

Программирование на Python

ПРАКТИКА



Guido van Rossum
(Netherlands, 31.01.1956...)

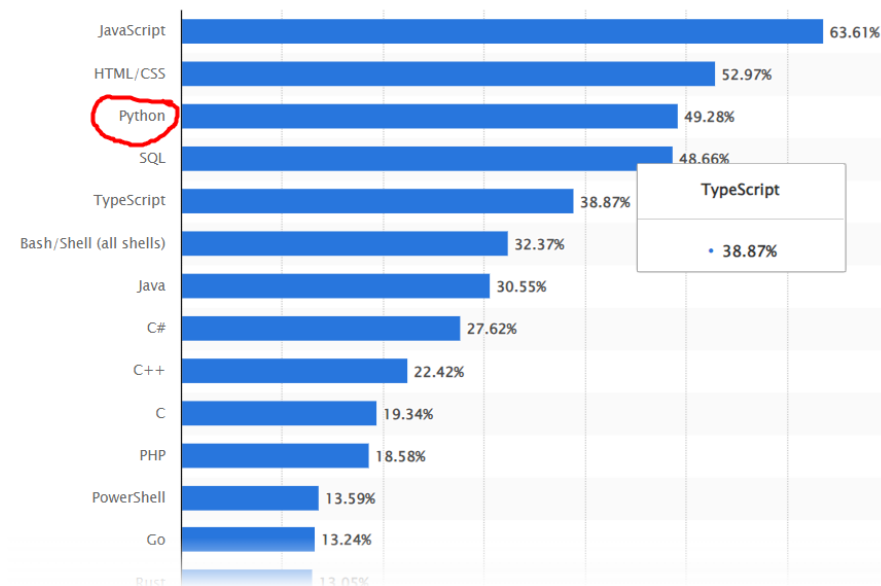


Преимущества Python:

- Простой базовый синтаксис -> низкий порог вхождения
- Расширяемость и гибкость
- Интерпретируемость
- Кроссплатформенность
- Стандарт написания кода (PEP8)
- Широта применения
- OpenSource, большое сообщество

- **1991** – первая публикация Python (0.9.0)
- **1994** – Python 1.0
- **2000** – Python 2.0
- **2008** – Python 3.0
- **2023** – Python 3.11

Наиболее популярные ЯП в 2023 году:

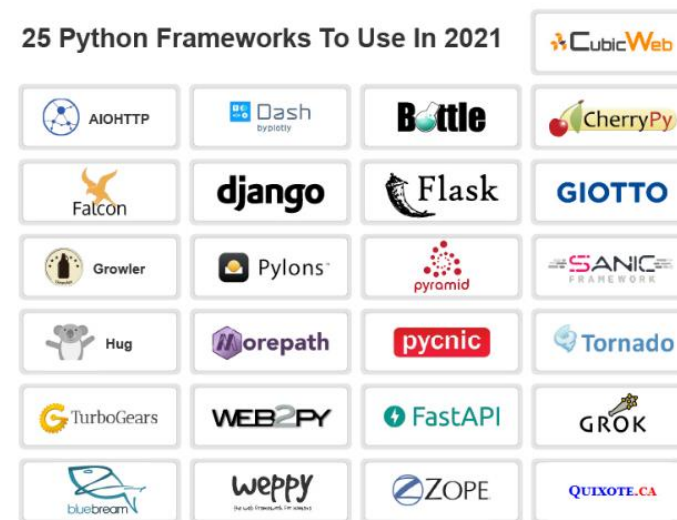


- Веб-разработка
- Data Science и машинное обучение
- Написание скриптов
- Тестирование
- Прототипирование
- Системное программирование
- Приложения баз данных
- Графические интерфейсы
- Бизнес-приложения
- ...



<https://brainstation.io/career-guides/who-uses-python-today>

25 Python Frameworks To Use In 2021



<https://www.trio.dev/python/resources/python-framework>

```
import random

class Person:

    def __init__(
        self,
        name,
        gender,
        age,
        favorite_things,
        happyness_level=50
    ):

        self.name = name
        self.gender = gender
        self.age = age
        self.favorite_things = favorite_things
        self.happyness_level = happyness_level

    def make_happier(self):
        if self.favorite_things:
            thing = random.choice(self.favorite_things)
            self.buy_thing(thing)

            self.happyness_level = self.happyness_level + 10

            if self.gender == 'М':
                verb = "стал"
            else:
                verb = "стала"

            print(f"{self.name} {verb} счастливее, уровень счастья: {self.happyness_level}")

    def buy_thing(self, thing):
        if self.gender == 'М':
            verb = "купил"
        else:
            verb = "купила"

        print(f"{self.name} {verb} {thing}")
```

```
person1 = Person('Алена', 'Ж', '30', ["духи", "цветы", "шоколад"])
✓ 0.0s

person1.make_happier()
✓ 0.0s

Алена купила шоколад
Алена стала счастливее, уровень счастья: 60
```

Строка	Значение
import random	Импорт библиотеки random
Class Person	Объявление класса Person
def __init__(...)	Задание начальных значений для объекта класса
def make_happier(...)	Объявление метода, который делает объект класса Person счастливее
random.choice(...)	Выбор произвольного элемента из списка
self.happyness_level = self.happyness_level + 10	Увеличение уровня счастья объекта класса на 10 пунктов
person1 = Person(...)	Объявление объекта person1 класса Person
person1.make_happier()	Вызов метода, который сделает person1 счастливее

PEP (Python Enhanced Proposal) – набор документов, описывающих особенности языка:

- PEP1: Purpose and Guidelines
- PEP2: Procedure for Adding New Modules
- PEP4: Deprecation of Standard Modules
- PEP5: Guidelines for Language Evolution
- PEP6: Bug Fix Releases
- PEP7: Style Guide for C Code
- **PEP8: Style Guide for Python Code**
- ...
- **PEP20: The Zen of Python** (Простое лучше, чем сложное. Сложное лучше, чем запутанное...)

<https://peps.python.org/pep-0000/>

PEP8 - документ, описывающий общепринятый стиль написания кода на языке Python

<https://peps.python.org/pep-0008/>

<https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html>

«Код читается намного больше раз, чем пишется»

- Правильное использование отступов.
- Правильное использование пробелов внутри строки.
- Правильное использование пустых строк.
- Длину строки рекомендуется ограничить 79 символами.
- Каждый импорт должен быть на отдельной строке.
- Импорты всегда помещаются в начало файла в порядке: стандартные библиотеки -> сторонние -> модули проекта
- Комментарии, противоречащие коду, хуже, чем отсутствие комментариев. Всегда исправляйте комментарии, если меняете код
- Соглашения по именованию.
- Другие рекомендации (исключения, кодировка, контроль версий и т.п.)

- 4 пробела на каждый уровень отступа.
- Выравнивание длинных строк с использованием висячего отступа или неявной линии внутри скобок.
- Закрывающие скобки могут находиться или под первым непробельным символом или под первым символом строки
- Для переноса строки можно использовать “\” или скобки вокруг предложения
- При переносе строк лучше использовать перенос после оператора

```
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)

def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)

foo = long_function_name(var_one, var_two,
                          var_three, var_four)

foo = long_function_name(
    var_one = 1,
    var_two = 2,
    var_three = 3,
    var_four = 4
)

my_list = [
    1, 2, 3,
    4, 5, 6,
]

with open('/path/to/some/file/you/want/to/read') as file_1, \
    open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Не используются:

- После открывающей или перед закрывающей скобкой.
- Перед двоеточием, запятой, точкой с запятой.
- Перед открывающей скобкой после вызова функции, обращения к массиву, списку и т.п.
- Более одного пробела вокруг операторов.
- Вокруг знака присваивания для именованных аргументов функции.

Используются:

- Для выделения операторов (=, +, - и т.п.).
- После двоеточия, запятой, точкой с запятой.
- Для выделения приоритета операторов внутри выражения

```
var1 = 1 ✓
var2 = 2 ✓
var_ = 3 ✗

list1 = [1, 2, 3] ✓
list2=[ 1,2,3_ ] ✗

dict1 = {'key1': 1, 'key2': 2} ✓
dict2 = { _key1_ : 1, 'key2_': 2 } ✗

dict1['key3'] = 3 ✓
dict2 [ 'key3' ] = 4 ✗

exp1 = var1*var1 + var2*var2 ✓
exp2 = var1*var1+_var2_*_var2_ ✗
```


Используются:

- Для отделения функций верхнего уровня и определения классов – 2 строки.
- Определения методов внутри классов.
- Разделения различных групп внутри функций.
- Для разделения логических разделов.

```
def _extract_data_from_forecast(self, forecast_raw: pd.DataFrame) -> N
    data = pd.DataFrame()
    data['temperature'] = forecast_raw['Температура, °C']
    data['wind_speed'] = forecast_raw['Ветер: скорость, м/с']
    data['wind_direction'] = forecast_raw['направление']
    data['pressure'] = forecast_raw['Давление, мм рт. ст.']

    data_len = data.shape[0] # Для контроля равенства длин основного и дополнительного прогнозов

    conditions, cloudiness = self.__get_cloudiness()
    data['conditions'] = conditions[-data_len:]
    data['cloudiness'] = cloudiness[-data_len:]

    precipitation, precipitation_info = self.__get_precipitation()
    data['precipitation'] = precipitation[-data_len:]
    data['precipitation_info'] = precipitation_info[-data_len:]
    data['humidity'] = self.__get_humidity()[-data_len:]
```

```
class Car:

    def __init__(self, model, color):
        self.model = model
        self.color = color

    def get_color(self):
        return self.color

class Moto:

    def __init__(self, model, color):
        self.model = model
        self.color = color
```

- Имена переменных:

lowercase, lower_case_with_underscores

- Имена модулей и пакетов:

lowercase, lower_case_with_underscores

- Имена классов:

CapitalizedWords.

- Исключения (Exceptions):

CapitalizedWords.

- Имена функций:

lower_case_with_underscores.

- Константы:

UPPERCASE_WITH_UNDERSCORES

```
class ForecastRumeteo(Forecast):

    def __init__(self, **kwargs) -> None:
        super().__init__('rumeteo', URL = "https://ru-meteo.ru/ekaterinburg/hour", **kwargs)

    @reconnect()
    def _get_data_from_source(self) -> pd.DataFrame:
        forecast_table = pd.read_html(self.URL, encoding="UTF-8", header=0)
        forecast = self.__forecast_from_table(forecast_table[0])

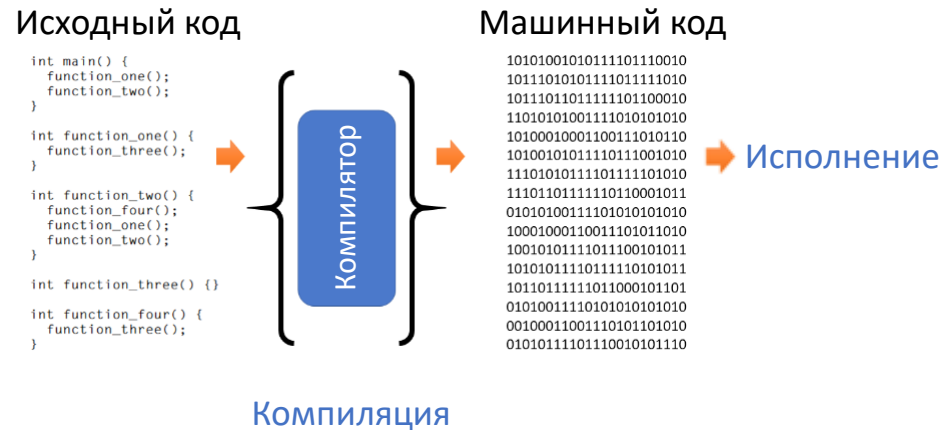
        for i, table in enumerate(forecast_table[1:]):
            # данные по разным дням находятся в разных таблицах
            next_forecast = self.__forecast_from_table(table, i)
            forecast = pd.concat([forecast, next_forecast])

        return forecast
```

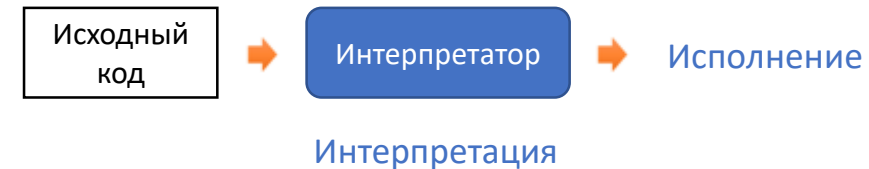
Общие правила:

- Следует избегать использования O, l, i в качестве одиночных идентификаторов
- Названия не должны начинаться с цифры
- Названия не должны совпадать со служебными словами
- Названия не должны содержать специальных символов, пробелов, дефисов и т.п.
- Названия должны быть информативны!**

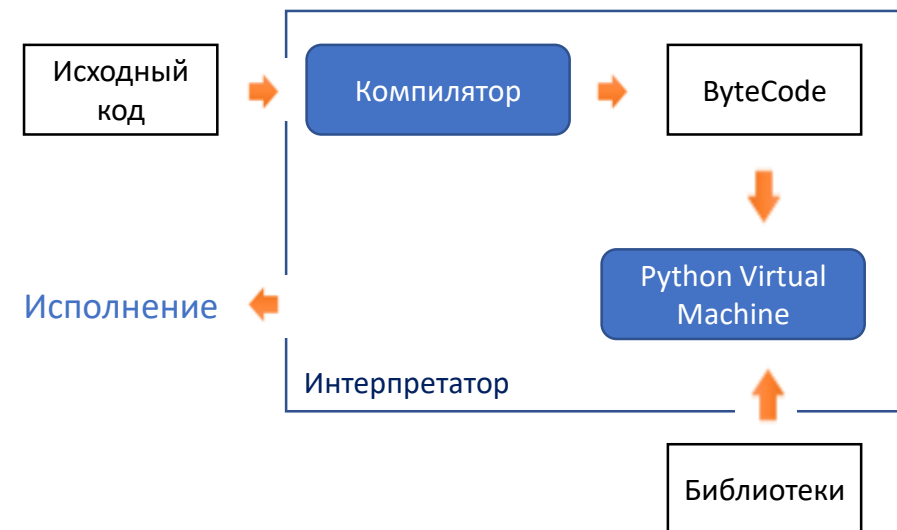
Компилируемые языки (C, .NET, Go, Java...)



Интерпретируемые языки (Python, Perl, JavaScript...)



Реализация Python

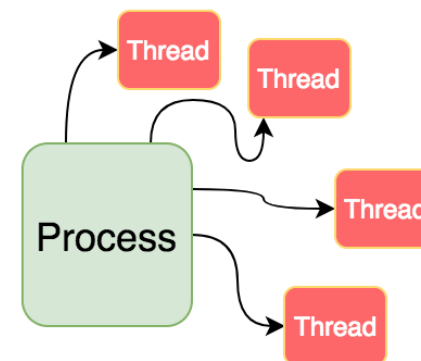


Компилируемые языки	Интерпретируемые языки
Высокая скорость исполнения	Низкая скорость исполнения
Сложность при отладке	Простота отладки
Платформено-зависимые	Кроссплатформенность

Процесс (Process) – часть виртуальной памяти и ресурсов, которую ОС выделяет для выполнения программы.

Поток (Thread) – наименьшая единица выполнения операций с независимым набором инструкций, часть процесса.

Многопоточность (Multithreading) – параллельное (без предписанного порядка во времени) выполнение потоков процесса, порожденного в ОС.



Процессы	Потоки
Каждое приложение имеет минимум один процесс	Каждый процесс имеет минимум один поток
Требуют выделения отдельного места в памяти	Используют память, выделенную под процесс -> создаются и завершаются быстрее
Оперируют только со своими данными. Обмен данными с другими процессами через межпроцессорное взаимодействие	Имеют прямой доступ к ресурсам других потоков процесса
Процесс можно прервать	Поток прервать нельзя

Jupyter Notebook (Jupyter Hub, Jupyter Lab, ...)

```

[2]: !python --version

Python 3.9.7

[3]: print(f'Pandas: {pd.__version__}')
print(f'Numpy: {np.__version__}')
print(f'Sklearn: {sklearn.__version__}')
print(f'TensorFlow: {keras.__version__}')

Pandas: 1.3.4
Numpy: 1.20.3
Sklearn: 0.24.2
TensorFlow: 2.7.0

```

Облачные ресурсы (Google Colab, Kaggle, ...)

←

→

↺

colab.research.google.com/#scrollTo=GJBs_fIovLc

Документ

Добро пожаловать в Colaboratory!

Файл Изменить Вид Вставка Среда выполнения Инструменты Справка

Содержание

🔍

Начало работы

⋮

Анализ и обработка данных

⋮

Машинное обучение

⋮

Ресурсы по теме

⋮

Примеры

⋮

Раздел

+ Код + Текст

📄 Копировать на Диск

+ Код + Текст

```
import numpy as np
from matplotlib import pyplot as plt

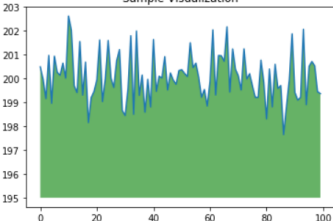
ys = 200 + np.random.randn(100)
x = [x for x in range(len(ys))]

plt.plot(x, ys, '-')
plt.fill_between(x, ys, 195, where=(ys > 195), facecolor='g', alpha=0.6)

plt.title("Sample Visualization")
plt.show()
```

🔍

Sample Visualization



datetime – операции со временем, датой

```
import datetime

print('Текущее время: ', datetime.datetime.now())
print('Год: ', datetime.datetime.now().year)
print('Месяц: ', datetime.datetime.now().month)
print('День: ', datetime.datetime.now().day)
print('10 дней назад: ', datetime.datetime.now() - datetime.timedelta(days=10))
```

✓ 0.0s

```
Текущее время: 2023-09-03 17:49:33.764543
Год: 2023
Месяц: 9
День: 3
10 дней назад: 2023-08-24 17:49:33.764543
```

random – генерация случайных значений

```
import random

print('randint: ', random.randint(1, 10))
print('choice: ', random.choice([10, 21, 32, 41, 5, 6, 7]))
print('random: ', random.random())
print('uniform: ', random.uniform(100, 200))
```

✓ 0.0s

```
randint: 6
choice: 41
random: 0.41935291602161207
uniform: 195.12661159265596
```

math – математические операции

```
import math

print('cos: ', math.cos(0.5))
print('exp: ', math.exp(2))
print('sqrt: ', math.sqrt(121))
print('e: ', math.e)
```

✓ 0.0s

```
cos: 0.8775825618903728
exp: 7.38905609893065
sqrt: 11.0
e: 2.718281828459045
```

os – взаимодействие с ОС

sys – взаимодействие с интерпретатором

```
import sys
import os

print(sys.executable)
print(sys.implementation)
print(os.getpid())
print(os.getcwd())
```

✓ 0.0s

```
c:\Users\askorhodov\Anaconda3\python.exe
namespace(name='cpython', cache_tag='cpython-39', ver
5528
c:\!ML\!Преподавание\urfu\2023 - Анализ данных в ИБ
```

Еще примеры:

- **re** – операции с регулярными выражениями
- **json** – операции с форматом JSON
- **logging** – модуль логгирования
- **pathlib** – модуль взаимодействия с файловыми путями
- ...

Импорт модулей:

- Простой импорт: **import** <модуль>
- Использование псевдонима: **import** <модуль> **as** <псевдоним>
- Выборочный импорт: **from** <модуль> **import** <объект>

Поддерживается, но **не рекомендуется** импортирование всех объектов модуля: **from** <модуль> **import ***

Согласно PEP8:

- Каждый импорт помещается на отдельной строке
- Все импорты помещаются в начало файла в порядке, вначале выполняется импорт стандартных модулей, затем сторонних, затем модулей проекта

В процессе выполнения программы есть риск (возможность) объявить объект с таким же названием, как импортируемый объект.

```
import datetime
print(datetime.datetime.now())
```

✓ 0.0s

2023-09-05 22:21:00.855918

```
import datetime as dt
print(dt.datetime.now())
```

✓ 0.0s

2023-09-05 22:21:19.637830

```
from datetime import timedelta as tdelta, datetime
print(datetime.now())
print(tdelta.days)
```

✓ 0.0s

-09-05 22:28:47.714270
ber 'days' of 'datetime.timedelta' objects>

```
print(dt.datetime.now())
```

```
dt = 'строка'
print(dt)
```

```
print(dt.datetime.now())
```

⊗ 0.4s

2023-09-05 22:40:03.035655

строка

```
-----
AttributeError
Cell In[12], line 6
      3 dt = 'строка'
      4 print(dt)
----> 6 print(dt.datetime.now())
```

AttributeError: 'str' object has no attribute 'datetime'

- `matplotlib` – отрисовка графиков
- `pandas` – работа с табличными данными
- `numpy` – работа с матрицами
- `requests` – работа с http запросами
- `PyTorch` – работа с нейронными сетями
- ...

Для использования библиотеки достаточно ее импортировать, если она уже установлена в системе. Если нет – тогда необходимо ее установить командой `pip install <имя библиотеки>`.

```
import pandas

data = {
    "Ограничение": [60, 90, 40, 20],
    "Локация": ['Город', 'Трасса', 'Ямы', 'Лежащий полицейский']
}
```

```
df = pandas.DataFrame(data)
df
```

✓ 0.0s

	Ограничение	Локация
0	60	Город
1	90	Трасса
2	40	Ямы
3	20	Лежащий полицейский

Python. Импорт внешних модулей

- Проверка установленных внешних модулей: `$ pip list` или `$ pip freeze`
- Установка нового внешнего модуля: `$ pip install <имя модуля>==<версия модуля>`
- Установка внешних модулей обычно производится из репозитория **pypi.org**
- Сохранение списка внешних модулей в файл: `$ pip freeze > <имя файла>`
- Пакетная установка внешних модулей из файла: `$ pip install -r <имя файла>`
- Обновление текущей версии модуля: `$ pip install <имя модуля> --upgrade`
- Удаление модуля: `$ pip uninstall <имя модуля>`

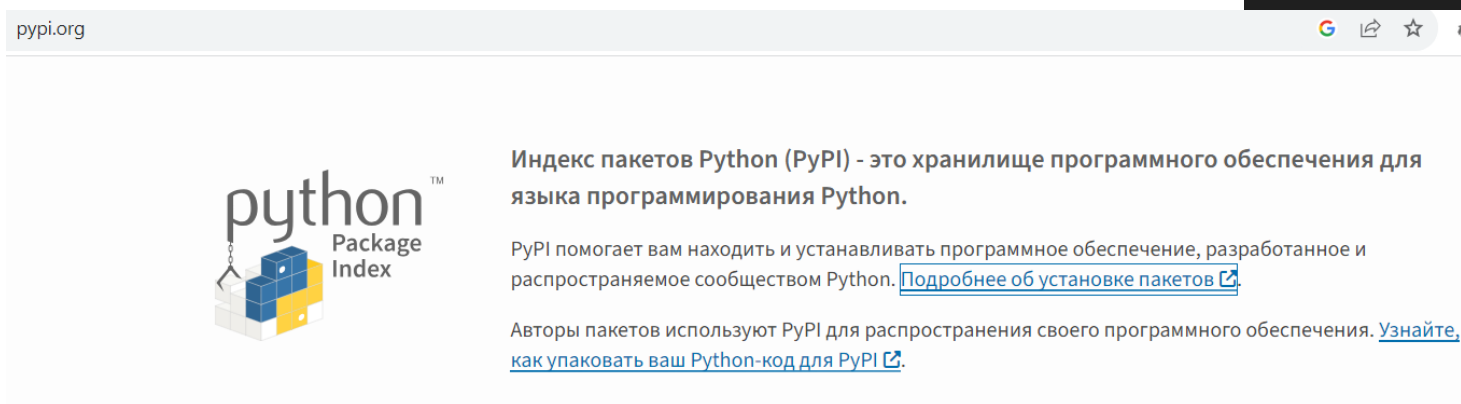
Для исполнения системных команд **из среды Jupyter Notebook** необходимо **добавлять к команде «!»**: `!pip install`

```
!pip list
✓ 2.5s
```

Package	Version
anyio	3.7.0
argon2-cffi	21.3.0
argon2-cffi-bindings	21.2.0
arrow	1.2.3
asttokens	2.2.1
async-lru	2.0.2
attrs	23.1.0
Babel	2.12.1

```
!pip install plotly==5.16.1
✓ 1m 55.2s
```

```
Collecting plotly==5.16.1
  Downloading plotly-5.16.1-py2.py3-none-any.whl (15.6 MB)
    0.0/15.6 MB ? eta -:
    0.0/15.6 MB ? eta -:
```



```
!pip freeze > requirements.txt
✓ 1.3s
```

```
!pip freeze
✓ 1.2s
```

```
anyio==3.7.0
argon2-cffi==21.3.0
argon2-cffi-bindings==21.2.0
arrow==1.2.3
asttokens==2.2.1
```

Пользовательский модуль может представлять собой файл (.py).

Импорт пройдет без проблем, если файл находится в известных интерпретатору директориях для импорта. Список этих директорий:

- **sys.path** (для работы нужно вначале выполнить `import sys`)

Можно добавить директорию с модулем в список **sys.path**.

Если файл модуля находится в поддиректории проекта:

- **import** <поддиректория>.<название модуля>

Если файл модуля находится в родительской директории проекта:

- **import** ..<название модуля>

В файле импортируемого модуля для кода, который не должен выполняться при импорте необходимо добавлять проверку имени исполняемого модуля:

- **if __name__ == "__main__":** <исполняемый код>

Импорт модуля в проекте производится только один раз, если нужно импортировать модуль повторно следует использовать функцию **reload** библиотеки **importlib**.

```
example.py  example.ipynb  new_module.py X
new_module.py > show_module_info
1  MODULE_NAME = 'new_module'
2
3  def show_module_info():
4      print(f"Название модуля: {MODULE_NAME}")
```

```
example.ipynb
example.py
new_module.py
import new_module
new_module.show_module_info()
[13] ✓ 0.0s
... Название модуля: new_module

from new_module import show_module_info
show_module_info()
[14] ✓ 0.0s
... Название модуля: new_module
```

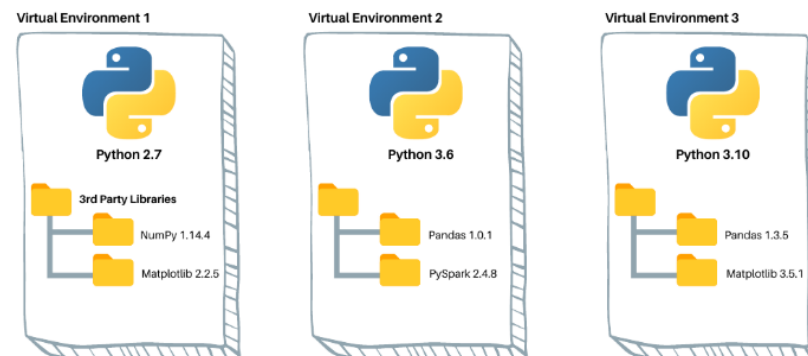
Основное назначение виртуального окружения (Virtual Environment) – создание изолированной конфигурации программного обеспечения с определенным зафиксированным набором библиотек определенных версий.

Для создания виртуального окружения существуют специальные утилиты, например:

- **venv** – встроен в Python, управление через CLI, не быстрый, нельзя поставить версию Python выше установленной
- **Virtualenv** – схожая с venv функциональность, сторонняя утилита
- **conda** – удобный, функциональный, есть GUI, но есть проблемы с производительностью
- **poetry** – функциональный, не простой в настройке

По сути утилиты для создания виртуального окружения создают отдельные директории для каждого виртуального окружения, в которых хранятся все конфигурации и непосредственно интерпретатор Python.

Для создания аналогичной конфигурации на другой машине достаточно установить модули из requirements.txt, который предварительно был выгружен из текущей конфигурации.



Типы данных:

- Встроенные: `int`, `float`, `complex`, `bool`, `str`, `None`.
- Специализированные – требуют загрузки определенного модуля (например, `datetime`)

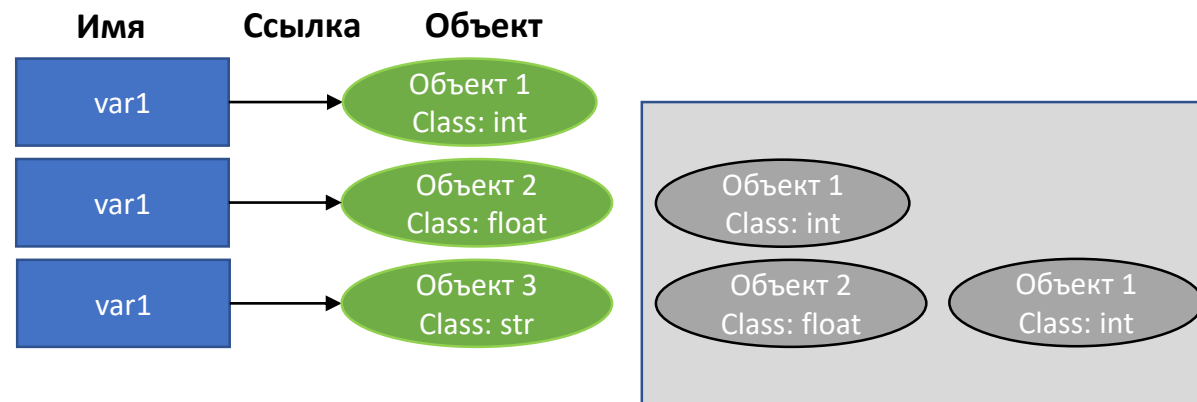
Все типы данных в Python являются классами, а значения – объектами классов.

Для проверки типа данных следует использовать функцию `type()` (например, `type(var1)`)

Определение типов производится автоматически в ходе выполнения программы.

```
var1 = 13
print(var1, type(var1), hex(id(var1)))
var1 = 2.4
print(var1, type(var1), hex(id(var1)))
var1 = 'some string'
print(var1, type(var1), hex(id(var1)))
```

```
13 <class 'int'> 0x7ff98cfff94a8
2.4 <class 'float'> 0x19f878fed70
some string <class 'str'> 0x19f8800aff0
```



1. Создайте переменную с именем `var1` и присвойте ей значение 10. Переменная – это ссылка на объект 10.
2. Выведите в консоль идентификатор объекта (`id(<переменная>)`), на который ссылается переменная `var1`.
3. Преобразуйте идентификатор в `hex` формат при помощи функции `hex()`.
4. Присвойте переменной `var2` значение 20.
5. Выведите в консоль идентификатор для `var1`, объекта 10 и объекта 20.

Python. Арифметические операции

Приоритет	Оператор Python	Операция	Пример	Результат
1	**	Возведение в степень	5 ** 5	3125
2	%	Деление по модулю (получение остатка)	16 % 7	2
3	//	Целочисленное деление (дробная часть отбрасывается)	13 // 3	4
4	/	Деление	39 / 2	19.5
5	*	Умножение	123 * 321	39483
6	-	Вычитание	999 – 135	864
7	+	Сложение	478 + 32	510

<https://proglib.io/p/samouchitel-po-python-dlya-nachinayushchih-chast-3-tipy-dannyh-preobrazovanie-i-bazovye-operacii-2022-10-14>

В арифметических операциях Python пытается автоматически выполнить преобразование операндов к одному типу:

- Если один из операндов **complex**, то делается попытка преобразовать другой в **complex**.
- Аналогичная ситуация с **float**.

Если автоматически преобразовать операнды не получается, необходимо применять явное преобразование типов: **int()**, **float()**, **str()**

Не все операнды могут быть преобразованы в другие типы, например, не получится преобразовать в **int** строки, содержащие что-либо, кроме цифр.

Также если число складывается со строкой, необходимо выполнить преобразование операндов к одному типу, т.к. интерпретатор не знает, что нужно нам: сложить строки или числа.

```
a = 2
b = 3.1
c = a + b
print(c, type(c))
```

```
5.1 <class 'float'>
```

```
b = '3.1'
c = a + b
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[14], line 2
      1 b = '3.1'
----> 2 c = a + b

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
b = float(3.1)
c = a + b
print(c, type(c))
```

```
5.1 <class 'float'>
```

```
b = int('3.1x')
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[16], line 1
----> 1 b = int('3.1x')
      2 c = a + b
      3 print(c, type(c))

ValueError: invalid literal for int() with base 10: '3.1x'
```

```
print(int(True))
print(int(False))
print(float(True))
print(float(False))
```

```
1
0
1.0
0.0
```

```
print("a" + str(2))
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[11], line 1
----> 1 print("a" + 2)

TypeError: can only concatenate str (not "int") to str
```

1. Создайте переменную с именем `var1` и присвойте ей значение 10. Выведите в консоль тип `var1`.
2. Присвойте переменной `var1` ее значение, разделенное на 3. Выведите в консоль тип `var1`.
3. Создайте переменную `var2` и присвойте ей значение остатка от деления переменной `var1` на 2. Выведите в консоль тип `var2`.
4. Измените тип переменной `var2` на строковый. Сложите `var1` и `var2`
5. Присвойте переменной `var1` результат целочисленного деления этой переменной на 1. Выведите в консоль тип `var1`. Если результат не является целочисленным значением, преобразуйте его в `int`.
6. Измените тип переменной `var1` на строковый. Сложите `var1` и `var2`. Выведите в консоль тип получившегося результата

Структура	Класс Python	Пример	Использование
Список	list	[1, 2, 3] ['cat', 'dog', 'duck', 'dog']	l = [1, 2, 3] l = list(('cat', 'dog', 'duck', 'dog'))
Кортеж	tuple	(1, 2, 3) ('cat', 'dog', 'duck', 'dog')	t = (1, 2, 3) t = tuple(('cat', 'dog', 'duck', 'dog'))
Множество	set	{1, 2, 3} {'cat', 'dog', 'duck'}	s = (1, 2, 3) s = set(('cat', 'dog', 'duck', 'dog'))
Словарь	dict	{'cat': 1, 'dog': 2, 'duck': 3}	d = dict('cat'=1, 'dog'=2, 'duck'=3)
Неизменяемое множество	frozenset	{1, 2, 3} {'cat', 'dog', 'duck'}	s = (1, 2, 3) s = frozenset(('cat', 'dog', 'duck', 'dog'))
Диапазон	range	1, 3, 5, 7, 9	r = range(1, 10, 2)

Все структуры данных в Python также являются классами, а значения – объектами классов.

Для проверки типа структуры данных следует также использовать функцию **type()** (например, *type(var1)*)

Определение типов структур производится автоматически в ходе выполнения программы.

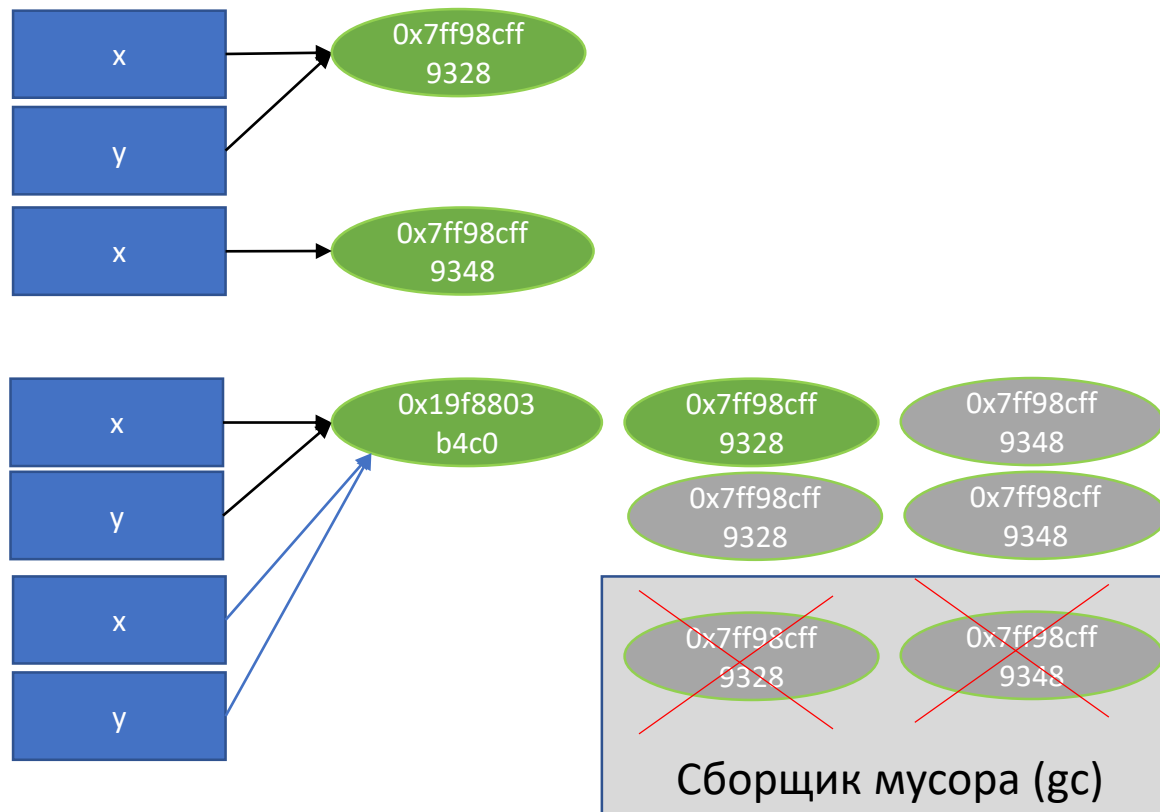
- **Неизменяемые (immutable):** int, bool, tuple, float, string, frozenset
- **Изменяемые (mutable):** list, set, dict

Доступ к неизменяемым объектам осуществляется быстрее.

```
print('== immutable ==')
x = 1
y = x
print(x, hex(id(x)), y, hex(id(y)))
x = x+1
print(x, hex(id(x)), y, hex(id(y)))
print(hex(id(2)), hex(id(1)))
```

```
print('== mutable ==')
x = list([1, 2, 3])
y = x
print(x, hex(id(x)), y, hex(id(y)))
x.append(4)
print(x, hex(id(x)), y, hex(id(y)))
```

```
== immutable ==
1 0x7ff98cff9328 1 0x7ff98cff9328
2 0x7ff98cff9348 1 0x7ff98cff9328
0x7ff98cff9348 0x7ff98cff9328
== mutable ==
[1, 2, 3] 0x19f8803b4c0 [1, 2, 3] 0x19f8803b4c0
[1, 2, 3, 4] 0x19f8803b4c0 [1, 2, 3, 4] 0x19f8803b4c0
```



Список – упорядоченная* изменяемая коллекция объектов произвольных типов

```
lst = [1, 2, 'one', 'two', ['three', 'four', 'five']]
print(f'lst[0]: {lst[0]}')
print(f'lst[3]: {lst[3]}')
print(f'lst[1:]: {lst[1:]}')
print(f'lst[-1:]: {lst[-1:]}')
print(f'lst[2:4]: {lst[2:4]}')
print(f'lst[::2]: {lst[::2]}')
print(f'lst[::-1]: {lst[::-1]}')
```

```
lst[0]: 1
lst[3]: two
lst[1:]: [2, 'one', 'two', ['three', 'four', 'five']]
lst[-1:]: [['three', 'four', 'five']]
lst[2:4]: ['one', 'two']
lst[::2]: [1, 'one', ['three', 'four', 'five']]
lst[::-1]: [['three', 'four', 'five'], 'two', 'one', 2, 1]
```

***Упорядоченная коллекция** – коллекция, значения которой представляют собой упорядоченное множество, с каждым элементом которого связан его порядковый номер. Любое значение, кроме первого, имеет предшественника, и любое значение, кроме последнего, имеет последователя, определяемых отношением порядка данного типа.

Метод	Описание
list.append(x)	Добавляет элемент в конец списка
list.extend(L)	Расширяет список, добавляя в конец все элементы списка L
list.insert(i, x)	Вставляет на i-ый элемент значение x
list.remove(x)	Удаляет первый элемент в списке, имеющий значение x.
list.pop([i])	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
list.index(x, [start [, end]])	Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end)
list.count(x)	Возвращает количество элементов со значением x
list.sort([key=функция])	Сортирует список на основе функции
list.reverse()	Разворачивает список
list.copy()	Копия списка
list.clear()	Очистка списка

Кортеж – упорядоченная неизменяемая коллекция объектов произвольных типов = неизменяемый список

```
tpl = (1, 2, 'one', 'two', ['three', 'four', 'five'])
print(f'tpl[1]: {tpl[1]}')
print(f'tpl*2: {tpl*2}')
print(f'tpl+2: {tpl+tpl}')
tpl[-1][0] = 111111111
print(f'tpl[-1][0]: {tpl[-1][0]}')
tpl[-1] = 5
```



```
tpl[1]: 2
tpl*2: (1, 2, 'one', 'two', ['three', 'four', 'five'], 1, 2, 'one', 'two', ['three', 'four', 'five'])
tpl+2: (1, 2, 'one', 'two', ['three', 'four', 'five'], 1, 2, 'one', 'two', ['three', 'four', 'five'])
tpl[-1][0]: 111111111
```

TypeError

Traceback (most recent call last)

Cell In[68], line 7

```
5 tpl[-1][0] = 111111111
6 print(f'tpl[-1][0]: {tpl[-1][0]}')
----> 7 tpl[-1] = 5
```

TypeError: 'tuple' object does not support item assignment

К кортежам применимы все операции, не изменяющие кортеж. Кортежи занимают меньше места в памяти.

1. Создайте переменную с именем `var1` и присвойте ей значение 10.
2. Выведите в консоль результат сложения переменной `var1` и 13, отдельно выведите значение переменной `var1`
3. Создайте переменную `tpl1` и присвойте ей в качестве значения кортеж, состоящий из целочисленных значений от 1 до 10.
4. Выведите в консоль по очереди: результат умножения кортежа `tpl1` на 2, последний элемент кортежа, элемент с первого по пятый включительно, значение кортежа.
5. Присвойте второму элементу кортежа значение 10, выведите в консоль значение второго элемента.
6. Создайте переменную `lst1` и присвойте ей в качестве значения список, состоящий из целочисленных значений от 1 до 5, расположенных в случайном порядке. Отсортируйте список по убыванию, но не присваивайте полученный результат переменной `lst1`, а выведите в консоль ее значение.
7. Создайте кортеж `tpl2`, состоящий из произвольных значений, одно из которых будет являться списком из трех элементов. Обратитесь через индекс кортежа к элементу, который является списком, и измените его второй элемент. Выведите значение кортежа в консоль.
8. Произведите действия, аналогичные предыдущему пункту, только в качестве списка для кортежа укажите переменную `lst1`. После манипуляций выведите значение кортежа `tpl2` и списка `lst1` в консоль.

Примеры операций со строками

```
s = "1. Пример строки"
print(type(s))
print(s)
s = '2. Можно использовать одинарные кавычки'
print(s)
s = '''3. Если текст длинный, можно использовать
многострочный вариант с тремя кавычками'''
print(s)
```

```
<class 'str'>
1. Пример строки
2. Можно использовать одинарные кавычки
3. Если текст длинный, можно использовать
многострочный вариант с тремя кавычками
```

```
print(f"4. Можно вычислять длину строки - len(s): {len(s)}")
print('5. Строки ' + 'можно ' + 'складывать')
print('6. И даже ' + 'умножать '*3)
```

```
4. Можно вычислять длину строки - len(s): 81
5. Строки можно складывать
6. И даже умножать умножать умножать
```

```
print('7. Можно искать вхождение подстроки: ', 'Если' in s, 'числа' in s)
print('8. Строки можно разбивать в списки: ', s.split())
print('9. А потом соединять: ', ' '.join(s.split()))
```

```
7. Можно искать вхождение подстроки: True False
8. Строки можно разбивать в списки: ['3.', 'Если', 'текст', 'длинный,', 'можно', 'ис']
9. А потом соединять: 3. Если текст длинный, можно использовать многострочный вариант
```

```
print('10. Можно все сделать заглавными буквами: ', s.upper())
print('11.', 'или сделать как в предложении'.capitalize())
```

```
10. Можно все сделать заглавными буквами: 3. ЕСЛИ ТЕКСТ ДЛИННЫЙ, МОЖНО ИСПОЛЬЗОВАТЬ
МНОГОСТРОЧНЫЙ ВАРИАНТ С ТРЕМЯ КАВЫЧКАМИ
11. Или сделать как в предложении
```

<https://proglab.io/p/samouchitel-po-python-dlya-nachinayushchih-chast-4-metody-raboty-so-strokami-2022-10-24>

<https://pythonworld.ru/typy-dannyx-v-python/stroki-funkcii-i-metody-strok.html>

Строка – итерируемый объект

```
print('Исходная строка: ', s, '\n')
print('Элемент строки: ', s[3])
print('Несколько последовательных элементов: ', s[:10])
print('То же самое через 1 символ: ', s[::2])
```

```
Исходная строка: 3. Если текст длинный, можно использовать
многострочный вариант с тремя кавычками
```

```
Элемент строки: Е
Несколько последовательных элементов: 3. Если те
то же самое через 1 символ: 3 ситктдин он совоаьмоотонйвратстєяквчаи
```

```
for i in s[3:8]:
    print(i)
```

```
Е
с
л
и
```

И еще методы работы со строками:

- **replace()** – замена символа.
- **isalnum(), isalpha(), isdigit()** – определение, состоит строка только из букв/цифр, только букв или только цифр соответственно.
- **find(), rfind()** – поиск искомой подстроки в строке, возвращает индекс первого вхождения слева или справа соответственно.
- **startswith()** – возвращает True, если строка начинается с заданного символа.
- **strip(), lstrip()** и **rstrip()** – удаление пробелов в начале и конце строки, только слева или только справа соответственно

1. Присвойте строковой переменной `s` любой текст из нескольких (>2) предложений длиной не менее 20 символов.
2. Выведите на экран `длину` переменной `s`.
3. Приведите весь текст к `нижнему регистру`, выведите на экран.
4. Замените все пробелы в переменной на знак `@`, выведите на экран.
5. Разбейте ваш текст на предложения и выведите их на экран в виде `списка`
6. Проверьте, состоит ли ваше первое предложение `только из букв`. Выведите результат на экран.
7. Выведите на экран второе слово первого предложения вашего текста `в обратном порядке`.
8. Разбейте ваше второе предложение и выведите на экран слова с первого по пятое включительно, через одно.

Множество – неупорядоченная изменяемая коллекция неповторяющихся объектов произвольных типов

```
st1 = {1, 2, 3, 4}
st2 = {4, 5, 6, 7, 8}
print(f"Длина st1: {len(st1)}")
print(f"Элемент 7 есть в st2: {7 in st2}")
print(f"Объединение st1 и st2: {st1 | st2}") #set.union
print(f"Пересечение st1 и st2: {st1 & st2}") #set.intersection
print(f"Есть в st1, но нет в st2: {st1 - st2}") #set.difference
print(f"Есть только в st1 или в st2: {st1 ^ st2}") #set.symmetric_difference
st1.add(10)
print(f"st1: {st1}")
st1[0]
```

Длина st1: 4

Элемент 7 есть в st2: True

Объединение st1 и st2: {1, 2, 3, 4, 5, 6, 7, 8}

Пересечение st1 и st2: {4}

Есть в st1, но нет в st2: {1, 2, 3}

Есть только в st1 или в st2: {1, 2, 3, 5, 6, 7, 8}

st1: {1, 2, 3, 4, 10}

TypeError Traceback (most recent call last)

Cell In[83], line 11

9 st1.add(10)

10 print(f"st1: {st1}")

---> 11 st1[0]

TypeError: 'set' object is not subscriptable

1. Создайте два множества `st1` и `st2`, каждое из которых заполните 10 случайными (модуль `random` функция `randint`) целочисленными значениями от 0 до 20.
2. Выведите в консоль длины множеств.
3. Выведите в консоль элементы, которые есть в множестве `st1`, но нет в `st2`.
4. Выведите в консоль элементы, которые встречаются только в одном множестве.
5. Выведите элементы, которые встречаются в обоих множествах.

Словарь (ассоциативный массив, хеш-таблица) – **неупорядоченная** **изменяемая** коллекция произвольных объектов с **доступом по ключу**.

```
d = {'Вася': 5, 'Алена': 10, 'Петя': 28}
print(f"d.items(): {d.items()}")
print(f"d.keys(): {d.keys()}")
print(f"d.keys(): {d.keys()}")
petya = d.pop('Петя')
print(f"petya: {petya}, d: {d}")
d['Марина'] = 29
print(f"d: {d}")
d.update([('Вася', 10), ('Петя', 20)])
print(f"d: {d}")
print(f"d['Вася']: {d['Вася']}")
print(f"d.get('Неизвестный чел', 'я такого не знаю'): {d.get('Неизвестный чел', 'я такого не знаю')}")
print(f"d['Неизвестный чел']: {d['Неизвестный чел']}")
```

```
d.items(): dict_items([('Вася', 5), ('Алена', 10), ('Петя', 28)])
d.keys(): dict_keys(['Вася', 'Алена', 'Петя'])
d.keys(): dict_keys(['Вася', 'Алена', 'Петя'])
petya: 28, d: {'Вася': 5, 'Алена': 10}
d: {'Вася': 5, 'Алена': 10, 'Марина': 29}
d: {'Вася': 10, 'Алена': 10, 'Марина': 29, 'Петя': 20}
d['Вася']: 10
d.get('Неизвестный чел', 'я такого не знаю'): я такого не знаю
```

```
-----
KeyError                                Traceback (most recent call last)
Cell In[90], line 13
     11 print(f"d['Вася']: {d['Вася']}")
     12 print(f"d.get('Неизвестный чел', 'я такого не знаю'): {d.get('Неизвестный чел', 'я такого не знаю')}")
--> 13 print(f"d['Неизвестный чел']: {d['Неизвестный чел']}")

KeyError: 'Неизвестный чел'
```

Метод	Описание
dict.fromkeys(seq[, value])	Создает словарь с ключами из seq и значением value (по умолчанию None).
dict.get(key[, default])	Возвращает значение ключа, но если его нет, не бросает исключение, а возвращает default (по умолчанию None).
dict.items()	Возвращает пары (ключ, значение).
dict.keys()	Возвращает ключи в словаре.
dict.pop(key[, default])	Удаляет ключ и возвращает значение. Если ключа нет, возвращает default (по умолчанию бросает исключение).
dict.popitem()	Удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение KeyError.
dict.setdefault(key[, default])	Возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ со значением default (по умолчанию None).
dict.update([other])	Обновляет словарь, добавляя пары (ключ, значение) из other. Существующие ключи перезаписываются
dict.values()	Возвращает значения в словаре.
dict.copy()	Поверхностная копия словаря
dict.clear()	Очистка словаря

Ключами в словаре могут выступить **только неизменяемые** объекты!

1. Создайте словарь *employees*, где ключом будет выступать кортеж из имени (*'name'*) и фамилии (*'surname'*) сотрудника, а значениями – словарь из возраста (*'age'*) и должности (*'position'*)
2. Добавьте еще одного сотрудника в словарь
3. Выведите на экран запрос несуществующего сотрудника, чтобы запрос вернул (*None*)
4. Выведите на экран *всех* сотрудников в формате <имя> <фамилия>

```
if <условие1>:
    <действие1>
elif <условие2>:
    <действие2>
elif <условие3>:
    <действие3>
else:
    <действие4>
```

<условие1> – выражение, возвращающее объект типа bool:

- Результат операций сравнения: **>**, **<**, **>=**, **<=**, **==**, **!=**
- Результат вычисления логических операций: **and**, **or**, **not**
- Результат работы оператора **in** / **not in**: наличие/отсутствие значения в наборе значений
- Результат работы оператора **is**: идентичность объектов

match <переменная>: Python >= 3.10

```
case <значение1>:
    <действие1>
case <значение2>:
    <действие2>
case _:
    <действие по умолчанию>
```

<переменная> – имя переменной, по которой будет идти сравнение

<значение1> – значение переменной, при котором будет выполняться **<действие1>**, аналогично конструкции:

```
if <переменная> == <значение1>:
    <действие1>
```

- Любое число, не равное 0, или непустой объект – истина (**True**).
- Числа, равные 0, пустые объекты и значение **None** – ложь (**False**)
- Операции сравнения применяются к структурам данных рекурсивно
- Операции сравнения возвращают **True** или **False**
- Логические операторы **and** и **or** возвращают истинный или ложный объект-операнд

```
True
False
True
False
False
True
True
False
True
print(1 == 1)
print(1 == '1')
print(bool('asf'))
print(bool(0))
print(bool([]))
print(bool([0, 0]))
print('a' in ['a', 'b', 'c'])
a = 1; b = 2; print(a is b)
a = 2; b = 2; print(a is b)
```

```
for <элемент> in <объект>:  
    <действие1>  
    break  
    <действие2>  
    continue  
    <действие3>  
else:  
    <действие4>
```

```
while <условие>:  
    <действие1>  
    break  
    <действие2>  
    continue  
    <действие3>  
else:  
    <действие4>
```

<объект> – итерируемый объект (списки, строки, кортежи, словари)

<элемент> – переменная, которой будут поочередно присваиваться значения итерируемого объекта

<условие> – условие, которое будет проверяться перед каждым выполнением цикла

else (опционально) – добавление действия, которое выполнится после прохождения всего цикла (или исключения StopIteration)

break (опционально) – прерывание цикла (else игнорируется)

continue (опционально) – переход к следующему итерируемому объекту без выполнения оставшегося кода.

- Если цикл **for** не сложный для ускорения обработки лучше использовать функцию **map()** или генераторы.
- По завершению цикла **for** переменная, **<элемент>** остаётся доступным, равным последнему значению итерируемого объекта.
- Изменение последовательности **<объект>** в ходе итерирования может приводить к ошибкам и пропускам элементов.
- Цикл **while** используется, когда точное число повторений неизвестно и может изменяться в зависимости от поведения переменной в теле цикла.
- В других случаях предпочтительнее использовать цикл **for**, т.к. он быстрее **while**.
- Цикл **while** может стать бесконечным, необходимо предусматривать механизмы защиты.

1. Минималистичный for

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

2. while для той же задачи выглядит не так изящно

```
i = 0

while i < 5:
    print(i)
    i += 1
```

```
0
1
2
3
4
```

3. Зато while подойдет для задач с условием. Но если переменная error никогда не изменится, цикл станет бесконечным

```
error = False

while not error:
    execute_some_function()
```

4. Работа операторов break и continue

```
for word in ['dog', 'table', 'cat', 'horse', 'tiger']:
    if word == 'table':
        print('table - лишнее слово')
        continue

    if word == 'horse':
        print('лошадь мы не ожидали')
        break

    print(word)
```

```
dog
table - лишнее слово
cat
лошадь мы не ожидали
```

5. Можно перебирать итерируемые объекты с порядковым номером (enumerate)

```
for i, word in enumerate(['dog', 'table', 'cat',
                           'horse', 'tiger']):
    print(i, word)
```

```
0 dog
1 table
2 cat
3 horse
4 tiger
```

6. Можно перебирать сразу несколько объектов (zip)

```
A = range(5)
print('A = ', A)
B = range(10,15)
print('B = ', A)

for a, b in zip(A, B):
    print(f"a = {a}, b = {b}")
```

```
A = range(0, 5)
B = range(0, 5)
a = 0, b = 10
a = 1, b = 11
a = 2, b = 12
a = 3, b = 13
a = 4, b = 14
```

7. Можно использовать несколько уровней вложений циклов (вложенные циклы)

```
A = [1, 2]
B = ['один', 'два']
C = ['one', 'two']

for a in A:
    for b in B:
        for c in C:
            print(f"a = {a}, b = {b}, c = {c}")
```

```
a = 1, b = один, c = one
a = 1, b = один, c = two
a = 1, b = два, c = one
a = 1, b = два, c = two
a = 2, b = один, c = one
a = 2, b = один, c = two
a = 2, b = два, c = one
a = 2, b = два, c = two
```

1. Используйте ваш словарь `employees`, добавьте каждому сотруднику в словарь его параметров параметр «зарплата» (`'salary'`) с произвольными целочисленными значениями.
2. Найдите (`for`) самого низкооплачиваемого и высокооплачиваемого сотрудника, выведите на экран (например, “Петров получает 100000, а Сидоров 10000”). Для форматирования можно воспользоваться f-string:

```
print(f"{key_max[1]} получает {employees[key_max]['salary']}р, а {key_min[1]} - {employees[key_min]['salary']}р")
```

Иванов получает 60866р, а Сидоров - 33825р

```
>>> name = "Eric"
>>> age = 74
>>> f"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.'
```

3. Повышайте зарплату самому низкооплачиваемому сотруднику до тех пор, пока (`while`) она не сравняется или не станет выше зарплаты самого высокооплачиваемого сотрудника. Шаг повышения зарплаты должен быть равен одной десятой разницы между зарплатами указанных сотрудников. После каждого повышения выводите на экран новую зарплату, а в конце – увольте сотрудника путем удаления его из словаря.

Создание списка

```
: %%time
lst_1 = []

for i in range(10_000_000):
    if i%2 == 0:
        lst_1.append(i)
```

Wall time: 1.22 s

```
: %%time
lst_2 = [i for i in range(10_000_000) if i%2==0]
```

Wall time: 694 ms

List comprehension

```
lst_1 = []

for i in range(10):
    if i%2 == 0:
        lst_1.append(i)

lst_1
```

[0, 2, 4, 6, 8]

```
lst_2 = [i for i in range(10) if i%2==0]
lst_2
```

[0, 2, 4, 6, 8]

Создание словаря

```
%%time
dict_1 = {}

for i in range(10_000_000):
    if i%2 == 0:
        dict_1[str(i)] = i
```

Wall time: 3.17 s

```
%%time
dict_2 = {str(i): i for i in range(10_000_000) if i%2==0}
```

Wall time: 2.63 s

Dict comprehension

```
: %%time
dict_1 = {}

for i in range(10_000_000):
    if i%2 == 0:
        dict_1[str(i)] = i
```

Wall time: 3.17 s

```
: %%time
dict_2 = {str(i): i for i in range(10_000_000) if i%2==0}
```

Wall time: 2.63 s

1. При помощи *list comprehension* создайте список из целочисленных значений от 1 до 20.
2. При помощи *list comprehension* создайте список из целочисленных значений от 1 до 20, но только нечетные числа.
3. При помощи *dict comprehension* создайте новый словарь из словаря *employees*, где ключом будет фамилия, а значением – должность сотрудника.
4. При помощи *dict comprehension* создайте новый словарь из словаря *employees*, где ключом будет фамилия, а значением – должность сотрудника, если его зарплата больше 50000.

Функция – объект, принимающий аргументы и возвращающий значение.

Функции позволяют многократно использовать повторяющийся код, упростить код, избежать ошибок.

```
price = 1000
vat_rate = 20

if price < 0 or vat_rate < 0:
    print("Некорректные входные данные.")

vat = (price * vat_rate) / 100
print("Сумма НДС:", vat)

price = 200
vat_rate = 18

if price < 0 or vat_rate < 0:
    print("Некорректные входные данные.")

vat = (price * vat_rate) / 100
print("Сумма НДС:", vat)

price = 2120
vat_rate = 33

if price < 0 or vat_rate < 0:
    print("Некорректные входные данные.")

vat = (price * vat_rate) / 100
print("Сумма НДС:", vat)
```

✓ 0.0s

Сумма НДС: 200.0
Сумма НДС: 36.0
Сумма НДС: 699.6



С использованием функций

```
def calculate_vat(price, vat_rate):
    if price < 0 or vat_rate < 0:
        print("Некорректные входные данные.")
        return None

    vat = (price * vat_rate) / 100
    print("Сумма НДС:", vat)

    return vat

vat_rate = 20

vat_amount = calculate_vat(price=1000, vat_rate=20)
vat_amount = calculate_vat(price=200, vat_rate=18)
vat_amount = calculate_vat(price=2120, vat_rate=33)
```

✓ 0.0s

Сумма НДС: 200.0
Сумма НДС: 36.0
Сумма НДС: 699.6

Python имеет огромный набор встроенных функций, например:

Функция	Описание
print()	Выводит данные на стандартный вывод (консоль).
len(iterable)	Возвращает длину (количество элементов) итерируемого объекта, такого как список, кортеж или строка.
type(object)	Возвращает тип объекта, например, int, str, list, и так далее.
input(prompt)	Считывает строку с консоли, часто используется для ввода данных пользователем.
range([start], stop[, step])	Создает последовательность чисел от start до stop (не включая stop) с заданным шагом step.
sum(iterable)	Возвращает сумму всех элементов итерируемого объекта, например, сумму всех элементов списка.
sorted(iterable[, key=функция, reverse=булево])	Возвращает новый список, отсортированный в порядке возрастания (или убывания). Может использовать функцию key для более сложных сортировок.
dir([object])	Возвращает список всех атрибутов и методов объекта (или текущего модуля, если объект не указан).
help([object])	Выводит справочную информацию об объекте (или текущем модуле, если объект не указан).
open(file, mode)	Открывает файл и возвращает объект файла, который может использоваться для чтения ('r'), записи ('w'), добавления ('a') и других операций над файлами.

```
s = 'Hello world'
print(s)
print(f"len(s): {len(s)}")
print(f"abs(-11): {abs(-11)}")
print(f'any[list]: {any([True, False, False, False])}')
print(f'max[list]: {max([1, 10, 2.5, 8])}')
```

```
✓ 0.0s
Hello world
len(s): 11
abs(-11): 11
any[list]: True
max[list]: 10
```

```
dir(s)
✓ 0.0s
['_add_',
 '_class_',
 '_contains_',
 '_delattr_',
 '_dir_',
 '_doc_',
 '_eq_',
 '_format_',
 '_ge_',
 '_getattr_',
 '_getitem_',
 '_getnewargs_',
 '_gt_',
 '_hash_',
 '_init_',
 '_init_subclass_',
 '_iter_',
 '_le_',
 '_len_',
 '_lt_',
 '_mod_',
 '_mul_',
 '_ne_',
 '_new_',
 '_reduce_',
 ...]
```

Пример пользовательской функции

```
# объявление функции
ключевое слово def simple_fun(a, b):
    имя функции
    аргументы функции
    print('Вызов функции simple_fun с аргументами', a, b)
    Тело функции
    return a + b
    Возвращаемое значение

# вызов функции
simple_fun(10, 21)
```

✓ 0.0s Python

Вызов функции simple_fun с аргументами 10 21

31

Аргументы:

- Позиционные
- Именованные

Сначала в функцию передаются позиционные аргументы, а затем – именованные.

В случае с именованными аргументами порядок перечисления может не совпадать

В функцию можно передавать неограниченное количество аргументов, при этом

позиционные обозначаются ***args**, а именованные ****kwargs**, при этом ***args**

распаковываются в кортеж, а ****kwargs** – в словарь.

Оператор * отвечает за распаковку итерируемого объекта:

```
lst = ['a', 'b', 'c', 'd']
print(lst[0], lst[1], lst[2], lst[3])
print(*lst)
```

✓ 0.0s

a b c d
a b c d

```
def calculate_vat(price, discount=10, vat=20):
    ...

calculate_vat(1234)
calculate_vat(1234, discount=20)
calculate_vat(1234, vat=30)
calculate_vat(1234, vat=30, discount=20)
```

```
def calculate_vat(*args, **kwargs):
    print('*args:', [arg for arg in args])
    print '**kwargs:', {key: value for key, value in kwargs.items()})
```

```
calculate_vat('1', 1234)
calculate_vat('2', 1234, discount=20)
calculate_vat('3', 1234, vat=30)
calculate_vat('4', 1234, vat=30, discount=20)
```

✓ 0.0s

```
*args: ['1', 1234]
**kwargs: {}
*args: ['2', 1234]
**kwargs: {'discount': 20}
*args: ['3', 1234]
**kwargs: {'vat': 30}
*args: ['4', 1234]
**kwargs: {'vat': 30, 'discount': 20}
```

Функция всегда возвращает значение, это может быть в т.ч. пустое значение (None).

Ключевое слово для возвращения значения – оператор **return**. Таких операторов в теле функции может быть несколько, после выполнения оператора **return** исполнение функции прекращается.

Возвращаемых значений может быть несколько, для этого нужно перечислить значения через запятую после оператора **return**. В этом случае функция будет возвращать кортеж (tuple):

```
def calculate_vat(price, vat_rate=20):
    if price < 0 or vat_rate < 0:
        print("Некорректные входные данные.")
        return None

    vat = (price * vat_rate) / 100
    print("Сумма НДС:", vat)

    return vat, price

result = calculate_vat(price=1000)
print(result, type(result))
vat_amount, price = calculate_vat(price=1000)
print(vat_amount, price)
```

✓ 0.0s

Сумма НДС: 200.0
(200.0, 1000) <class 'tuple'>
Сумма НДС: 200.0
200.0 1000

```
def calculate_vat(price, vat_rate=20):
    if price < 0 or vat_rate < 0:
        print("Некорректные входные данные.")
        return None

    vat = (price * vat_rate) / 100
    print("Сумма НДС:", vat)

    return vat

vat_rate = 20

vat_amount = calculate_vat(price=1000)
vat_amount = calculate_vat(price=200, vat_rate=18)
vat_amount = calculate_vat(price=2120, vat_rate=33)
vat_amount = calculate_vat(price=-2120, vat_rate=33)
```

✓ 0.0s

Сумма НДС: 200.0
Сумма НДС: 36.0
Сумма НДС: 699.6
Некорректные входные данные.

Область видимости переменных:

- Локальная
- Глобальная

Переменные, объявленные в теле функции, создаются во время выполнения функции и остаются видимыми только внутри этой функции. Для доступа к глобальной области видимости необходимо использовать ключевое слово **global**, в этом случае локальная переменная создаваться не будет, а будет меняться значение глобальной переменной. Использование глобальных переменных усложняет анализ кода и в целом не рекомендуется.

```
var1 = 'value_1='

def modify_value(value):
    return value*2

print('Возвращаемое значение: ', modify_value(var1))
print('Исходное значение: ', var1)
```

✓ 0.0s

Возвращаемое значение: value_1=value_1=
Исходное значение: value_1=

```
var1 = 'value_1='

def modify_value(value):
    var1 = value*2
    return var1

print('Возвращаемое значение: ', modify_value(var1))
print('Исходное значение: ', var1)
```

✓ 0.0s

Возвращаемое значение: value_1=value_1=
Исходное значение: value_1=

```
var1 = 'value_1='

def modify_value(value):
    global var1
    var1 = value*2
    return var1

print('Возвращаемое значение: ', modify_value(var1))
print('Исходное значение: ', var1)
```

✓ 0.0s

Возвращаемое значение: value_1=value_1=
Исходное значение: value_1=value_1=

Если в качестве аргумента функции передается изменяемый параметр, то есть риск этот параметр изменить.

Следует очень аккуратно взаимодействовать с изменяемыми аргументами, т.к. они могут привести к неожиданному поведению алгоритма.

```
lst = ['a', 'b', 'c']

def modify_list(lst):
    lst1 = lst
    lst1.append('new')
    return lst1

print('Возвращаемое значение: ', modify_list(lst))
print('Исходное значение: ', lst)

lst = ['a', 'b', 'c']

def create_modify_list(lst):
    lst1 = lst[:]
    lst1.append('new')
    return lst1

print('Возвращаемое значение: ', create_modify_list(lst))
print('Исходное значение: ', lst)
```

✓ 0.0s

```
Возвращаемое значение: ['a', 'b', 'c', 'new']
Исходное значение: ['a', 'b', 'c', 'new']
Возвращаемое значение: ['a', 'b', 'c', 'new']
Исходное значение: ['a', 'b', 'c']
```


1. Создайте функцию `calculate_bmi()`, которая рассчитывает индекс массы тела (ИМТ, `bmi`). На вход функция будет принимать вес (`weight: float`) в килограммах и рост (`height: float`) в метрах.
2. Предусмотрите проверку отрицательных, нулевых и нечисловых значений для аргументов функции. В случае некорректных входных аргументов необходимо вывести на экран, например «Вес и/или рост имеют некорректные значения!», и остановить выполнение функции.
3. ИМТ – отношение массы человека к квадрату его роста. Функция должна возвращать значение ИМТ и выводить его на экран.

1. Для словаря `employees` из предыдущей части напишите функцию `change_currency()`, которая будет переводить зарплату работников в доллары.
2. В качестве параметров функции нужно передать данные сотрудника ("`employee`" – значение ключа словаря `employees`) валюту ("`currency`") и обменный курс ("`exchange_rate`"), новое значение зарплаты будет вычисляться как текущее значение `salary`, умноженное на `exchange_rate`.
3. Для того, чтобы понимать в какой валюте работник получает зарплату, нужно добавить в словарь каждого сотрудника ключ "`currency`", изначально равный "`rub`". При смене валюты значение ключа меняется на соответствующую валюту (например, "`usd`").
4. Также напишите функцию `show_salary()`, которая будет принимать на вход элемент словаря `employees` и выводить на экран сообщение формата: {должность} {фамилия} получает {зарплата} {валюта}.

```
for key, value in employees.items():  
    employees[key] = change_currency(value, 'usd', 0.01)  
    show_salary(key, employees[key])
```

```
Инженер Иванов получает  978.30 usd  
Управляющий Петров получает  982.98 usd  
Электрик Сидоров получает  935.10 usd  
Рабочий Прохоров получает  455.36 usd
```

1. Для словаря *employees* из напишите функцию *add_employee()*, которая будет добавлять сотрудников в исходный словарь.
2. В качестве обязательных параметров в функцию нужно передать исходный словарь (*employees: dict*), имя (*name: str*), фамилию (*surname: str*), должность (*position: str*), возраст (*age: int*), в качестве необязательных параметров – зарплату (*salary: float, по умолчанию 50000*) и валюту (*currency: str, по умолчанию "rub"*), в которой выплачивается зарплата.
3. Функция должна добавлять сотрудника в исходный словарь в принятом для словаря формате. Если добавляемый сотрудник уже существует в исходном словаре, следует вывести на экран предупреждение и прекратить выполнение функции.
4. Также напишите функцию *fire_employee()*, которая будет удалять сотрудника по его имени и фамилии. Если указанный сотрудник отсутствует, следует вывести на экран предупреждение и прекратить выполнение функции.

1. Создайте функцию `group_types()`, которая будет принимать на вход любое количество позиционных аргументов и группировать их по типу. Аргументы могут быть типов `str`, `int`, `float`, другие типы игнорируются.
2. Функция должна возвращать словарь, в котором ключи – это типы данных, а значения – отсортированные по убыванию значения соответствующих типов.

```
group_types(4, 2, 3, 5.0, 6.5, 99.2, 'cat', 'dog', 'bee')  
✓ 0.0s  
{'int': [4, 3, 2], 'float': [99.2, 6.5, 5.0], 'str': ['dog', 'cat', 'bee']}
```

Функции высшего порядка принимают одну (или более) функций в качестве аргументов и/или в качестве результата возвращают функцию.

В Python существуют 3 встроенные функции высшего порядка: `map()`, `filter()` и `reduce()`:

- **`map(function, iter)`** – применяет функцию ко всем элементам итерируемого объекта.
- **`filter(condition, iter)`** – отбирает элементы итерируемого объекта согласно условию.
- **`reduce(function, iter, [, initial])`** – применяет функцию двух аргументов кумулятивно к элементам итерируемого объекта, необязательно начиная с начального аргумента. (для использования необходим импорт модуля `functools`). По сути функция уменьшает итерируемый объект до одного значения.

```
def some_func(n):
    return n**2 + 10

numbers = (1, 2, 3, 4)
result = map(some_func, numbers)
print(list(result))
```

✓ 0.0s

[11, 14, 19, 26]

```
def some_condition(value):
    if value > 3:
        return True
    return False

numbers = (1, 2, 3, 4, 5, 6, 7, 8)
result = filter(some_condition, numbers)
print(list(result))
```

✓ 0.0s

[4, 5, 6, 7, 8]

```
from functools import reduce

def some_func(x, y):
    return x * y

numbers = (1, 2, 3, 4, 5, 6)
result = reduce(some_func, numbers)
print(result)
```

✓ 0.0s

720

Лямбда-функция – анонимная функция с неограниченным количеством элементов, но только с одним выражением, которое вычисляется и возвращается.

lambda arguments: expression

Лямбда-функции можно не объявлять и использовать, например, в качестве аргументов для функций высшего порядка (**map()**, **filter()**, **reduce()**) или для других операций с итерируемыми объектами, например, для сортировки словарей.

```
dict1 = {
    (0, 0): {'parameter1': 10, 'parameter2': 21, 'parameter3': 30},
    (0, 1): {'parameter1': 20, 'parameter2': 23, 'parameter3': 30},
    (1, 1): {'parameter1': 30, 'parameter2': 2, 'parameter3': 30},
    (1, 2): {'parameter1': 40, 'parameter2': 0, 'parameter3': 30},
    (2, 1): {'parameter1': 50, 'parameter2': 10, 'parameter3': 30},
}

sorted(list(dict1.items()), key=lambda x: x[1]['parameter2'])
```

✓ 0.0s

```
[((1, 2), {'parameter1': 40, 'parameter2': 0, 'parameter3': 30}),
 ((1, 1), {'parameter1': 30, 'parameter2': 2, 'parameter3': 30}),
 ((2, 1), {'parameter1': 50, 'parameter2': 10, 'parameter3': 30}),
 ((0, 0), {'parameter1': 10, 'parameter2': 21, 'parameter3': 30}),
 ((0, 1), {'parameter1': 20, 'parameter2': 23, 'parameter3': 30})]
```

```
def double(x):
    return x*2

print(double(10), hex(id(double)), type(double))

double_lambda = lambda x: x*2

print(double_lambda(10), hex(id(double_lambda)), type(double_lambda))
```

✓ 0.0s

```
20 0x15886e48550 <class 'function'>
20 0x15886e480d0 <class 'function'>
```

```
numbers = (1, 2, 3, 4)
result = map(lambda x: x**2+10, numbers)
print(list(result))
```

✓ 0.0s

```
[11, 14, 19, 26]
```

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8)
result = filter(lambda x: x > 3, numbers)
print(list(result))
```

✓ 0.0s

```
[4, 5, 6, 7, 8]
```

1. При помощи конструкции `List Comprehension` и модуля `random` создайте список `lst` из 1 миллиона случайных целочисленных значений.
2. При помощи функции `map()` и `лямбда-выражений` сгенерируйте список `lst1`, в котором каждый элемент будет равен квадратному корню соответствующего значения списка `lst`.
3. Выполните пункт 2, используя цикл `for`
4. Выполните пункт 2, используя `List Comprehension`
5. Сравните время выполнения формирования списка `lst1`, для этого можно использовать модуль `datetime` (`datetime.datetime.now()` – получение текущего времени в формате `datetime`, времена можно вычитать), выведите на экран время выполнения для каждого варианта.
6. При помощи функции `filter()` и `лямбда-выражений` сгенерируйте список `lst2`, в котором будут только те элементы списка `lst`, которые больше среднего значения списка `lst`.

Замыкание (closure) функции – концепция, в которой **вложенная функция имеет доступ** к локальным переменным функции более высокого порядка, после того, как **внешняя функция уже завершила свою работу**.

Вложенная функция

```
def print_some_text(text):
    print("Вызов функции print_some_text")

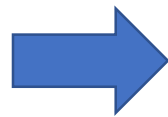
    def add_important_information():
        print("Ниже будет напечатан какой-то текст:")
        print(text)

    add_important_information()

print_some_text('Текст основной функции')
```

[219] ✓ 0.0s

... Вызов функции print_some_text
Ниже будет напечатан какой-то текст:
Текст основной функции



Замыкание функции

```
def print_some_text(text):
    print("Вызов функции print_some_text")

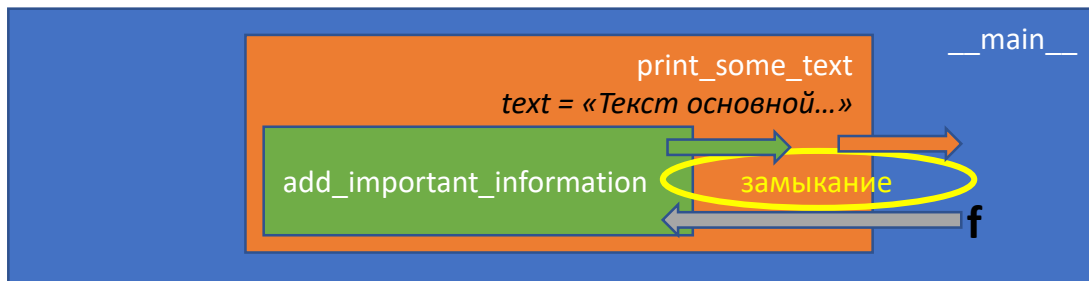
    def add_important_information():
        print("Ниже будет напечатан какой-то текст:")
        print(text)

    return add_important_information

f = print_some_text('Текст основной функции')
print(f)
f()
```

✓ 0.0s

Вызов функции print_some_text
<function print_some_text.<locals>.add_important_information at 0x0000015884C91280>
Ниже будет напечатан какой-то текст:
Текст основной функции



Замыкания полезны, например, когда вам нужно создать функции с долгоживущими переменными или когда вы хотите скрыть некоторые данные от внешнего кода, делая их доступными только внутри функции.

1. Создайте функцию `create_multiplier(factor)`, которая принимает на вход число, которое будет являться множителем.
2. Внутри функции `create_multiplier()` создайте функцию `multiplier(x)`, которая принимает на вход число, которое нужно умножить на множитель.
3. Используя концепцию замыкания функций объявите два объекта `double` и `triple`, которые будут ссылаться на функцию `create_multiplier`, но передавать в нее разные аргументы: 2 и 3 соответственно.
4. Присвойте значения функций `double` и `triple` переменным `result1` и `result2` соответственно. В функции `double` и `triple` в качестве аргумента передайте одинаковое число, например, 10.

```
# Создаем две функции для умножения на разные факторы
double = create_multiplier(2)
triple = create_multiplier(3)

# Используем созданные функции
result1 = double(10)
result2 = triple(10)

print(result1)
print(result2)
```

✓ 0.0s

```
20
30
```

Декораторы – функции, которые позволяют изменить поведение произвольной функции без изменения ее кода.

Принцип работы декоратора основан на принципах механизме замыкания функций. Т.е. в функцию-декоратор передается функция, которую нужно декорировать: `decorator(function)`. Для упрощения синтаксиса декоратор обычно обозначается с использованием «@» и указывается непосредственно перед объявлением функции. Для функции можно использовать несколько декораторов, при этом они указываются друг под другом, **порядок декорирования важен**.

В указанном примере декоратор сохраняет значение времени до запуска декорируемой функции и вычитает его из текущего времени при завершении работы декорируемой функции, что позволяет вычислить время работы функции. При этом декорируемая функция выполняет свой алгоритм без изменений.

```
import time

def some_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        print(f"Время работы функции: {time.time() - start_time: .2f} s")

        return result
    return wrapper

@some_decorator
def some_function(name, args):
    print(f"Выполнение '{name}' с аргументами '{args}'...")
    time.sleep(5)
    return 'Результат работы функции some_function'

# some_function = some_decorator(some_function) -> @some_decorator

result = some_function('Имя функции', 'Аргумент')
result
```

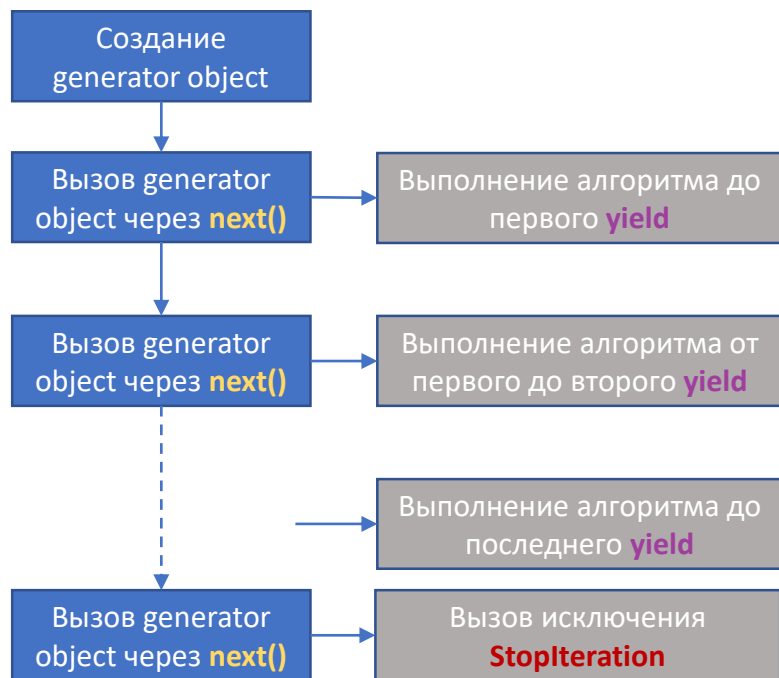
Декоратор

Функция, которую нужно расширить

вызов функции

Выполнение 'Имя функции' с аргументами 'Аргумент'...
Время работы функции: 5.00 s
'Результат работы функции some_function'

Генератор – объект, который сразу **при создании не вычисляет** значения всех своих элементов, а хранит в памяти только последний вычисленный элемент, правило перехода к следующему и условие, при котором выполнение прерывается. Вычисление следующего значения происходит лишь при выполнении метода **next()**, код выполняется до следующего оператора **yield**. Предыдущее значение при этом теряется. Генератор уменьшает потребление памяти и ускоряет процесс обработки



```

a = generator_example(4)
next(a)
next(a)
next(a)
next(a)

```

⊗ 0.0s

```

=== Вызов функции generator_example ===
Обработка итерации i = 1, count = 1
Обработка итерации i = 2, count = 2
Обработка итерации i = 3, count = 3

```

```

-----
StopIteration                               Trace
C:\Users\ASKORO~1\AppData\Local\Temp\ipykernel_
3 next(a)
4 next(a)
----> 5 next(a)

StopIteration:

```

```

def generator_example(m):
    count = 1
    print('=== Вызов функции generator_example ===')

    for i in range(1, m):
        print(f'Обработка итерации i = {i}, count = {count}')
        yield i**2 + count
        count += 1

a = generator_example(8)

print("Генератор возвращает значение только при выполнении метода next()")
for i in a:
    print("Вычисленное значение: ", i)

```

✓ 0.0s

```

Генератор возвращает значение только при выполнении метода next()
=== Вызов функции generator_example ===
Обработка итерации i = 1, count = 1
Вычисленное значение: 2
Обработка итерации i = 2, count = 2
Вычисленное значение: 6
Обработка итерации i = 3, count = 3
Вычисленное значение: 12
Обработка итерации i = 4, count = 4
Вычисленное значение: 20
Обработка итерации i = 5, count = 5
Вычисленное значение: 30
Обработка итерации i = 6, count = 6
Вычисленное значение: 42
Обработка итерации i = 7, count = 7
Вычисленное значение: 56

```

```

generator = generator_example(4)
generator

```

✓ 0.0s

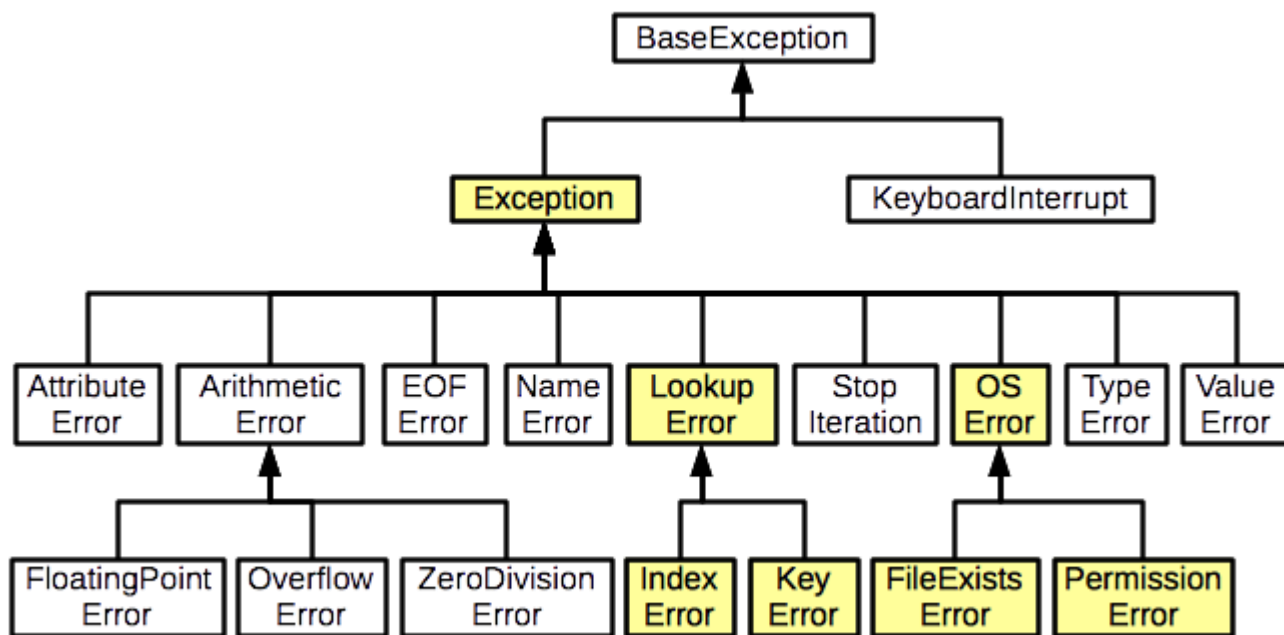
```

<generator object generator_example at 0x0000015886EC3BA0>

```

Python. Исключения (class Exception)

Во время выполнения программы могут возникать различные ошибки, которые в Python вызывают сработку так называемых исключений – специального типа данных, в котором передается тип ошибки, ее описание и трейс вызова инструкции, вызвавшей ошибку.



<https://betacode.net/11421/python-exception-handling>

Если возникшее исключение никак не обработать, это приведет к завершению выполнения программы.

Конструкция try-except нужна для обработки исключений. Базовый синтаксис конструкции:

try:

<код, в котором может произойти ошибка>

except *<один тип исключения>:*

<код, который выполнится при возникновении ошибки>

except *<другой тип исключения>:*

<код, который выполнится при возникновении ошибки>

else:

<код, который выполнится, если ошибки не было>

finally:

<код, который выполнится в любом случае> В функции выполняется до оператора return!

После выполнения этих инструкций программа продолжает работу.

- В блоке **except** можно обрабатывать несколько типов исключений, например, `except (ValueError, ZeroDivisionError)`
- Блоки **except** обрабатываются последовательно, по аналогии с конструкцией **if-elif-elif-else**.
- Исключения имеют иерархию, например `BaseException->Exception->ArithmeticError->ZeroDivisionError`. Блок **except ArithmeticError** будет перехватывать все ошибки, связанные с арифметическими ошибками. Если мы поставим после него блок **except ZeroDivisionError**, то он уже не выполнится, т.к. выполнится `except ArithmeticError`.
- Если после **except** не указывать никакое исключение, то блок будет перехватывать абсолютно все исключения.
- Для получения исключения в виде объекта можно использовать конструкцию `except <исключение> as <имя переменной>`. Указанная переменная будет ссылаться на экземпляр исключения, это может пригодиться, например, для вывода исключения в консоль

Распространение исключений (propagation exceptions) – механизм, при котором полученное **исключение** **распространяется на все уровни** вызова программы.

```
-----
ZeroDivisionError                                Traceback (most recent call last)
C:\Users\ASKORO~1\AppData\Local\Temp\ipykernel_54132\3163494175.py in <module>
    10 print('1')
    11 print('2')
--> 12 print(func3())
    13 print('3')

C:\Users\ASKORO~1\AppData\Local\Temp\ipykernel_54132\3163494175.py in func3()
     6
     7 def func3():
--> 8     func2()
     9
    10 print('1')

C:\Users\ASKORO~1\AppData\Local\Temp\ipykernel_54132\3163494175.py in func2()
     3
     4 def func2():
--> 5     func1()
     6
     7 def func3():

C:\Users\ASKORO~1\AppData\Local\Temp\ipykernel_54132\3163494175.py in func1()
     1 def func1():
--> 2     return 1/0
     3
     4 def func2():
     5     func1()

ZeroDivisionError: division by zero
```

```
1 def func1():
2     return 1/0
3
4 def func2():
5     func1()
6
7 def func3():
8     func2()
9
10 print('1')
11 print('2')
12 print(func3())
13 print('3')
```



```
1 def func1():
2     try:
3         return 1/0
4     except:
5         return 0
6
7 def func2():
8     try:
9         func1()
10    except:
11        return 0
12
13 def func3():
14     func2()
15
16 print('1')
17 print('2')
18 print(func3())
19 print('3')
```

1
2
None
3

Можно обрабатывать исключения на любом уровне стека вызова функций.

В коде можно самостоятельно генерировать исключения при помощи инструкции **raise**:

```

1 def some_function(a):
2     for i in range(a):
3         if i==3:
4             raise AttributeError('<Новое описание ошибки>')
5
6 some_function(10)

```

```

-----
AttributeError                                Traceback (most recent call last)
C:\Users\ASKORO~1\AppData\Local\Temp\ipykernel_54132\1171433506.py in <module>
      4             raise AttributeError('<Новое описание ошибки>')
      5
----> 6 some_function(10)

C:\Users\ASKORO~1\AppData\Local\Temp\ipykernel_54132\1171433506.py in some_function(a)
      2     for i in range(a):
      3         if i==3:
----> 4             raise AttributeError('<Новое описание ошибки>')
      5
      6 some_function(10)

AttributeError: <Новое описание ошибки>

```

Также можно создавать собственные классы исключений, которые должны наследоваться от класса `BaseException`.

1. Напишите собственную функцию `get_value_by_key()` для получения значения из словаря по ключу.
2. Функция должна принимать на вход словарь (`dictionary`), ключ (`key`) и значение (`def_value`, по умолчанию `None`), которое будет возвращаться, если ключ в словаре отсутствует. Т.е. вести себя схожим образом со стандартной функцией `dict.get ()`.
3. В случае, если ключ есть, то функция возвращает его значение.
4. В случае, если на вход функции в качестве `dictionary` передается не словарь, в консоль должно выводиться сообщение о недопустимом значении функции, а функция должна возвращать `None`.
5. В случае, если ключ `key` в словаре отсутствует, функция должна выводить в консоль сообщение, что такого ключа нет, но все равно возвращать значение, указанное в `def_value`.

1. Напишите функцию `check_value()`, которая будет проверять соответствие переданного ей значения определенным критериям и две функции `check_number()`, `check_str()`, в которых будет выполняться обработка исключений.
2. На вход функция `check_value()` будет принимать значение (`value`), а также неопределенное количество именованных аргументов.
3. В случае, если на вход пришло число и среди именованных аргументов есть аргумент `limits` (список из верхней и нижней границы разрешенного диапазона), должна вызываться функция `check_number()`.
4. В случае если на вход пришла строка и среди именованных аргументов есть аргумент `length` (максимальная длина строки), должна вызываться функция `check_str()`.
5. Функции `check_str()` и `check_number()` должны проверять строку и число на соответствие длине и диапазону соответственно, на вход им должно приходиться проверяемое значение и необходимые для проверки параметры. При превышении максимальной длины строки или выходе числа за диапазон функции должны генерировать исключение `AttributeError`.
6. При возникновении исключения в функции `check_value()`, исключение нужно обработать и вывести его текст в консоль.
7. В любом случае в конце выполнения функции она должна выводить в консоль сообщение о завершенной проверке, значение `value` и его тип.

ООП – методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования (Wikipedia).

Основные принципы ООП:

- **Полиморфизм** – разное поведение одного и того же метода в разных классах
- **Инкапсуляция** – ограничение доступа к методам и переменным объекта
- **Наследование** – наследование дочерним классом атрибутов родительского класса
- **Абстракция** – скрывание внутренних реализаций процесса или метода от пользователя

Основные понятия:

- **Класс** – модель для создания объектов определённого типа, описывающая их структуру.
- **Объект** – экземпляр класса.
- **Атрибут** – свойства (переменные), принадлежащие классу или объекту класса
- **Метод** – функции, принадлежащие классу или объекту класса

Python. Пример объявления и вызова объекта класса

```
class Country:

    def __init__(self, name, population, continent):
        self.name = name
        self.population = population
        self.continent = continent    АТРИБУТЫ

    def increase_population(self, value):    МЕТОД
        self.population += value
```

[3] ✓ 0.0s

```
ruusia = Country('Russia', 150_000_000, ['Asia', 'Europe'])
print(Country)
print(russia)
print("Население: ", russia.population)
ruusia.increase_population(99999)
print("Население: ", russia.population)
```

[8] ✓ 0.0s

```
<class '__main__.Country'>    КЛАСС
<__main__.Country object at 0x00000149CD8691C0>    ОБЪЕКТ
Население: 150000000
Население: 150099999    АТРИБУТ
```

Определение класса:

`class NewClass:`

`def method1(self, x):`

`self.attribute = x`

`def method2(self, ...):`

`<инструкции метода>`

`def method3(self, ...):`

`<инструкции метода>`

Создание объекта класса:

`class_object = NewClass()`

Вызов метода класса:

`class_object.method1(<аргументы>)`

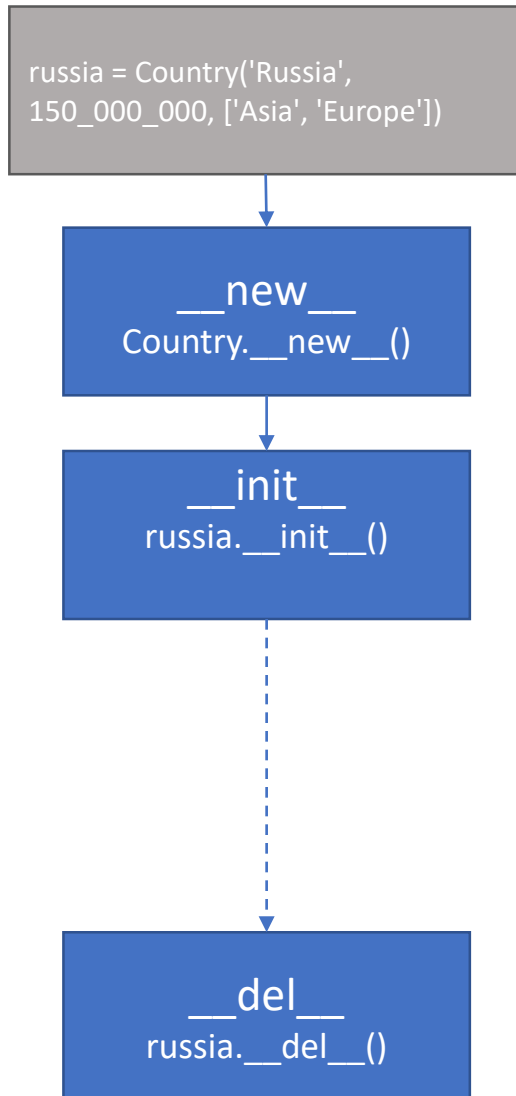
Доступ к атрибуту класса:

`class_object.attribute`

Python. Встроенные (магические, dunder) методы класса

Это далеко не все
методы!

Метод	Когда вызывается
<code>__new__(cls, [...])</code>	При создании объекта класса
<code>__init__(self, [...])</code>	При инициализации объекта
<code>__del__(self)</code>	Перед удалением объекта
<code>__eq__(self, other)</code>	При сравнении через оператор == (Операторы !=, <, > <=, >= : <code>__ne__</code> , <code>__lt__</code> , <code>__gt__</code> , <code>__le__</code> , <code>__ge__</code>)
<code>__call__(self, [args...])</code>	При вызове объекта
<code>__str__(self)</code>	При вызове через функцию <code>str(<объект>)</code>
<code>__abs__(self)</code>	При вызове через функцию <code>abs(<объект>)</code>
<code>__len__(self)</code>	При вызове через функцию <code>len(<объект>)</code>
<code>__setattr__(self, name, value)</code>	При изменении атрибута объекта (При обращении к атрибуту или его удалении: <code>__getattr__</code> , <code>__delattr__</code>)
<code>__add__(self, other)</code>	При сложении через оператор + (Операторы -, *, / : <code>__sub__</code> , <code>__mul__</code> , <code>__div__</code>)
<code>__hash__(self)</code>	При вызове через функцию <code>hash()</code>
<code>__next__(self)</code>	При вызове через функцию <code>hash()</code>



После объявления объекта класса:

- вызывается магический метод `__new__`, который возвращает ссылку на объект нового класса
- вызывается магический метод `__init__` для объекта класса, где выполняются инициализирующие процедуры
- происходит работа с объектом
- при отсутствии ссылок на объект или при вызове инструкции `del` вызывается магический метод `__del__`, после чего объект удаляется.

Для получения атрибутов объекта или класса можно воспользоваться методом `__dict__`.

```
print("russia:", russia.__dict__)
Country.__dict__
```

✓ 0.0s

```
russia: {'name': 'Russia', 'population': 150099999, 'continent': ['Asia', 'Europe']}
```

Атрибуты объекта

```
mappingproxy({'__module__': '__main__',
              'NAME': 'class_Country',
              '__init__': <function __main__.Country.__init__(self, name, population, continent)>,
              'increase_population': <function __main__.Country.increase_population(self, value)>,
              '__dict__': <attribute '__dict__' of 'Country' objects>,
              '__weakref__': <attribute '__weakref__' of 'Country' objects>,
              '__doc__': None})
```

Атрибуты класса

Получение значений атрибута:

При обращении к несуществующему атрибуту объекта этот объект будет искать в атрибутах класса.

При отсутствии атрибута в пространстве имен объекта или класса будет вызвано исключение.

Для получения значения атрибута объекта / класса можно использовать функцию `getattr()`.

Изменение значений атрибута:

При обращении к несуществующему атрибуту объекта будет создан новый атрибут объекта, даже если имя атрибута есть в пространстве имен класса.

Для изменения значения атрибута объекта / класса можно использовать функцию `setattr()`.

```
print(Country.NAME)
print(russia.NAME)
```

✓ 0.0s

```
class_Country
class_Country
```

```
russia.NAME = 'Russia'
russia.__dict__
```

✓ 0.0s

```
{'name': 'Russia',
 'population': 150099999,
 'continent': ['Asia', 'Europe'],
 'NAME': 'Russia'}
```

```
print(Country.NAME)
print(russia.NAME)
```

✓ 0.0s

```
class_Country
Russia
```

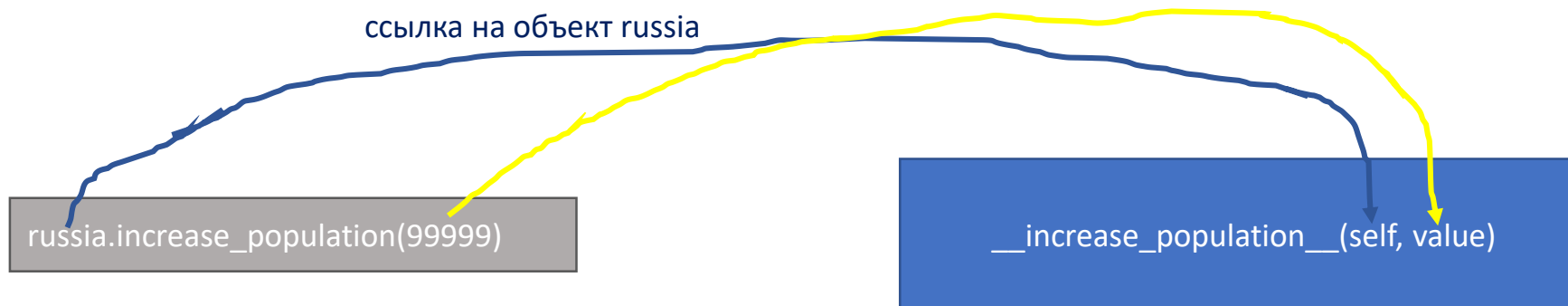
Метод	Описание
setattr (<класс/объект>, <имя атрибута>, <значение атрибута>)	Присваивание значения атрибуту класса/объекта
getattr (<класс/объект>, <имя атрибута>, <значение при отсутствии>)	Получение значения атрибута класса/объекта
delattr (<класс/объект>, <имя атрибута>)	Удаление атрибута класса/объекта
hasattr (<класс/объект>, <имя атрибута>)	Наличие указанного атрибута у класса/объекта. Возвращает True/False. В случае отсутствия атрибута у объекта, но присутствия его в класса все равно возвратит True.

Python. Вызов методов класса, параметр self

При определении методов класса первым параметром всегда идет параметр `self`, который **указывает на объект класса**.

При вызове метода класса интерпретатор автоматически передает в метод в качестве первого аргумента ссылку на объект, который вызвал этот метод.

Передача ссылки на объект, вызвавший метод, нужны для работы с объектом внутри метода.



Python. Статические методы и методы класса

	Статический метод	Метод класса
Ссылка на объект/класс	Не требуется	Нужна ссылка на класс
Доступ к атрибутам объекта	НЕТ	НЕТ
Доступ к атрибутам класса	НЕТ	ДА
Доступ внутри класса	через <code>self</code> .	через <code>self</code> .
Доступ извне	через <code><объект класса></code> . через <code><имя класса></code> .	через <code><объект класса></code> . через <code><имя класса></code> .
Декоратор	<code>@staticmethod</code>	<code>@classmethod</code>

В метод класса вместо обычной ссылки на объект класса (`self`)

передается ссылка на сам класс: `cls`

```
class Country:
    favorite_country = 'Russia'

    def __init__(self, name, population, continent):
        self.name = name
        self.population = population
        self.continent = continent

    def increase_population(self, value):
        self.population += value

    @classmethod
    def is_favorite(cls, country_name):
        return country_name in cls.favorite_country

    @staticmethod
    def calc_millions(population):
        return population / 1000000

russia = Country('Russia', 150_000_000, ['Asia', 'Europe'])
✓ 0.0s

print(russia.is_favorite('Russia'))
print(russia.calc_millions(5900000))
```

Python. Режимы доступа к атрибутам класса

В Python предусмотрено 3 режима доступа к объектам:

- **public (публичный)** – доступ извне разрешен. Имена публичных атрибутов не содержат «_» в начале: **attribute**
- **protected (защищенный)** – не ограничивает доступ, но предупреждает разработчика о том, что атрибут является защищенным и его не следует менять. Имена защищенных атрибутов начинаются с «_»: **_attribute**
- **private (локальный)** – доступ есть только внутри класса. Имена локальных атрибутов начинаются с «__»: **__attribute**

Режимы доступа обеспечивают один из принципов ООП – **инкапсуляцию**.

Правила наименования атрибутов справедливы как для переменных, так и для функций внутри класса. Для изменения режима доступа к функциям кроме подчеркиваний («_», «__») можно также использовать декораторы **@private** или **@protected** (предварительно нужно их импортировать: `from accessify import protected, private`).

На самом деле доступ к локальным атрибутам класса извне есть, но для этого нужно обращаться не по имени атрибута, а по кодовому имени «_**имя класса**>**имя атрибута**>».

```
class Country:
    favorite_country = 'Russia'

    def __init__(self, name, population, continent):
        self.name = name
        self.population = population
        self.continent = continent
        self.__private_attribute = 'пример локального атрибута'
        self._protected_attribute = 'пример защищенного атрибута'
```

```
russia._protected_attribute = россия._protected_attribute + ' да пофиг'
✓ 0.0s
```

```
russia.__dict__
✓ 0.0s
{'name': 'Россия',
 'population': 150000000,
 'continent': ['Азия', 'Европа'],
 '_Country__private_attribute': 'пример локального атрибута',
 '_protected_attribute': 'пример защищенного атрибута да пофиг'}
```

Python. Доступ к атрибутам класса. Сеттеры и геттеры

Для изменения и получения значений локальных атрибутов рекомендуется использовать специальные методы – сеттеры (изменение атрибута) и геттеры (получение значения). Можно их создавать как отдельные методы, например, для переменной `x`: `set_x()` и `get_x()`, но тогда придется обращаться к различным методам для получения и изменения атрибута. Также можно использовать специальный декоратор `@property`, который позволяет обращаться к сеттеру и геттеру по названию переменной. Для этого геттер называется по имени локальной переменной, а к геттеру добавляется декоратор `@property`. Сеттер также называется по имени переменной и к нему добавляется декоратор `@<имя геттера>.setter`.

```
class Country:
    favorite_country = 'Russia'

    def __init__(self, name, population, continent):
        self.name = name
        self.population = population
        self.continent = continent
        self.__private_attribute = 'пример локального атрибута'
        self._protected_attribute = 'пример защищенного атрибута'

    @property
    def private_attribute(self):
        return self.__private_attribute

    @private_attribute.setter
    def private_attribute(self, value):
        self.__private_attribute = value
```

```
print(russia.private_attribute)
russia.private_attribute = "его можно менять"
russia.private_attribute
```

✓ 0.0s

пример локального атрибута

'его можно менять'

`__setattr__(self, name, value)`

Данный метод вызывается при каждом изменении атрибута класса. Это можно использовать, например, для проверки передаваемых значений или если нужно запретить создание атрибутов с определенным именем. Если переопределять этот метод, то он должен возвращать `object.__setattr__(self, key, value)`, иначе атрибут не будет изменен.

```
class Country:
    favorite_country = 'Russia'

    def __setattr__(self, key, value):
        if key == 'id':
            if value != 'id':
                return

        return object.__setattr__(self, key, value)

    def __init__(self, name, population, continent):
        self.id = 'id'
        self.name = name
```

```
russia.id = 'idd'
russia.id
```

✓ 0.0s

'id'

Python. Магические методы. `__call__`, `__str__`, `__eq__`

`__call__(self, [args...])`

Данный метод вызывается при обращении к объекту как к методу (`()`). Классы, которые могут себя вести как функции называются функторы. Это может применяться, например для использования класса в качестве декоратора.

`__str__(self)`

Данный метод вызывается при обращении к объекту через функцию `str()`.

`__eq__(self, other)`

Данный метод вызывается при сравнении объекта с другим объектом (оператор `==`). При определении метода `__eq__` обычно также будет возможно применение к объектам оператора `!=`.

```
class Country:
    favorite_country = 'Russia'

    def __eq__(self, other):
        return self.name == other.name

    def __call__(self, *args, **kwargs):
        print(f'{self.name} нуждается в вас!')

    def __str__(self):
        return f"Страна {self.name} с населением {self.population}"
```

```
ruusia == Country('Russia', 150, ['Asiaaa', 'Europe'])
✓ 0.0s

True

print(russia)
ruusia()
✓ 0.0s

Страна Russia с населением 150000000
Russia нуждается в вас!
```

Наследование – механизм создания класса на основе другого существующего класса.

Основные понятия:

- **Базовый (родительский) класс** – класс, от которого производится наследование
- **Подкласс (дочерний класс)** – класс, который наследуется

В подклассе доступны все атрибуты и методы родительского класса.

В подклассе возможно переопределить атрибуты и методы родительского класса.

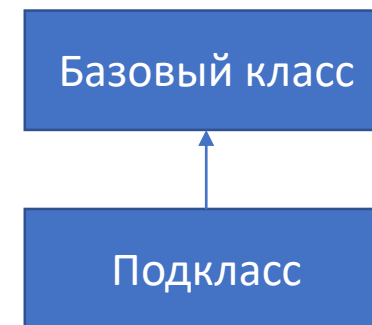
Все объекты в Python в конечном итоге наследуются от базового класса **object**.

Для определения того, что какой-либо класс является подклассом другого класса, можно использовать функцию `issubclass(<подкласс>, <базовый класс>)`.

Для определения того, что какой-либо класс/объект наследуется от другого класса, можно использовать функцию `isinstance(<подкласс/объект>, <базовый класс>)`.

Если подкласс переопределяет атрибуты базового класса – **переопределение (overriding) класса**.

Если подкласс дополняет атрибуты базового класса – **расширенный (extended) класс**.



```

print(isinstance(russia, object))
print(isinstance(russia, Country))
print(issubclass(Country, object))
print(issubclass(russia, object))

```

⊗ 0.0s

```

True
True
True

```

```

TypeError                                Traceback
C:\Users\ASKORO~1\AppData\Local\Temp\ipykernel_605
  2 print(isinstance(russia, Country))
  3 print(issubclass(Country, object))
----> 4 print(issubclass(russia, object))

TypeError: issubclass() arg 1 must be a class

```

Python. Делегирование и функция super()

Если в подклассе переопределен метод базового класса, то при вызове метода будет вызван метод подкласса. Это касается в т.ч. магических методов.

При определении в подклассе метода `__init__` метод инициализатор базового класса не будет вызван без явной инструкции, поэтому, если нам нужно вызвать инициализатор базового класса, в инициализаторе подкласса необходимо использовать метод `super()`, который ссылается на базовый класс:

`super().__init__(<аргументы>)`. Причем его нужно вызывать в самом начале инициализации! Вызов методов базового класса через функцию `super()` называется **делегированием**.

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

class B(A):
    def __init__(self, x, y, z):
        #super().__init__(x, y)

        self.z = z
```

b = B(1, 2, 3)
 b.__dict__
 ✓ 0.0s
 {'z': 3}

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

class B(A):
    def __init__(self, x, y, z):
        super().__init__(x, y)

        self.z = z
```

b = B(1, 2, 3)
 b.__dict__
 ✓ 0.0s
 {'x': 1, 'y': 2, 'z': 3}

Полиморфизм – возможность работы единым образом с различными объектами, через единый интерфейс.

Для создания единого интерфейса необходимо, чтобы методы интерфейса были определены во всех классах.

Для реализации этого требования можно воспользоваться абстрактными методами, которые определить в базовом классе.

Абстрактный метод – метод, который не содержит реализации. Для реализации абстрактного метода в Python есть декоратор **abstractmethod** модуля **abc** (from abc import ABC, abstractmethod).

```
from abc import ABC, abstractmethod

class A(ABC):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @abstractmethod
    def do_something(self):
        pass

class B(A):
    def __init__(self, x, y, z):
        super().__init__(x, y)

        self.z = z
```

```
b = B(1, 2, 3)
```

⊗ 0.0s

```
-----
TypeError                                 Traceback (most recent call last)
C:\Users\ASKORO~1\AppData\Local\Temp\ipykernel_...
----> 1 b = B(1, 2, 3)

TypeError: Can't instantiate abstract class B with
```

```
class A(ABC):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @abstractmethod
    def do_something(self):
        pass

class B(A):
    def __init__(self, x, y, z):
        super().__init__(x, y)

        self.z = z
    def do_something(self):
        print('okay')
```

```
b = B(1, 2, 3)
b.do_something()
```

✓ 0.0s

okay

Пока все.