## Запускаем модуль как самостоятельную программу

Для выполнения функции main() при запуске модуля как программы необходимо поместить её после проверки атрибута \_\_name\_\_:

|  |  |
| --- | --- |
| 1  2 | if \_\_name\_\_ == '\_\_main\_\_':  main() |

Разумеется, эта строчка не выполняется при обычном импорте модуля из другого файла.

## Приручаем Python списки

В языке Python удобно получать срез списка от элемента с индексом from\_inclusive до to\_exclusive с шагом step\_size:

|  |  |
| --- | --- |
| 1 | <list> = <list>[from\_inclusive : to\_exclusive : step\_size] |

Посмотрите, как добавить элемент или коллекцию элементов в список:

|  |  |
| --- | --- |
| 1  2 | <list>.append(<el>)  <list> += [<el>] |

А вот так мы расширим список другим списком:

|  |  |
| --- | --- |
| 1  2 | <list>.extend(<collection>)  <list> += <collection> |

Перейдём к прямой и обратной сортировке. Кроме reverse() и reversed() для обратной сортировки можно также использовать sort() и sorted() с флагом reverse=True.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <list>.sort()  <list>.reverse()  <list> = sorted(<collection>)  <iter> = reversed(<list>)  sorted\_by\_second = sorted(<collection>, key=lambda el: el[1])  sorted\_by\_both = sorted(<collection>, key=lambda el: (el[1], el[0])) |

Суммируем элементы:

|  |  |
| --- | --- |
| 1  2 | sum\_of\_elements = sum(<collection>)  elementwise\_sum = [sum(pair) for pair in zip(list\_a, list\_b)] |

Преобразовываем несколько списков в один:

|  |  |
| --- | --- |
| 1  2 | flatter\_list = list(itertools.chain.from\_iterable(<list>))  product\_of\_elems = functools.reduce(lambda out, x: out \* x, <collection>) |

Получаем список символов из строки:

|  |  |
| --- | --- |
| 1 | list\_of\_chars = list(<str>) |

Чтобы получить первый индекс элемента:

|  |  |
| --- | --- |
| 1 | index = <list>.index(<el>) |

Делаем вставку и удаление по индексу:

|  |  |
| --- | --- |
| 1  2  3  4 | <list>.insert(index, <el>)  <el> = <list>.pop([index]) # удаляет элемент по индексу и возвращает его или последний элемент.  <list>.remove(<el>) # удаляет элемент в первом обнаружении или выдаёт ошибку ValueError.  <list>.clear() # Удаляет все элементы |

## Работаем со словарями

Получаем ключи, значения и пары ключ-значение из словарей:

|  |  |
| --- | --- |
| 1  2  3 | <view> = <dict>.keys()  <view> = <dict>.values()  <view> = <dict>.items() |

Обязательно скопируйте себе этот код. Пригодится, чтобы получить элемент по ключу:

|  |  |
| --- | --- |
| 1  2 | value = <dict>.get(key, default=None) # Возвращает default, если ключ не найден.  value = <dict>.setdefault(key, default=None) # То же, только с добавлением значения в словарь. |

Python позволяет создавать словари со значениями по умолчанию.

|  |  |
| --- | --- |
| 1  2 | <dict> = collections.defaultdict(<type>) # Создаёт словарь с дефолтным значением type.  <dict> = collections.defaultdict(lambda: 1) # Создаёт словарь с дефолтным значением 1. |

Также можно создавать словари из последовательностей пар ключ-значение или из двух последовательностей:

|  |  |
| --- | --- |
| 1  2 | <dict> = dict(<collection>)  <dict> = dict(zip(keys, values)) |

Как и у Python списков, словари поддерживают операцию pop. Только удаление элемента происходит по ключу:

|  |  |
| --- | --- |
| 1 | value = <dict>.pop(key) |

Смотрите, как красиво можно отфильтровать словарь по ключу:

|  |  |
| --- | --- |
| 1 | {k: v for k, v in <dict>.items() if k in keys} |

## Операции над множествами

Подобно спискам и словарям в Python, во множество можно добавить как один элемент, так и целую последовательность:

|  |  |
| --- | --- |
| 1  2  3 | <set> = set()  <set>.add(<el>)  <set>.update(<collection>) |

Конечно, куда без объединения, пересечения и вычисления разности множеств:

|  |  |
| --- | --- |
| 1  2  3  4 | <set> = <set>.union(<coll.>) # Или: <set> | <set>  <set> = <set>.intersection(<coll.>) # Или: <set> & <set>  <set> = <set>.difference(<coll.>) # Или: <set> - <set>  <set> = <set>.symmetric\_difference(<coll.>) # Или: <set> ^ <set> |

## Инструменты Python: лучшая шпаргалка для начинающих

Хотите проверить, является ли коллекция элементов частью множества? Запросто:

|  |  |
| --- | --- |
| 1  2 | <bool> = <set>.issubset(<coll.>) # Или: <set> <= <set>  <bool> = <set>.issuperset(<coll.>) # Или: <set> >= <set> |

Далее рассмотрим удаление элемента из множества. В первом случае операция бросит исключение при отсутствии элемента во множестве:

|  |  |
| --- | --- |
| 1  2 | <set>.remove(<el>)  <set>.discard(<el>) |

## Именованный кортеж

Напомним, что кортеж — неизменяемый список. А именованный кортеж — его подкласс с именованными элементами. Из него можно получить элемент как по индексу, так и по имени:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> from collections import namedtuple  >>> Point = namedtuple('Point', 'x y')  >>> p = Point(1, y=2)  Point(x=1, y=2)  >>> p[0]  1  >>> p.x  1 |

## Функции-генераторы

В отличие от обычных функций, функции-генераторы поставляют значения, приостанавливая свою работу для передачи результата вызывающей программе. При этом они сохраняют информацию о состоянии, чтобы возобновить работу с места, где были приостановлены:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def count(start, step):  while True:  yield start  start += step    >>> counter = count(10, 2)  >>> next(counter), next(counter), next(counter)  (10, 12, 14) |

## Определяем тип

Несмотря на то, что язык программирования Python с динамической типизацией, бывают случаи, когда нужно проверить тип элемента:

|  |  |
| --- | --- |
| 1  2  3 | <type> = type(<el>) # <class 'int'> / <class 'str'> / ...  from numbers import Integral, Rational, Real, Complex, Number  <bool> = isinstance(<el>, Number) |

## Выполняем преобразования со строками Python

Очищаем строку от лишних пробелов или символов <chars> в начале и конце:

|  |  |
| --- | --- |
| 1  2 | <str> = <str>.strip()  <str> = <str>.strip('<chars>') |

Разделяем строку по пробелу или разделителю sep:

|  |  |
| --- | --- |
| 1  2 | <list> = <str>.split()  <list> = <str>.split(sep=None, maxsplit=-1) |

Обратите внимание, как легко склеить список элементов в строку, используя <str> в качестве разделителя:

|  |  |
| --- | --- |
| 1 | <str> = <str>.join(<list>) |

Заменяем значение в строке со старого на новое:

|  |  |
| --- | --- |
| 1 | <str> = <str>.replace(old, new [, count]) |

Хотите проверить, начинается ли строка с определённого символа или подстроки? Можно передавать кортеж из строк для проверки сразу на несколько вариантов. То же самое и с проверкой окончания строки:

|  |  |
| --- | --- |
| 1  2 | <bool> = <str>.startswith(<sub\_str>)  <bool> = <str>.endswith(<sub\_str>) |

Также можно проверить, состоит ли строка только из числовых символов:

|  |  |
| --- | --- |
| 1 | <bool> = <str>.isnumeric() |

Об этом и многом другом подробнее можно почитать в [книгах по Python](https://proglib.io/p/python-best-books/). Ну а мы двигаемся дальше.

## Передаём аргументы в функции

На первом месте при определении и вызове функции всегда находятся позиционные аргументы, а на втором — именованные:

|  |  |
| --- | --- |
| 1  2 | def f(<nondefault\_args>, <default\_args>): # def f(x, y=0)  <function>(<positional\_args>, <keyword\_args>) # f(0, y=0) |

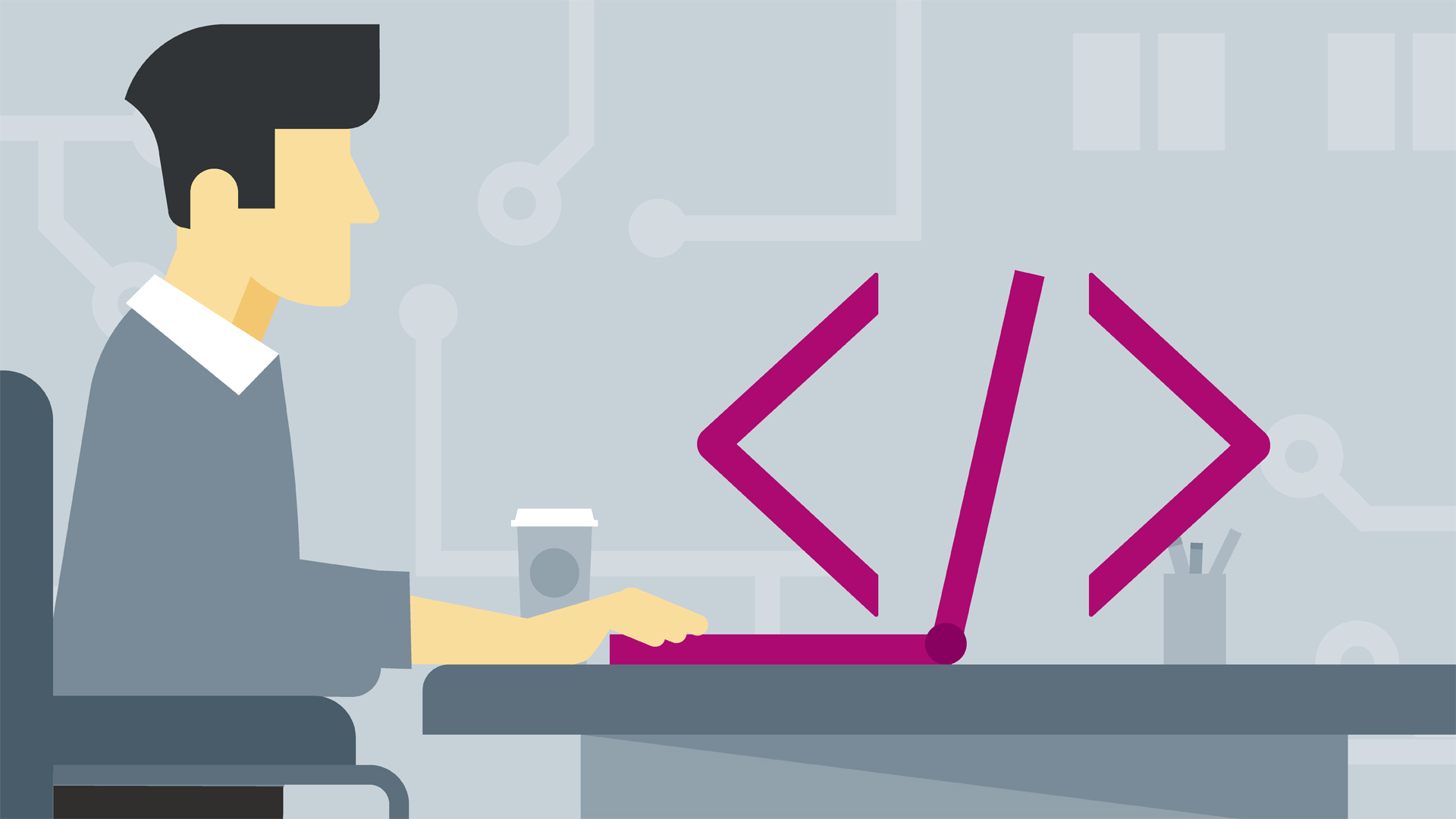
В Python функциям можно передавать список аргументов произвольной длины. В этом случае последовательность \*args передаётся в качестве позиционных аргументов. Пары ключ-значение из словаря \*\*kwargs передаются как отдельные именованные аргументы:

|  |  |
| --- | --- |
| 1  2  3 | args = (1, 2)  kwargs = {'x': 3, 'y': 4, 'z': 5}  func(\*args, \*\*kwargs) |

То же самое:

|  |  |
| --- | --- |
| 1 | func(1, 2, x=3, y=4, z=5) |

## Пишем компактно



Для этого удобно использовать такие инструменты Python, как [анонимные функции](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%BE%D0%BD%D0%B8%D0%BC%D0%BD%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) и генераторы коллекций.

|  |  |
| --- | --- |
| 1  2 | <function> = lambda: <return\_value>  <function> = lambda <argument\_1>, <argument\_2>: <return\_value> |

Создадим список от 1 до 10 в одну строку:

|  |  |
| --- | --- |
| 1 | <list> = [i+1 for i in range(10)] # [1, 2, ..., 10] |

Также в Python есть генераторы множеств:

|  |  |
| --- | --- |
| 1 | <set> = {i for i in range(10) if i > 5} # {6, 7, 8, 9} |

Согласитесь, изящное создание словаря:

|  |  |
| --- | --- |
| 1 | <dict> = {i: i\*2 for i in range(10)} # {0: 0, 1: 2, ..., 9: 18} |

Еще можно создать выражение-генератор, который возвращает объект с результатами по требованию:

|  |  |
| --- | --- |
| 1 | <iter> = (i+5 for i in range(10)) # (5, 6, ..., 14) |

## Инструменты Python поддерживают функциональное программирование

Применяем функцию к каждому элементу последовательности:

|  |  |
| --- | --- |
| 1 | <iter> = map(lambda x: x + 1, range(10)) # (1, 2, ..., 10) |

Отфильтруем элементы последовательности, которые больше **5**:

|  |  |
| --- | --- |
| 1 | <iter> = filter(lambda x: x > 5, range(10)) # (6, 7, 8, 9) |

Следующая функция вычисляет сумму элементов последовательности:

|  |  |
| --- | --- |
| 1  2 | from functools import reduce  <int> = reduce(lambda out, x: out + x, range(10)) # 45 |

## Декораторы

Декоратор принимает функцию, добавляет дополнительную логику и возвращает её.

|  |  |
| --- | --- |
| 1  2  3 | @decorator\_name  def function\_that\_gets\_passed\_to\_decorator():  ... |

Например, декоратор для отладки, возвращающий имя функции при каждом вызове, выглядит следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | from functools import wraps    def debug(func):  @wraps(func)  def out(\*args, \*\*kwargs):  print(func.\_\_name\_\_)  return func(\*args, \*\*kwargs)  return out    @debug  def add(x, y):  return x + y |

wraps — это вспомогательный декоратор, который копирует метаданные функции add() в функцию out().

Без неё 'add.\_\_name\_\_' возвращает 'out'.

## Создаём классы

Далее рассмотрим простейший класс. Метод \_\_init\_\_ вызывается при создании нового экземпляра класса. Метод \_\_str\_\_ вызывается при выполнении преобразования объекта в строку для вывода.

|  |  |
| --- | --- |
| 1  2  3  4  5 | class <name>:  def \_\_init\_\_(self, a):  self.a = a  def \_\_str\_\_(self):  return str(self.a) |

Конечно, классы могут наследоваться от других классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person:  def \_\_init\_\_(self, name, age):  self.name = name  self.age = age    class Employee(Person):  def \_\_init\_\_(self, name, age, staff\_num):  super().\_\_init\_\_(name, age)  self.staff\_num = staff\_num |

## Обрабатываем исключения

Когда в программном коде допускается ошибка, в языке Python автоматически возбуждается исключение. Для перехвата и выполнения восстановительных операций используется try/except:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | while True:  try:  x = int(input('Пожалуйста, введите число: '))  except ValueError:  print('Упс! Ввод неверный! Попробуйте ещё раз...')  else:  print('Thank you.')  break |

Исключение можно возбудить программно с помощью инструкции:

|  |  |
| --- | --- |
| 1 | raise ValueError('Очень конкретное сообщение!') |

Инструкция finally позволяет выполнить заключительные операции Python независимо от того, появилось исключение или нет:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> try:  ... raise KeyboardInterrupt  ... finally:  ... print('Goodbye, world!')  Goodbye, world!  Traceback (most recent call last):  File "<stdin>", line 2, in <module>  KeyboardInterrupt |

## Считываем стандартный ввод

Читаем строку из пользовательского ввода или именованного канала.

Символ новой строки в конце обрезается. Перед чтением ввода отображается строка подсказки prompt:

|  |  |
| --- | --- |
| 1 | <str> = input(prompt=None) |

## Аргументы командной строки

Для получения имени запущенного сценария и аргументов понадобится модуль sys:

|  |  |
| --- | --- |
| 1  2  3 | import sys  script\_name = sys.argv[0]  arguments = sys.argv[1:] |

## Работа с файлами

Для того, чтобы открыть файл в Python, передаём путь <path> в open:

|  |  |
| --- | --- |
| 1 | <file> = open('<path>', mode='r', encoding=None) |

Итак, режимы:

* 'r' — чтение (по умолчанию)
* 'w' — запись (предыдущее данные в файле удаляются)
* 'x' — запись или ошибка, если файл уже существует
* 'a' — добавление
* 'w+' — чтение и запись (с предварительным удалением)
* 'r+' — режим чтения и записи с начала
* 'a+' — то же самое, только с конца
* 't' — текстовый режим (по умолчанию)
* 'b' — бинарный режим

Читаем текст из файла:

|  |  |
| --- | --- |
| 1  2  3 | def read\_file(filename):  with open(filename, encoding='utf-8') as file:  return file.readlines() |

Пишем текст в файл:

|  |  |
| --- | --- |
| 1  2  3 | def write\_to\_file(filename, text):  with open(filename, 'w', encoding='utf-8') as file:  file.write(text) |

## Выполняем обработку каталогов

Чтобы создавать системные инструменты в Python, используйте модуль стандартной библиотеки os. Его прелесть в том, что он пытается предоставить переносимый интерфейс для операционной системы. Например, проверим, существует ли путь path, является ли он файлом или директорией:

|  |  |
| --- | --- |
| 1  2  3  4 | from os import path, listdir  <bool> = path.exists('<path>')  <bool> = path.isfile('<path>')  <bool> = path.isdir('<path>') |

Отобразим список файлов и директорий в каталоге:

|  |  |
| --- | --- |
| 1 | <list> = listdir('<path>') |

Модуль glob реализует механизм подстановки имён файлов. Найдём все файлы с расширением **.gif**:

|  |  |
| --- | --- |
| 1  2  3 | >>> from glob import glob  >>> glob('../\*.gif')  ['1.gif', 'card.gif'] |

## Стандартные инструменты Python позволяют хранить объекты

В стандартной библиотеке Python есть модуль pickle, который позволяет сохранять и восстанавливать объекты:

|  |  |
| --- | --- |
| 1  2  3 | import pickle  <bytes> = pickle.dumps(<object>)  <object> = pickle.loads(<bytes>) |

Таким образом можно извлечь объекты из файла:

|  |  |
| --- | --- |
| 1  2  3 | def read\_pickle\_file(filename):  with open(filename, 'rb') as file:  return pickle.load(file) |

Для записи объектов файл достаточно выполнить:

|  |  |
| --- | --- |
| 1  2  3 | def write\_to\_pickle\_file(filename, an\_object):  with open(filename, 'wb') as file:  pickle.dump(an\_object, file) |

## Используем интерфейс для базы данных [SQLite](https://www.sqlite.org/)

Выполняем подключение к базе данных. Не забываем закрыть его после выполнения всех операций:

|  |  |
| --- | --- |
| 1  2  3  4 | import sqlite3  db = sqlite3.connect('<path>')  ...  db.close() |

Для чтения из базы передаём запрос в метод execute:

|  |  |
| --- | --- |
| 1  2  3  4 | cursor = db.execute('<query>')  if cursor:  <tuple> = cursor.fetchone() # Первая запись.  <list> = cursor.fetchall() # Оставшиеся записи. |

Чтобы внести запись в базу, одного запроса недостаточно. Обязательно нужно сохранить транзакцию:

|  |  |
| --- | --- |
| 1  2 | db.execute('<query>')  db.commit() |