
class B:
        def g(s, y): s.y = y; return 3 * s.y
class C(A, B): pass
p = C()
print(p.f(3), p.g(3)) → 6 9
```

Python позволяет добавлять методы в экземпляры классов. Этим свойством можно иногда использовать имеющийся класс без механизма наследования. Вот как это можно делать:

```
class A:
        def f(s,x): s.x = x; return 2 * s.x
def g(s, y): s.y = y; return s.x + s.y
A.g = g
p = A()
print(p.f(3), p.g(4)) → 6 7
```

Строка **A.g = g** добавляет метод **g** во все создаваемые экземпляры класса **A** и этот метод можно применять точно также, как и методы, объявленные в самом классе **A**. Каким-то образом функция **g**, объявленная вне класса, знает, что в классе **A** в теле метода **f** есть переменная **x**, которую можно использовать. Таким же способом можно добавлять в класс не только методы, но и переменные.

Впрочем, в документации этот стиль программирования не считается хорошим.

Аргументы класса могут быть заданы по умолчанию (в конструкторе) точно так же, как это делается для функций:

```
class Prob:
        def __init__(s, x, y, z = 4):
                s.x = x; s.y = y; s.z = z
        def f(s): return s.x + s.y + s.z
p = Prob(2, 3)
print(p.f()) → 9
p = Prob(2, 3, 5)
print(p.f()) → 10
```

Аргументы классов также можно применять, как именованные, располагая их при передаче значений в произвольном порядке:

```
p = Prob(y = 3, x = 2)
```

Встроенная функция-атрибут **__del__** (смотрите далее) позволяет отменить экземпляр класса. Обычно эту операцию называют деконструктором. Например:

```
class A:
        def __init__(s): s.x = 0; print('Это конструктор')
        def f(s) : s.x = s.x + 1; print('x = ',s.x)
        def __del__(s): print('Это деконструктор, x = ', s.x)
p = A() →Это конструктор
p.f() → x =  1
p.f() → x =  2
p = 42 → Это деконструктор, x =  2
print('p = ',p) → p =  42
```

Функция-атрибут **__del__** вызывается автоматически, когда экземпляр класса используется как операнд или как переменная. В данном случае эта функция была вызвана при инициализации переменной **p**.

Функции-атрибуты

Python располагает богатым набором функций-атрибутов. Идентификаторы таких функций начинаются и заканчиваются двумя знаками нижней черты. В частности, с функцией **__init__**, выполняющей роль конструктора классов, мы уже познакомились. В документации такие функции называют «магическими» в связи с их нестандартным поведением. (Эти функции называют также иногда «Dunder-методами» от слов double-underscope - двойное подчёркивание). Кроме того, эти функции называют также скрытыми или специальными внутренними методами, которые каждый объект использует для взаимодействия с другими объектами.

Функции-атрибуты позволяют задавать поведение классов в том случае, когда экземпляры этих классов используются как операнды в выражениях. Например, функция-атрибут **__repr__** позволяет в самом классе организовать вывод результатов в нужном формате с тем, чтобы не повторять этот код при выводе для каждого экземпляра класса. В следующем примере эта функция вызывается, когда экземпляр класса (**p**) передаётся оператору **print**.

```
class A:
        def __init__(s, x, y, z = 0): s.x = x; s.y = y; s.z = z
        def f(s): s.z = s.x + s.y
        def __repr__(s): return 'Результат: %s' % s.z
p = A(x = 5, y = 7)
p.f(); print(p) →Результат: 12
r = A(x = 'Борис ', y = 'Уваров')
r.f(); print(r) → Результат: Борис Уваров
```

Получается, что если экземпляр класса передать оператору **print**, то на терминал будет выведено то, что возвращает в качестве результата функция **__repr__.** В примере также использован аргумент по умолчанию и применён метод экземпляра **f**, изменяющий значение одного из аргументов (**z**) класса.

Функции-атрибуты используются со служебным словом **def**, то-есть, они как бы определяются заново, что позволяет задавать их

функциональность. Фактически это механизм перегрузки (переопределения) методов и операторов.

В следующем примере использована функция-атрибут **__add__**, которая вызывается, когда экземпляр класса участвует в качестве операнда для операторов сложения и конкатенации.

```
class A:
        def __init__(s, x): s.x = x
        def f(s): print(s.x)
        def __add__(s, y): return A(s.x + y)
a = A(25)
a.f() →25
b= a + 17
b.f() → 42
c = A('abc')
d = c + 'xyz'
d.f() → abcxyz
```

Значит, если мы к экземпляру класса что-то прибавляем, то это значение передаётся аргументу функции-атрибута **__add__** и эта функция выполняется.

Покажем ещё, как с помощью нескольких функций-атрибутов можно реализовать разные операции над векторами:

```
class V:
        def __init__(s, x, y): s.x = x; s.y = y
        def __neg__(s): return V(-s.x, -s.y)
        def __add__(s, t): return V(s.x + t.x, s.y + t.y)
        def __sub__(s, t): return V(s.x - t.x, s.y - t.y)
        def __eq__(s, t): return s.x == t.x and s.y == t.y
        def __abs__(s): return (s.x*s.x + s.y*s.y)**0.5
        def __repr__(z): return 'Vector({}, {})'.format(z.x, z.y)
a = V(1, 4); b = V(2, 0)
print(-a) →Vector(-1, -4)
print(a + b) → Vector(3, 4)
print(a - b) → Vector(-1, 4)
print(a == b) → False
print(a + b == b + a) → True
print(abs(a + b)) → 5.0
```

Функция-атрибут **__call__** позволяет передавать экземпляру класса значение, которое на самом деле будет передано этой **__call__**:

```
class A:
        def __init__(s, x): s.x = x
```

```
    def __call__(s, y): return s.x + y
p = A(2)
print(p(1)) → 3
print(p(2)) → 4
r = A('Hello, ')
print(r('World!')) → Hello, World!
```

Если выполнить следующую программу:

```
class A: pass
print(dir(A))
```

то с помощью функции **dir** получим список функций-атрибутов, которые по умолчанию присоединены к пустому классу **A** на правах его членов:

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']

Этот список представляет лишь малую часть всего набора таких функций в языке Python. Трудно сказать, насколько все эти функции повышают качество программного кода, но то, что они и в самом деле придают программе некие «магические» свойства, несомненно.

Модули и документирование

Модули в Python создаются очень просто. Для этого достаточно поместить нужный программный код в отдельный файл, имеющий в названии расширение **.py**. Этот файл далее можно импортировать с помощью оператора **import** и использовать в свой программе. Например, создадим такой модуль **mod.py**:

```
def f(x, y): z = x // y; r = x - z * y; return z, r
```

Здесь функция **f** вычисляет целую часть и остаток при делении двух чисел. Затем напишем программу **prog.p**y, использующую этот модуль:

```
import mod
a, b = mod.f(23, 4)
print(a, b) → 5 3
```

Импортируя модуль, получаем доступ ко всем его членам: к переменным, к функциям, к классам и так далее. Применив ключевое слово **from**, можно импортировать только один нужный член:

```
from mod import f
```

a,b = f(23, 4)

Теперь префикс **mod** больше не нужен. Используя звёздочку, импортируем все члены, и их можно применять также без префикса:

from mod import *
a,b = f(23, 4)

Импортируемый модуль должен находиться в той же папке, где и вызывающая программа, или в корневой папке *Python312*, или в любой другой папке, абсолютный адрес которой содержится в переменной *path* «переменных сред». Можно также использовать полные (квалифицированные) названия модулей. Как это конкретно делается, можно найти в документации.

Оператор **del** позволяет отказаться от импортированного ранее модуля:

import mod
a, b = mod.f(23, 4)
print(a, b)
del mod
a, b = mod .f(33, 5)

Здесь второй вызов функции **f** приведёт к ошибке; получим сообщение о том, что имя **mod** не определено.

В модуле **inspect** есть функция **getsource**, которая позволяет получить содержимое всего модуля

import mod
import inspect
print(inspect.getsource(mod)) →
*def f(x, y): z = x // y; r = x - z * y; return z, r*

Модуль **mod** представлен одной только функцией **f**.

Кроме комментариев в программах можно размещать текстовую информацию, в которой объясняется назначение, содержание и возможности модулей, функций, классов и так далее. Обычно такую информацию называют документированием программы. Документирование может быть полезным при программировании, но для этого должна быть возможность извлечения этой информации. На Python это делается очень просто. Покажем на примере. Пусть модуль **mod** содержит такой код:

"""Module documentation:
информация о модуле prog"""
def f(x):
"""Function documentation:
информация о функции f"""

```
    return x ** 2
class A:
    "Информация о классе A"
    pass
```

Свободный текст на уровне модуля, или в теле функций и классов здесь и представляет документирование. Свободным является любой текст в кавычках, не являющийся значением переменной, не используемый оператором **print** и так далее. Для текста на нескольких строках применяются тройные кавычки, а на одной строке двойные или одинарные. Для извлечения документирования можно применять функцию-атрибут **__doc__**. Получим такие результаты:

```
import mod
print(mod.__doc__) →
```
Module documentation:
    информация о модуле mod
```
print(mod.f.__doc__) →
```
Function documentation:
    информация о функции f

print(mod.A.__doc__) → *Информация о классе A*

В модуле **inspect** есть также функция **getdoc**, которая делает то же самое:

```
import inspect
print(inspect.getdoc(mod.f)) →
```
Function documentation:
информация о функции f

Исходный код для встроенных функций и некоторых стандартных модулей (например **math**) написан на **C** и документирование из них можно извлечь только с привлечением средств языка **C**, но это уже другая тема.

Кроме этих средств документирования есть ещё встроенная функция **help**, которая позволяет получить полную информацию о модуле и его членах. Допустим, что модуль **mod** имеет такой код:

```
"""Module documentation:
    информация о модуле mod"""
class A:
    "Информация о классе A"
    def __init__(s, x): s.x = x
    def f(s):
```

```
        'Информация о методе f экземпляра класса A'
        return s.x**3
```

Если теперь выполнить такую программу:

```
import mod
p = mod.A(3)
print(p.f())
help(mod.A)
```

то получим следующий результат

27
Help on class A in module mod:
class A(builtins.object)
| A(x)
| Информация о классе A
| Methods defined here:
| __init__(s, x)
| Initialize self. See help(type(self)) for accurate signature.
| f(s)
| Информация о методе f экземпляра класса

Тут всё ясно без слов!

Декораторы

Функции в Python являются объектами и как и, всякие другие объекты, могут служить аргументами для других функций, а также возвращаться другими функциями в качестве результата. Рассмотрим пример, в котором использованы эти свойства функций:

```
def f(p):
    def r(): return p() + 'Мир!'
    return r
def g(): return 'Привет, '
g = f(g)
print(g()) → Привет, Мир!
```

Здесь объявлена некая исходная функция **g**, которая только возвращает текст **'Привет, '**. Эта функция передана функции **f** в качестве значения для аргумента **p**. В теле функции **f** объявлена функция **r**, которая к результату функции **p** добавляет текст **'Мир!'**. Затем функция **r** возвращается как результат функции **f**. В строке **g = f(g)** исходная функция **g** заменяется результатом функции **f**, а именно функцией **r**, поскольку для новой функции мы применили

старое название **g**. Теперь при вызове новой функции **g** вместо текста *Привет,* получим текст *Привет, Мир!* Значит, функция **f** позволяет изменять переданную ей функцию. Саму эту операцию изменения функции называют обёрткой или чаще декорированием (украшением) заданной (декорируемой) функции, а функцию **f** называют декоратором. При этом сама исходная функция
def g(): return 'Привет, ' не изменяется, мы можем, например, её скопировать и использовать где-то в другом месте.

Конечно, можно не вводить новый идентификатор **p**, а использовать то же название **g**:

```
def f(g):
    def r(): return g() + 'Мир!'
    return r
def g(): return 'Привет, '
g = f(g)
```

Имеется “синтаксический сахар”:

```
def f(g):
    def r(): return g() + 'Мир!'
    return r
@f
def g(): return 'Привет, '
```

print(g()) → *Привет, Мир!*

Значит, вместо **g = f(g)** можно писать **@f**, а после этой строки должно располагаться объявление декорируемой функции, то-есть, должно быть слово **def**. Такой синтаксис хорош тем, что наличие строки со знаком **@** сразу указывает на использование декоратора.

Декорировать можно любые функции, включая функции из стандартных библиотек, например:

```
from math import sin
def f(p):
    def r(x):
        return(p(x) ** 2.0)
    return r
sin = f(sin)
```

print(sin(4.0)) → *0.5727500169043067*

Теперь функция **sin** вычисляет квадрат синуса.

При использовании декоратора **@f** в данном случае приходится ввести новый идентификатор для квадрата синуса:

```
from math import sin
def f(p):
```

```
    def r(x):
        return(p(x) ** 2.0)
    return r
@f
def p(x): return sin(x)
print(p(4.0)) → 0.5727500169043067
```

С таким же успехом можно декорировать методы классов:

```
def f(g):
    def r(s, x):
        x = x + ' раму'
        return g(s, x)
    return r
class A:
    def __init__(s): s.y = 'Мама '
    @f
    def g(s, x): print("Результат: {}".format(s.y + x))
p = A()
p.g('мыла ') → Результат: Мама мыла  раму
```

Здесь декорирован метод **g** экземпляра класса **A**.

В качестве декоратора можно также использовать экземпляр класса, например:

```
class A:
    def __init__(s, f):
        s.f = f
    def __call__(s, x):
        return sum(s.f(s, x))
class C:
    @A
    def g(s, m):
        return list(map(lambda t: t * t, m))
print(C.g([1, 2, 3])) → 14
```

Здесь экземпляру класса **A** передаётся в качестве аргумента декорирующая функция **f**. Декорируемым тут является метод **g** класса **C**. Функция-атрибут **__call__** вызывается автоматически при создании экземпляра класса A.

Аргументы могут присутствовать как у декорируемой функции, так и у самого декоратора. Для передачи аргументов декоратору надо использовать специальную функцию. Это можно сделать приблизительно так:

```
def f(x, y):
```

```
    def r(g):
        def p(a, b):
            print(x, y, a, b)
        return p
    return r
f1 = f("Люда", "Наташа")
def g(a, b): print(a, b)
g = f1(g)
```

g("Петя", "Серёжа") → *Люда Наташа Петя Серёжа*

Здесь генератором служит функция **r**, а функция **f** создаёт (возвращает) этот генератор, передавая ему аргументы. Вызывая функцию **f** мы получаем декоратор, с которым работаем далее как обычно. Конечно, здесь можно использовать также и «синтаксический сахар»:

```
def f(x, y):
    def r(g):
        def p(a, b):
            print(x, y, a, b)
        return p
    return r
@f("Люда", "Наташа")
def g(a, b): print(a, b)
```

g("Петя", "Серёжа") → *Люда Наташа Петя Серёжа*

В сущности декораторы не имеют ничего принципиально нового и используют только способность функций служить аргументами и возвращаемыми результатами для других функций.

Работа с файлами

Всякая информация хранится в файлах. Средства чтения из файлов и записи в них в Pyton очень просты. Пусть например в нашей рабочей папке расположен файл с названием **stihi.txt**, в котором записаны стихи Есенина:

Отговорила роща золотая
Берёзовым весёлым языком

Прежде чем работать с файлом, его надо открыть, для чего применяется оператор **open**:

```
x = open('stihi.txt')
print(x) →
```

<_io.TextIOWrapper name='stihi.txt' mode='r' encoding='cp1251'>

В переменной **x** будет записаны такие сведения: название некоей функции *TextIOWrapper* из модуля *io*, название файла *'stihi.txt'*, параметр *mode* (будем называть его словом флаг) со значением *'r'* и название кодировки *'cp1251'*. Чтобы из переменной **x** извлечь сам текст можно применить несколько способов. Например, функция **list** возвратит текст в форме списка:

```
x = open('stihi.txt')
y = list(x)
print(y) →
```

['Отговорила роща золотая\n', 'Берёзовым весёлым языком\n']

Каждый элемент списка **y** представляет одну строку текста.

Для распаковки списка есть уже знакомая нам функция **join**:

```
print(''.join(y)) →
```

Отговорила роща золотая
Берёзовым весёлым языком

Можно использовать цикл:

```
x = open('stihi.txt')
for y in x: print(y, end = '')
```

Здесь **end = ''** нужна потому, что без этого оператор **print** будет выводить перевод строки, а поскольку и в самом тексте есть перевод строки, то получим два перевода строки.

Флаг **'r'** указывает на то, что файл надо открыть для чтения. Это значение принято по умолчанию, но его можно указать и явно:

```
x = open('stihi.txt, 'r')
```

Проще всего прочитать файл можно с помощью функции **read**:

```
x = open('stihi.txt', 'r')
y = x.read()
print(y) →
```

Отговорила роща золотая
Берёзовым весёлым языком

Функции **read** можно передать целое число, задающее количество прочитанных символов:

```
x = open('stihi.txt', 'r')
y = x.read(13)
print(y) → Отговорила ро
```

Можно читать текст порциями:

```
x = open('stihi.txt', 'r')
y = x.read(9)
```

print(y) → *Отговорил*
y = x.read(17)
print(y) →*а роща золотая*
Бе
y = x.read()
print(y) → *рёзовым весёлым языком*

Для записи информации файл надо открыть с флагом '**w**'. Сама запись выполняется с помощью функции **write**:

```
x = open('text.txt', 'w')
s = 'Hello, World!'
x.write(s)
x = open('text.txt', 'r')
print(x.read()) →Hello, World!
x.close()
```

Здесь мы записали текст в файл, а потом прочитали. Если запись выполняется в существующий файл, прежнее содержание стирается. Если файл с заданным названием не существует, как в нашем примере, создаётся новый файл.

После окончания работы с файлом его обязательно нужно закрыть с помощью метода **close**:

```
x.close()
```

То-есть, методу **close** передаётся не название файла, а идентификатор переменной, использованной при его открытии.

Можно прочитать даже код самой программы, работающей с файлом:

```
x = open('prog.py')
print(x.read()) →
                    x = open('prog.py')
                    print(x.read())
                    x.close()
x.close()
```

Кроме флагов '**r**' и '**w**' есть и другие:

Флаг '**x**' **—** открытие файла на запись при несуществующем файле. В отличие от флага '**w**' при попытке записи в существующий файл получим исключение (сообщение об ошибке).

Флаг '**a**' — открытие файла на дозапись (добавление текста в конец существующего файла). Если файл не существует, создаётся новый файл.

Флаг '**b**' **—** открытие файла для работы с информацией в двоичном режиме.

Флаг '**t**' - открытие файла для работы с информацией в текстовом режиме.

Флаги могут комбинироваться попарно. Например '**rb**' означает открытие на чтение в двоичном режиме, а '**wt**' — на запись в текстовом режиме.

Конечно, допустимо применение как относительного, так и абсолютного (полного) имени файла.

Давайте теперь заставим программу спрашивать, какой файл надо открыть:

```
name = input('Ввдите имя файла: ')
x = open(name)
print('Содержимое файла', name, ':\n', x.read())
```

Получим например такой диалог:

Введите имя файла: stihi.txt
Содержимое файла stihi.txt :
Отговорила роща золотая
Берёзовым весёлым языком

Регулярные выражения

Как и другие современные языки, Python позволяет использовать регулярные выражения при работе с текстами. Нужные для этого функции находятся в модуле **re**.

Для поиска заданного текста в файле применяется функция **search**:

```
import re
s = open('stihi.txt')
for x in s:
    if re.search('зов', x): print(x)
```

→ *Берёзовым весёлым языком*

Функция **search** принимает два аргумента: заданный текст (**'зов'**) и строку (**x**), в которой этот текст отыскивается. Функция возвращает **True**, если заданный текст присутствует в строке. Эта программа выведет все строки файла, в которых встретится заданный текст. Тот текст, который задаётся для поиска, обычно и называют регулярным выражением (не очень ясно, почему). Условимся дальше регулярные выражения для краткости называть шаблонами. Имеется несколько видов таких шаблонов. Например шаблон в виде двух знаков с расположенными между ними точками означает кусок текста, ограниченного этими знаками, с произвольными знаками между ними в количестве, равном числу точек. Например:

```
iimport re
```

s = open('stihi.txt')
for x in s:
if re.search('в….е', x): print(x) → *Берёзовым весёлым языком*

Пробел тоже надо отмечать точкой.

Комбинация знаков **.+** соответствует произвольному числу букв, поэтому предыдущий шаблон можно заменить таким:

re.search('в.+е', x)

Шаблон, подобный этому, удобен например для поиска электронного адреса, начинающегося с заданной буквы:

re.search('b.+@', x)

Если впереди шаблона стоит угловой апостроф (^), искомый текст должен быть вначале строки. Шаблон в виде **'^ф….д'** соответствует строке, которая начинается с буквы **ф**, далее имеет четыре произвольных знака, а за ними должна следовать буква **д**. Например:

import re
s = open('stihi.txt')
for x in s:
if re.search('^O..o', x): print(x) → *Отговорила роща золотая*

В следующем примере использована функция **findall** с шаблоном **r'\S+в\S+'**, которая позволяет найти отдельные слова текста, соответствующих шаблону. Под словами здесь подразумевается набор символов, ограниченных пробелами.

import re
x = open('stihi.txt')
y = ''.join(list(x))
r = re.findall(r'\S+в\S+', y)
print(r) → *['Отговорила', 'Берёзовым']*

Шаблону **r'\S+в\S+'** соответствуют слова, содержащие букву **в** внутри слова (не на краю). Буква **r** вначале нужна для того, чтобы отменить экранирующее качество обратной косой черты внутри текста. Без буквы **r** сочетание символов **\S** не имеет смысла и будет выдано сообщение о синтаксической ошибке. Выбранные слова представлены в виде списка.

На месте одной буквы в можно применять комбинацию букв. Так шаблон в виде **r'\S+лы\S+'** соответствует результату *['весёлым']*. Можно применять различные комбинации. Например шаблон **r'в\S+'** выдаст такой результат: *['ворила', 'вым', 'весёлым']*. Набор знаков в квадратных скобках означает выбор по каждому знаку. Например шаблон **r'\S+[вл]\S+'** приведёт к такому результату:
['Отговорила', 'золотая', 'Берёзовым', 'весёлым']

С помощью тире можно задавать диапазон букв или цифр. Так шаблон **r'\S+[и-л]\S+'** даст такой результат:
['Отговорила', 'золотая', 'весёлым', 'языком']

Вместо знака + можно применять звёздочку *****. Разницу покажем на примере: при шаблоне **r'\S+a\S+'** получим *['золотая']*, а шаблон **r'\S*a\S*'** даст *['Отговорила', 'роща', 'золотая']*. Значит, если знак **+** исключает буквы на краях слова, то знак ***** их не исключает.

Известно, что в русском языке нет слов, начинающихся с буквы «а», кроме заимствованных из других языков. Посмотрим есть ли такие слова в каком-нибудь художественном произведении, ну например в рассказе Чехова «Каштанка». Это можно сделать приблизительно так:

```
import re
x = open('каштанка.txt')
y = list(x.read().split(' '))
s = []
for z in y:
    r = re.findall('^[Aa].+', z)
    if len(r) > 0: s = s + r
print(s) →
```

['Александрыч,', 'Александрыча', 'алчности', 'Александрыч', 'Александрычем', 'Ах,', 'Александрыча,', 'Ах', 'артистку', 'артисткой?', 'Ах,', 'Арабеллы.', 'аплодисменты.', 'ахнул.', 'Александрыч']

Получили только пять слов, если не считать имён и междометий . Из них слова *алчность*, *артистка* и *аплодисменты* конечно заимствованные, а глагол «*ахнул*» создан искусственно из междометия «ах» и конечно не в счёт. Применив знак + (а не *), мы исключили из поиска предлог «а».

Такой же эксперимент над повестью Тургенева «Ася» не обнаружил ни одного слова на букву «а». Интересно, какой результат дал бы роман Толстого «Война и мир»? Может быть кто-нибудь захочет попробовать?

Исключения

Исключения (exceptions) - ещё один тип данных в Python. В основном они служат для выявления и обработки ошибок программ, иногда для реакции на некоторые события. Имеется большой список встроенных исключений. Эти исключения мы постоянно встречаем

при отладке своих программ, в которых ошибки неизбежны. Например, попробуем выполнить такую программу:

```
def f(x, y): z = x/y
print(f(5, 0))
```

Вместо результата на консоль будет выведено:

```
Traceback (most recent call last):
  File "C:\Python312\MyPython\prog.py", line 2, in <module>
    print(f(5, 0))
          ^^^^^^
  File "C:\Python312\MyPython\prog.py", line 1, in f
    def f(x, y): z = x/y
ZeroDivisionError: division by zero
```

Здесь *ZeroDivisionError* это название одного из встроенных исключений, которое выводит сообщение о попытке деления на ноль. Кроме того выводится название файла программы, номер строки и выражение в котором имела место ошибка. При срабатывании исключения программа останавливает работу после вывода сообщения.

Встроенное исключение можно вызвать самостоятельно, передав ему своё собственное сообщение. Для этого используется оператор **raise**, например:

```
def f(x, y):
    if y == 0:
        raise ZeroDivisionError("Деление на ноль не имеет смысла")
    return x/y
print(f(9, 2)) → 4.5
print(f(5, 0)) →
Traceback (most recent call last):
  File "C:\Python312\MyPython\prog.py", line 6, in <module>
    print(f(5, 0))
          ^^^^^^
  File "C:\Python312\MyPython\prog.py", line 3, in f
    raise ZeroDivisionError("Деление на ноль не имеет смысла")
ZeroDivisionError: Деление на ноль не имеет смысла
```

Получаем полную информацию: в шестой строке было передано недопустимое значение аргументу функции, а в третьей строке имело место исключение.

Исключение можно организовать по любому условию. Пусть например функция должна принимать только нечётные числа:

```
def f(x):
        if x % 2 != 1:
                raise ValueError("Функции f передано чётное число")
        return x + 1
print(f(5)) → 6
print(f(4)) →
Traceback (most recent call last):
  File "C:\Python312\MyPython\prog.py", line 6, in <module>
    print(f(4))
          ^^^^
  File "C:\Python312\MyPython\prog.py", line 3, in f
    raise ValueError("Функции f передано чётное число")
ValueError: Функции f передано чётное число
```

Здесь мы применили исключение **ValueError** (ошибка значения).

Иногда нам хотелось бы избежать аварийной остановки программы и при появлении исключения что-то сделать, чтобы обработать ошибку и продолжать работу программы дальше. Для этой цели имеется оператор **try — except**. Например:

```
def f(x): return x + 5
y = input('Введите значение x = ')
try: z = f(int(y))
except:
        print('Надо ввести число, а не текст')
        y = input('Снова введите значение x = ')
        z = f(int(y))
print(z)
```

Если мы введём число, то получим такой диалог:

```
Введите значение x = 7
12
```

Если по ошибке вместо числа введём текст, то диалог будет другим:

```
Введите значение x = Анна
Надо ввести число, а не текст
Снова введите значение x = 7
12
```

В блоке оператора **try** (try - пробовать) мы выполняем какой-то код. Если никакой ошибки не произойдёт, оператор **except** (исключать, возражать) со своим блоком будет просто пропущен. Если исключение произойдёт, будет выполнен блок оператора **except**, но программа при этом не остановится.

Если применить цикл, то требование сделать правильный ввод будет повторяться до тех пор, пока не введём число:

```
def f(x): return x + 5
for _ in range(10):
    y = input('Введите значение x = ')
    try: z = f(int(y))
    except:
        print('Надо ввести число, а не текст')
        continue
    break
print(z)
```

Обычно после слова **except** пишут название исключения. В данном случае исключение будет **ValueError** (ошибка значения) и строку записывают так:

```
except ValueError:
```

Но фактически это служит только для улучшения читабельности кода и на работе программы не сказывается.

Совместно с **try-except** можно применять оператор **else** блок которого выполняется только когда исключение отсутствует. Например:

```
import math
x = float(input('x = '))
try: y = math.log(x)
except:
    print('x должно быть больше 0')
else: print(y)
print('конец')
```

Если **x>0**, то логарифм будет вычислен и в блоке оператора **else** будет выведен результат:

x = 3
1.0986122886681098
конец

При **x <= 0** происходит исключение и блок **else** будет пропущен:

x = -2
x должно быть больше 0
конец

Можно также применять оператор **finally**, блок которого выполняется всегда. Например **finally** удобно применять для закрытия файла:

```
finally:
```

```
    file.close()
    print('закрыли файл.')
```

Программирование в функциональном стиле

Функциональная программа должна содержать только функции, и ничего более. При этом функции не должны иметь побочных эффектов (чистые функции). Такие функции должны принимать только исходные данные и не могут иметь каких-то параметров и переменных, которые потенциально могут быть изменены путём внешних воздействий и, таким образом, повлиять на результат. После выполнения чистая функция должна только вернуть результат и не оставлять каких-либо последствий своей работы в памяти машины. Существует многочисленная литература по теории и практике программирования в функциональном стиле, где, в частности, рассматриваются его достоинства и недостатки.

Существуют чисто функциональные языки программирования и прежде всего это прекрасный во всех отношениях язык Haskell. Другие языки хотя и не являются чисто функциональными, но также поддерживают этот стиль программирования. Python не имеет специальных средств для функционального программирования, но его возможности позволяют программировать в этом стиле. Здесь я просто приведу несколько примеров по этой теме.

Одним из основных элементов в функциональном программировании являются рекурсивные функции, позволяющие организовывать циклы. Ранее мы уже использовали рекурсию для вычисления факториала числа. Приведу здесь этот пример снова:

```
def f(n):
    if n <= 1: return 1
    else: return n * f(n-1)
print(f(5))
```
→ *120*

Функция **f** не имеет побочных эффектов кроме вывода на терминал, но без операторов ввода-вывода программы не имеют смысла и эти побочные эффекты всегда неизбежны. Алгоритм вычисления факториала здесь такой. Сначала создаются и помещаются в память все сомножители от **n** до **1**, которые после выхода из цикла перемножаются. Память можно сэкономить, применив аккумулятор. Это можно сделать например так:

```
r = 1
def f(n):
```

```
    global r
    if n <= 1: return r
    else: r = r * n; return f(n-1)
print(f(5))
```
→ *120*

В этом варианте сомножители не сохраняются в памяти, а результат накапливается в аккумуляторе **r**. Данная функция не является чистой, поскольку использована глобальная переменная **r**, начальное значение которой задаётся вне функции. Например, при **r = 0** результат будет равен нулю при любом значении аргумента **n**. Но эту функцию с аккумулятором легко сделать чистой, например, так:

```
def f(n, r = 1):
    if n <= 1: return r
    else: return f(n-1, r * n)
print(f(5))
```
→ *120*

Здесь мы аккумулятор **r** ввели в список аргументов функции, задав его первоначальное значение по умолчанию.

В этих примерах мы использовали условный оператор **if-else**. Более эффективным был бы переключатель с применением сопоставления с образцом (*pattern matching*). Обычно это оператор **switch-case** но в Python, к сожалению, этот оператор не реализован.

Возможность делать срезы позволяет создавать чистые функции для работы с коллекциями. Вычислим сумму элементов списка:

```
def f(m):
    if not m: return 0
    else: return m[0] + f(m[1:])
print(f([1, 2, 3, 4, 5]))
```
→ *15*

Можно также применить аккумулятор:

```
def f(m, s = 0):
    if not m:  return s
    else: return f(m[1:], s + m[0])
print(f([1, 2, 3, 4, 5]))
```
→ *15*

Или может быть так более читабельно:

```
def f(m, s = 0):
    if m: s = s + m[0]; return f(m[1:], s)
    return s
print(f([1,2,3,4,5]))
```
→ *15*

Можно не применять срезы, если воспользоваться групповым присваиванием в таком виде:

```
x, *y = [2,3,4,5]
```

Получим **x = 2, y = [3,4,5]**. Принято называть: **x** — голова (*head*) списка, **y** – хвост (*tail*) списка. Используя этот приём, программу для суммы списка можно написать так:

```
def f(m):
    x, *y = m
    if not y: return x
    else: return x + f(y)
print(f([1,2,3,4,5])) → 15
```

Или, если больше нравится, так:

```
def f(m):
    x, *y = m
    return x if not y else x + f (y)
print(f([1,2,3,4,5])) → 15
```

Рекурсивный вызов можно поместить во внешнюю функцию:

```
def f(m):
    if not m: return 0
    return g(m)
def g(m):
    return m[0] + f(m[1:])
print(f([1.1, 2.2, 3.3, 4.4])) → 11.0
```

Впрочем, это мало что меняет, по существу.

Кажется, что оператор цикла **while** вполне может заменить рекурсивный цикл:

```
def f(m, s = 0):
    while m:
        s = s + m[0]
        m = m[1:]
    return s
print(f([1,2,3,4,5])) → 15
```

Здесь цикл будет выполняться до тех пор, пока список **m** не станет пустым. Однако, здесь использован «внешний» оператор **while** и функция **f** может считаться чистой только при условии, что **while** не имеет побочных эффектов. Поскольку это вопрос открытый, то видимо подобных ситуаций лучше избегать.

Ещё несколько примеров. Пусть список представлен набором пар. Вычислим сумму разностей элементов в парах. Это можно сделать, например, так:

```
def f(m):
    if not m: return 0
    else: return (m[0][0] - m[0][1]) + f(m[1:])
```

print(f([(3,2), (5,3), (8,4)])) → *7*

Теперь проверим, отсортирован ли заданный список:

```
def f(m):
    if len(m) == 1: return True
    else: return m[0] <= m[1] and f(m[1:])
```

print(f([1,2,3,4,5,6])) → *True*
print(f([1,3,3,4,5,6])) → *True*
print(f([1,3,7,4,5,6])) → *False*

Здесь мы считаем список с одним элементом отсортированным.

Из заданного диапазона выберем числа, которые одновременно делятся на 3, 5 и 7:

```
m = range(1,999)
def f(t):
    if not t: return []
    elif t[0] % 3 != 0 or t[0] % 5 != 0 or t[0] % 7 != 0: return f(t[1:])
    return [t[0]] + f(t[1:])
```

print(f(m)) → *[105, 210, 315, 420, 525, 630, 735, 840, 945]*

Приведённые примеры показывают, что можно и на Python успешно программировать в функциональном стиле. Для тренировки можно составлять программы на Python для примеров из книг по программированию на языке Haskell. Там этих примеров великое множество.

Во всех рассмотренных примерах использованы списки (*list*). Как уже отмечалось, списки изменяемы (*mutable*). Однако в функциональном программировании следует отдавать предпочтение неизменяемым (*immutable*) переменным и коллекциям. В Python к таким коллекциям относятся кортежи (*tuple*) и во многих случаях их можно использовать вместо списков:

```
def f(m, s = 0):
    if not m:  return s
    else: return f(m[1:], s + m[0])
```

print(f((1, 2, 3, 4, 5))) → *15*

То-есть, достаточно квадратные скобки заменить на круглые.

Продвинутые (advanced) свойства функций

Функции в Python обладают довольно богатыми возможностями. Для программистов, ранее незнакомых с Python, некоторые свойства

функций могут даже показаться неожиданными. Но, давайте по-порядку.

Функции в Python являются объектами и могут использоваться на тех же правах, что и любые другие объекты. Тип таких объектов называется **function**. Рассмотрим на примерах основные свойства объектов **function**.

Функции могут служить значениями для переменных и функциями можно инициировать переменные:

```
def f(x): return x**2
g = f
print(g(5)) → 25
print(type(g)) → <class 'function'>
```

Переменная **g** стала функцией в результате инициализации функцией **f**. На самом деле такая инициализация ничего не даёт, кроме присвоения функции второго имени. Переменную можно инициировать безымянной функцией, и тогда эта функция получает имя. Ранее мы это уже применяли.

Функция может быть аргументом для другой функции:

```
def p(x):  return x**2
def f(g, y): return 3 * g(y)
print(f(p, 5)) → 75
```

Здесь функция **p** передана в качестве значения для аргумента **g** функции **f**. О том, что аргумент **g** является функцией, можно узнать только из тела функции **f**. Аргументу **g** можно передавать разные функции, кардиально меняя функциональность для **f**.

Функции могут быть членами коллекций:

```
def g1(x):  return x ** 2
def g2(x): return x ** 3
m = [ (g1, 6), (g2, 3) ]
for (f, y) in m: print(f(y))
```

На консоль будет выведено:

```
36
27
```

Список **m** содержит пары-кортежи, первый член которых представлен функциями, а второй — значениями их аргументов. В цикле **for** аргументу ***f*** передаётся очередная функция, а её аргументу **y** — значение.

Функции могут быть результатом, возвращаемым другой (родительской) функцией.

```
def f():
```

```
    def g(x): return 2 * x
    return g
print(f()(7)) → 14
```

В этом примере также мало пользы — просто вызывая родительскую функцию, на самом деле мы вызываем дочернюю. Но попробуем придумать более содержательный пример:

```
def f(x):
    def g(y): print(x+ ', ' + y)
    return g
f1 = f('Привет')
f1('Люда!') → Привет, Люда!
f1('Катя!') →Привет, Катя!
f2 = f('Столица')
f2('Москва') → Столица, Москва
f2('Пекин') → Столица, Пекин
```

Теперь дочерняя функция **g** использует свой аргумент **y** и аргумент родительской функции **x**. Вызывая функцию **f**, мы получаем новую функцию **f1**. На самом деле это функция **g** в теле которой уже задано и сохраняется значение переменной **x**. Далее можно многократно вызывать функцию **f1**, передавая ей значения аргумента **y**.

В Python имеются средства, позволяющие в самой программе исследовать параметры и свойства применяемых функций. В документации эти действия названы словом *introspection* (самоанализ). Главным образом, для этого применяются функции-атрибуты, о которых говорилось ранее. В Python все объекты без исключения имеют прикреплённые к ним функции-атрибуты (или компоненты). Например, любая переменная имеет функцию-атрибут **__class__**. Попробуем применить её к какой-нибудь переменной:

```
x = 'Marta'
print(x.__class__) → <class 'str'>
```

Значит, **__class__** возвращает тип объекта (**str**) (или класс, которому этот объект принадлежит). Знакомая нам функция **type** просто вызывает эту функцию-атрибут. Получить весь список прикреплённых компонентов можно с помощью функции **dir**: **print(dir(x))**. Получим около 80 названий. Поскольку функции — тоже объекты, они имеют свой список компонентов. Применим к какой-нибудь функции функцию-атрибут **__code__**:

```
def  f(x, y):
    z = 7
```

```
    return x + y +z
print(f.__code__)
```

Получим такую информацию:

<code object f at 0x00000295018AC510, file "D:\progi\Python\prog.py", line 1>

Здесь сообщается шестнадцатеричный адрес объекта, название и адрес скрипта, а также номер строки на которой объявлена функция.

Если выполнить команду:

print(dir(f.__code__))

то получим ещё новый большой список присоединенных компонентов. В частности, там будут компоненты **co_varnames** и **co_argcount**. Попробуем применить их:

print(f.__code__.co_varnames) → *('x', 'y', 'z')*

print(f.__code__.co_argcount) → *2*

Значит, компонент **co_varnames** возвращает список всех переменных функции, а компонент **co_argcount** — количество аргументов функции.

Попробуйте самостоятельно вывести список прикреплённых компонентов для какой-нибудь функции и поэкспериментировать с ними.

При работе с коллекциями можно применять функции-генераторы. Чтобы получить функцию-генератор, надо вместо оператора return использовать функцию **yield**. Функция-генератор создаёт некий объект, который можно использовать в итерациях (в циклах). В документации такие объекты называют *iterable object's*. Давайте разберёмся с этим на примере:

```
m = [2,5,7,4]
def f(s):
    for i in s: yield i ** 2
r = f(m)
```

print(r) → *<generator object f at 0x0000018E26FB4880>*

for i in r: print(i, end = ', ') → *4, 25, 49, 16,*

```
v = f(m)
```

print(tuple(v)) → *(4, 25, 49, 16)*

Функция-итератор **f** возвращает объект **r**. При выводе на консоль получаем только его шестнадцатеричный адрес. Но его можно использовать в циклах, передавать функциям **list**, **tuple** и так далее, получая требуемые результаты. Объект **r** похож на объекты, возвращаемые итераторами. Ранее мы их уже применяли. Можно также к объекту **r** применять функцию **next**, как это мы делали ранее.

Надо только иметь ввиду, что после использования в цикле, *iterable object* становится пустым и для повторного применения его надо создавать заново.

После оператора **yield** можно задавать дополнительные действия над переменной цикла:

```
def f(x):
    for t in x.split(','):  yield t.upper()
print(tuple(f('aaa,bbb,ccc'))) → ('AAA', 'BBB', 'CCC')
print({i: s for (i, s) in enumerate(f('aaa,bbb,ccc'))}) →
{0: 'AAA', 1: 'BBB', 2: 'CCC'}
```

Текстовые значения переменной цикла мы перевели в верхний регистр. В результате создали кортеж и хеш. С помощью метода **send** можно к элементам добавлять разные значения:

```
m = [3,7,5,9]
def f():
    for i in m:
        x = yield i
        print(x, end = ', ')
r =f()
print(next(r))
print(r.send('aa'))
print (r.send('bb'), r.send('cc'))
print(next(r))
print(list(f()))
```

Получим такой результат:

```
3
aa, 7
bb, cc, 5 9
None, 4
None, None, None, None, None, [3, 7, 5, 9, 4]
```

Попробуйте проанализировать, что тут происходит.

```
class A(type):
    def __new__(m, x, y, z):
```

```
            print('In A.new: ', x, y, z, sep='\n...')
            return type.__new__(m, x, y, z)
        def __init__(Class, x, y, z):
            print('In A.init:', x, y, z, sep='\n...')
            print('...init class object:', list(Class.__dict__.keys()))
class B:pass
print('making class')
class C(B, metaclass = A):
        b = 1
        def f(s, a):
            return s.b + a
print('making instance')
X = C()
print('b:', X.b, X.f(2))
```