Запускаем модуль как самостоятельную программу

Для выполнения функции `main()` при запуске модуля как программы необходимо поместить её после проверки атрибута `__name__`:

```
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`:

```
<list> = <list>[from_inclusive : to_exclusive : step_size]
```

Посмотрите, как добавить элемент или коллекцию элементов в список:

```
<list>.append(<el>)
<list> += [<el>]
```

А вот так мы расширим список другим списком:

```
<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`.

```
<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:

```
sum_of_elements = sum(<collection>)
elementwise_sum = [sum(pair) for pair in zip(list_a, list_b)]
```

Преобразовываем несколько списков в один:

```
flatter_list = list(itertools.chain.from_iterable(<list>))
product_of_elems = functools.reduce(lambda out, x: out * x, <collection>)
```

Получаем список символов из строки:

```
list_of_chars = list(<str>)
```

Чтобы получить первый индекс элемента:

```
index = <list>.index(<el>)
```

Делаем вставку и удаление по индексу:

```
<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:

```
<view> = <dict>.keys()
<view> = <dict>.values()
<view> = <dict>.items()
```

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

```
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.

```
<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:

```
<dict> = dict(<collection>)
<dict> = dict(zip(keys, values))
```

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

```
value = <dict>.pop(key)
```

Смотрите, как красиво можно отфильтровать словарь по ключу:

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

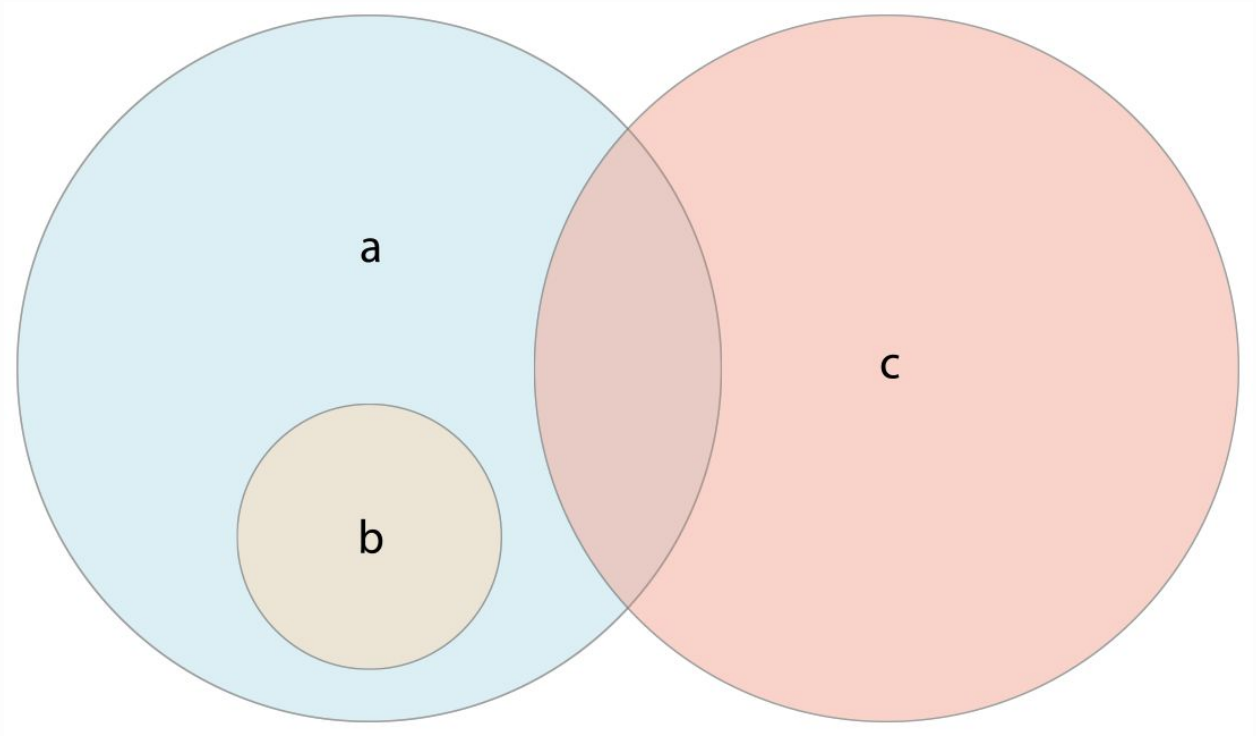
Операции над множествами

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

```
<set> = set()
<set>.add(<el>)
<set>.update(<collection>)
```

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

```
<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>
```



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

```
<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:

```
<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:

```
>>> 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:

```
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:

```
<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:

```
<str> = <str>.strip()
<str> = <str>.strip('<chars>')
```

Разделяем строку по пробелу или разделителю `sep`:

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

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

```
<str> = <str>.join(<list>)
```

Заменяем значение в строке со старого на новое:

```
<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:

```
<bool> = <str>.startswith(<sub_str>)
<bool> = <str>.endswith(<sub_str>)
```

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

```
<bool> = <str>.isnumeric()
```

Об этом и многом другом подробнее можно почитать в книгах по Python. 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:

```
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

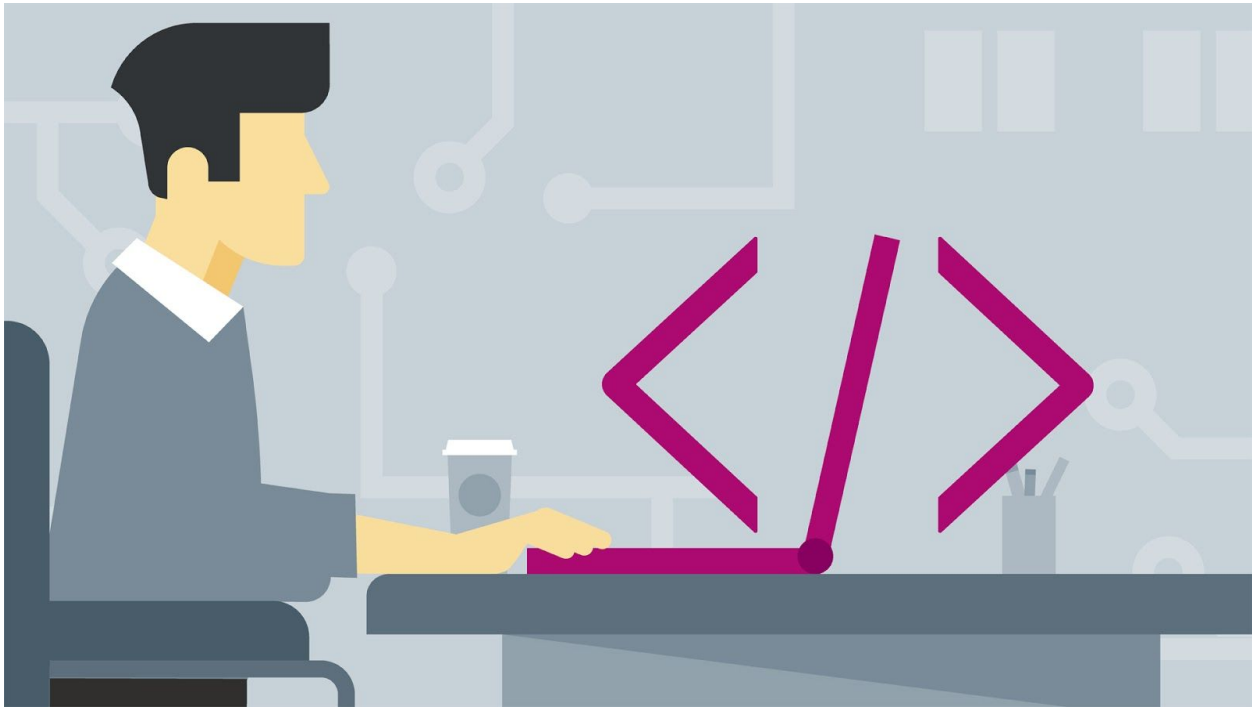
аргументов. Пары ключ-значение из словаря `**kwargs` передаются как отдельные именованные аргументы:

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

То же самое:

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

Пишем компактно



Для этого удобно использовать такие инструменты Python, как анонимные функции и генераторы коллекций.

```
<function> = lambda: <return_value>
```

```
<function> = lambda <argument_1>, <argument_2>: <return_value>
```

Создадим список от 1 до 10 в одну строку:

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

Также в Python есть генераторы множеств:

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

Согласитесь, изящное создание словаря:

```
<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:

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

Инструменты Python поддерживают функциональное программирование

Применяем функцию к каждому элементу последовательности:

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

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

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

Следующая функция вычисляет сумму элементов последовательности:

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

Декораторы

Декоратор принимает функцию, добавляет дополнительную логику и возвращает её.

```
@decorator_name
def function_that_gets_passed_to_decorator():
...
```

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

```
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.

```
class <name>:
    def __init__(self, a):
        self.a = a
    def __str__(self):
       return str(self.a)
```

Конечно, классы могут наследоваться от других классов:

```
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`:

```
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:

```
raise ValueError('Очень конкретное сообщение!')
```

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

```
>>> 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`:

```
<str> = input(prompt=None)
```

Аргументы командной строки

Для получения имени запущенного сценария и аргументов понадобится модуль `sys`:

```
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`:

```
<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:

```
def read_file(filename):
```

```
    with open(filename, encoding='utf-8') as file:
        return file.readlines()
```

Пишем текст в файл:

```
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:

```
from os import path, listdir
<bool> = path.exists('<path>')
<bool> = path.isfile('<path>')
<bool> = path.isdir('<path>')
```

Отобразим список файлов и директорий в каталоге:

```
<list> = listdir('<path>')
```

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

```
>>> 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:

```
import pickle
<bytes> = pickle.dumps(<object>)
<object> = pickle.loads(<bytes>)
```

Таким образом можно извлечь объекты из файла:

```
def read_pickle_file(filename):
    with open(filename, 'rb') as file:
        return pickle.load(file)
```

Для записи объектов файл достаточно выполнить:

```
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

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

```
import sqlite3
db = sqlite3.connect('<path>')
...
db.close()
```

Для чтения из базы передаём запрос в метод `execute`:

```
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:

```
db.execute('<query>')
db.commit()
```