

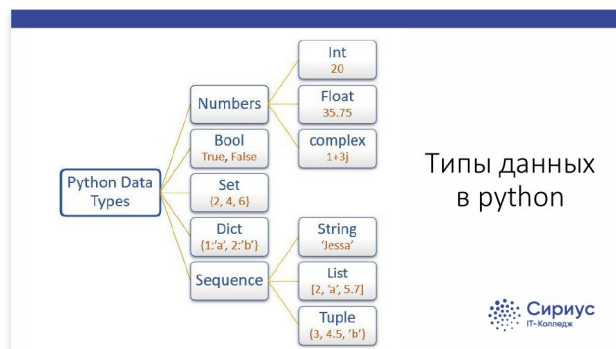


# Основы программирования

## BASH

## ТИПЫ ДАННЫХ PYTHON

- list
- tuple
- string (str)
- float
- integer (int)
- dict
- bool (True / False)
- Set (list of numbers that don't repeat themselves)



## КОММЕНТАРИИ

sharp #, " "

## ХРАНИТЕ БАЗОВЫЕ ТИПЫ ДАННЫХ В ПАМЯТИ

- hasjhx

- jashxj
- 

## СТРОКИ И ОПЕРАЦИИ СО СТРОКАМИ

- сложение (конкатенация)
- сравнение
- замена (replace) — заменяет одну строку на другую
- обращение по индексу

```
s = 'abcde'  
s[0]  
s = a
```

- интерполяция (f строки — f{\_\_\_\_})

### ФУНКЦИИ:

- str(n) — преобразует любое значение в строку
- len(n) — считает кол-во элементов в строке

## Методы для работы со строками

Кроме функций, для работы со строками есть немало методов:

- `find(s, start, end)` — возвращает индекс первого вхождения подстроки в `s` или `-1` при отсутствии. Поиск идет в границах от `start` до `end` ;
- `rfind(s, start, end)` — аналогично, но возвращает индекс последнего вхождения;
- `replace(s, new)` — меняет последовательность символов `s` на новую подстроку `new` ;
- `split(x)` — разбивает строку на подстроки при помощи выбранного разделителя `x`;
- `join(x)` — соединяет строки в одну при помощи выбранного разделителя `x`;
- `strip(s)` — убирает пробелы с обеих сторон;
- `lstrip(s), rstrip(s)` — убирает пробелы только слева или справа;
- `lower()` — перевод всех символов в нижний регистр;
- `upper()` — перевод всех символов в верхний регистр;
- `capitalize()` — перевод первой буквы в верхний регистр, остальных — в нижний.

| join, find, replace, split

---

## СЛОВАРИ

- `dict.keys` — выводит ключи (первый столбик **a =**)
- `dict.values` — вывод значений (второй столбик **a = 3**)
- `dict.items` (выводит все)

key = value pairs



**СЛОВАРЬ** — упорядоченный список элементов, где ни один не повторяется и элементы можно изменять.

Пример словаря:

```
https://www.notion.so/89d8670535f349468d19186cda1b37c5BarbieLand
    "Barbie": "protagonist",
    "Ken": "unsuccessful boyfriend"
}
a = BarbieLand.items()
print(a)
```

```
c = {
    'a': 1,
    'b': 2,
    'b': 4,
    'c': 4
}
#{'a': 1, 'b': 4, 'c': 4}
#если два одинаковых ключа
```

---

## МНОЖЕСТВА



**Множество** — неупорядоченная последовательность элементов, каждый из которых представлен один раз.

- объединение + поиск пересечений

```
a = [1, 2, 3, 4]
b = [1, 5, 6]

c = list(set(a) & set(b)) # пересечение
d = list(set(a) | set(b)) # объединение (union)
```

```
print(c)
print(d)
```

---

## ЦИКЛЫ

| while \_\_ , for



ЦИКЛЫ — программные конструкции, выполняющие определённые действия до тех пор, пока выполняется заданное условие.

Цикл в Python объявляется ключевыми словами `for` и `in`; после объявления ставится двоеточие. Ниже объявления пишут **тело цикла** — код, который описывает, что же нужно сделать с каждым элементом списка.

```
for переменная in список_элементов: # Вот оно, объявление цикла
# Тут будет тело цикла.
```

Имя переменной в цикле вы можете дать любое, но традиционно эти имена образуют от имени обрабатываемого списка, в единственном числе.

Например, если список называется `musicians`, то переменную лучше назвать `musician`; если список называется `pigs` — переменную называют `pig`.

for переменная in список\_элементов:

# Тут тело цикла: код, который выполняется для каждого элемента

# Здесь можно обработать переменную, объявленную в условии цикла,

# например, напечатать её значение: print(переменная)



```
bremen_musicians = ['Кот', 'Пёс', 'Трубадур', 'Осёл', 'Петух']
```

```
for musician in bremen_musicians:
```

```
# Каждый элемент списка bremen_musicians
```

```
# по очереди будет передан в переменную musician
```

```
# и напечатан
```

```
print(musician)
```



Цикл берёт значение первого элемента из списка `bremen_musicians` и передаёт его в переменную `musician`. Затем выполняется код в теле цикла: печатается содержимое переменной `musician`.

Каждый такой «круг» называется **итерацией цикла**.

Когда список закончится — программа выйдет из цикла; после этого сработает код, который написан после цикла.

---

## ФУНКЦИИ

Вопрос: (Функции. Внутренние функции. Замыкания. Лямбды.)

**Функция** — это именованный блок кода, выполняющий определённую задачу. Код функции можно использовать многократно, надо лишь вызвать её — обратиться к ней по имени.

В Python есть множество заготовленных («встроенных») функций, некоторые вы уже вызывали: `print()`, `str()`, `int()`, `float()`, `len()`.

Но можно создавать и собственные функции.

Всё начинается с **объявления функции**, со строки, которая означает «здесь мы создаём новую функцию»

```
def hello():  
    # А здесь началось тело функции  
    print('Приветствую тебя, джедай Питона!')
```

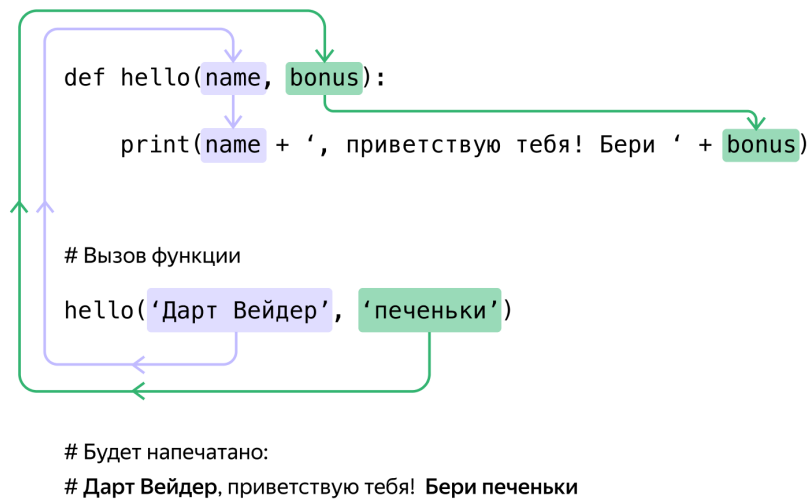
Пока функция не вызвана — она не выполняется: она просто лежит и ждёт своего часа!

```
def hello(name):  
    # Началось тело функции, с четырьмя отступами  
    print(name + ', приветствую тебя!')  
    # В теле функции может быть ещё много кода  
    ...  
    ...  
  
# Тело функции кончилось, когда начался код без отступов.  
  
# Вызов функции:  
hello('Максим')  
  
# Будет напечатано: Максим, приветствую тебя!
```

- ① ключевое слово
- ② имя функции
- ③ параметры функции, в скобках
- двоеточие обязательно

● аргумент, передаваемый при вызове функции

КАК ПИСАТЬ?!



## ПРИМЕР КОДА:

```
resorts = ['Сочи', 'курорты Краснодар', 'Санкт-Петербург']  
def choose_vacation_place(resorts):  
    for resort in resorts:  
        if resort == 'Сочи':  
            return resort  
resort = choose_vacation_place(resorts)  
print('Поехали в ' + resort)
```

Вложенные функции — это функции внутри других функций. Они могут быть использованы для повышения структурированности кода, инкапсуляции и реализации декораторов.

```
def outer_function():  
    print("Я внешняя функция")  
  
    def inner_function():  
        print("Я внутренняя функция")  
  
    inner_function()
```



```
outer_function()
```



**LAMBDA** — ключевое слово, которое определяет анонимную функцию (функция, вызываемая один)

```
double = lambda x: x*2
```

Эквивалентна:

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

В вышеуказанном коде `lambda x: x*2` — это лямбда-функция. Здесь `x` — это аргумент, а `x*2` — это выражение, которое вычисляется и возвращается.

Эта функция безымянная. Она возвращает функциональный объект с идентификатором `double`.



lambda-функция может использоваться вместе с другими функциями:

Часто используют `map()`, чтобы лямбда применилась ко всем элементам списка, как в данном случае

```
ss = list(map(lambda x: x*2, a))  
print(ss)
```

В статье ещё говорили про `filter()`

```
my_list = [1, 3, 4, 6, 10, 11, 15, 12, 14]  
new_list = list(filter(lambda x: (x%2 == 0) , my_list))  
print(new_list)  
#Выводит чётные числа
```

Функция `filter()` в Python применяет другую функцию (в данном случае лямбду) к заданному итерируемому объекту (список, строка, словарь и так далее), проверяя, нужно ли сохранить конкретный элемент или нет. Простыми словами, она отфильтровывает то, что не проходит и возвращает все остальное.

```
# кстати, вот та же функция, написанная через def
def filter_odd_num(in_num):
    if(in_num % 2) == 0:
        return True
    else:
        return False

out_filter = filter(filter_odd_num, numbers)
```

Функция `reduce()` применяет определённую операцию ко всем элементам итерируемого объекта (списка)

```
from functools import reduce
# это нужно импортировать, значит это использовать нам скорее всего
# но просто для общего развития, почему бы и нет

current_list = [5, 15, 20, 30, 50, 55, 75, 60, 70]
summa = reduce((lambda x, y: x + y), current_list)
print(summa)
```

## РЕКУРСИВНЫЕ ФУНКЦИИ

Если коротко, то рекурсивная функция, которая в процессе выполнения обращается к себе

Например, вот функция, которая пошагово выполняет что-то

```
def summa(n):
    x = 0
    for n in range(1, n+1):
```

```
x += n  
return x
```

А вот функция, которая делает это что-то, но через рекурсию

```
def summa(n):  
    if n == 1:  
        return 1  
    return n + summa(n-1)
```

Если в первом варианте мы просто на каждой итерации цикла прибавляем следующее из промежутка число, то в рекурсии мы не просто прибавляем число, мы прибавляем число к предыдущему значению функции, а не переменной



summa(5) — то же самое, что 5 + summa(4)  
summa(4) — то же самое, что 4 + summa(3)  
summa(3) — то же самое, что 3 + summa(2)  
summa(2) — то же самое, что 2 + summa(1)  
summa(1) — это 1

Всё, что я помню из алгоритмов, так это то, что не надо помещать рекурсию в цикл while, иначе код никогда не закончится

## ГЕНЕРАТОРЫ

Генератор генерирует значения элементов, но хранит в памяти только последнее из них

Вычисление следующего элемента происходит через next()

В общем, генераторы неплохо так экономят память

```
40
41 v = (i**2 for i in range(1, 5))
42 print(next(v))
43 print(next(v))
44 print(next(v))

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
1
user@shroooms:~/exams$ /bin/python3.11 /home/user/exams/dictionary.py
1
4
9
user@shroooms:~/exams$
```

В Функциях-генераторах вместо return используется yield, она не останавливает функцию, как return, а лишь приостанавливает

```
def f_gen(m):
    s = 1
    for n in range(1,m):
        yield n**2 + s
        s += 1

ge = f_gen(5)
for i in ge:
    print(i)
```

Есть ещё этот кусок кода, он в конце концов бесконечность выводит

```
def iterate(x0, m):
    x = x0
    while True:
        yield x
        x *= m

i = iterate(1, 1.1)
for j in i:
    print(j)
```

А это было на контрольной:

```
def task_4(n: int, step: int = 1):
    if step == 0:
```

```

        if n == 0:
            yield 0
        else:
            raise Exception('всё плохо')
    if n * step < 0:
        step *= -1

    for i in range(0, n, step):
        yield i

print([i for i in task_4(7, 2)])

```

И ещё кусок кода, который у меня сохранён

```

def parity(n: int, even: bool = True):
    start = 0 if even else 1
    step = 2
    if n < 0:
        start, step = -start, -step
        n += 1 if n > 0 else -1
    for i in range(start, n, step):
        yield i

for i in parity(23):
    print(i)

```

У генераторов есть ещё методы `close()`, `throw()`, `send()`, но мы их вроде не использовали

## ДЕКОРАТОРЫ

Функция, которая добавляет дополнительный функционал к другой функции или классу

Что бы применить декоратор к функции, мы его через @ пишем перед функцией

Вот код с кр (декоратор, который выводит кол-во позиционных и ключевых элементов функции)

```
def task_5(func):
    def new(*args, **kwargs):
        print(f'позиционных элементов: {len(args)}')
        print(f'ключевых аргументов: {len(kwargs.items())}')
        return func(*args, **kwargs)
    return new

@task_5
def f(a, b, x, c, d):
    return a + b

# print(f(2,5, x=0, c=1, d =1 ))
```

И простенький код, который я для себя написала. В общем, в декораторе основная функция принимает функцию, а внутренняя принимает позиционные и ключевые аргументы(\*args \*\*kwargs(если декорируемая функция принимает аргументы)) Здесь она просто пишем привет и выполняет декорируемую функцию

```
def dec(func):
    def wrapper(*args, **kwargs):
        print('Hello')
        return func(*args, **kwargs)
    return wrapper

@dec
def summ(x, y):
    return x + y
```

```
print(summ(1, 2))
```

Вот ещё примеры декораторов

```
def uppercase(func):  
    def wrapper():  
        original_result = func()  
        modified_result = original_result.upper()  
        return modified_result  
    return wrapper  
  
@uppercase  
def greet():  
    return 'Hello!'
```

## ПРОСТРАНСТВО ИМЁН. ГЛОБАЛЬНЫЕ И ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ



Пространства имён в Python — это способ организации переменных, функций, классов и других объектов в коде. Они помогают избегать конфликтов имен и делают код более читаемым и понятным

Ну тут про то, что мы можем объявить глобальные переменные, которые можно использовать во всём модуле, а есть переменные, которые мы объявляем например внутри цикла или функции и можем их использовать только там.

```
a = 1 # глобальная переменная  
def f(x):  
    return x # локальная переменная
```

## ОБРАБОТКА ОШИБОК И ИСКЛЮЧЕНИЯ

Обработка исключений — это процесс написания кода для перехвата и обработки ошибок или исключений, которые могут возникать при выполнении программы.

Что происходит в коде ниже:

у нас блок, который выдаёт ошибку

```
41
42 c = 1/0
43 print(c)
44
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

Traceback (most recent call last):  
File "/home/user/exams/decorator.py", line 42, in <module>  
c = 1/0  
ZeroDivisionError: division by zero  
user@shroooms:~/exams\$

поэтому суём его под try и вылавливаем ошибку, вместо неё делаем что-то другое

```
42 try:
43     c = 1/0
44 except ZeroDivisionError:
45     print([0])
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

user@shroooms:~/exams\$ /bin/python3.11 /home/user/exams/decorator.py  
[0]  
user@shroooms:~/exams\$

Либо ловим все ошибки

```
try:
    c = 1/0
except Exception as e:
    print('ахахахахаха')
```

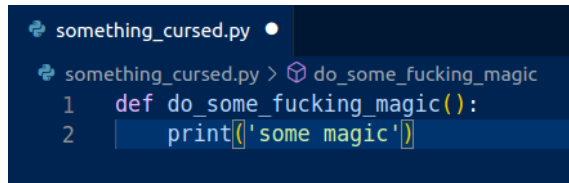
## МОДУЛИ

Модуль в языке *Python* представляет отдельный файл с кодом, который можно повторно использовать в других программах.

Ну короче:

Создаём кусок кода



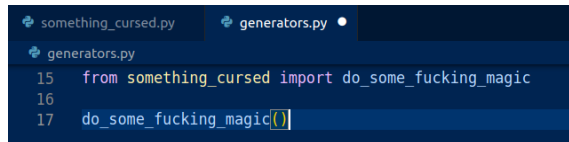


```

something_cursed.py
something_cursed.py > do_some_fucking_magic
1 def do_some_fucking_magic():
2     print('some magic')

```

Теперь мы можем импортировать этот кусок в другой кусок кода и использовать его там



```

something_cursed.py  generators.py
generators.py
15 from something_cursed import do_some_fucking_magic
16
17 do_some_fucking_magic()

```

## ПАКЕТНЫЙ МЕНЕДЖЕР

pip

## МОДУЛИ ДЛЯ РАБОТЫ С ОПЕРАЦИОННОЙ СИСТЕМОЙ, ТЕРМИНАЛЬНЫМИ КОМАНДАМИ, ВРЕМЕНЕМ И ДАТАМИ

os, sh, datetime

### os

Что-то делает с операционкой, вот её методы

Будьте внимательны: некоторые функции из этого модуля поддерживаются не всеми ОС.

**os.name** - имя операционной системы. Доступные варианты: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.

**os.environ** - словарь переменных окружения. Изменяемый (можно добавлять и удалять переменные окружения).

**os.getlogin()** - имя пользователя, вошедшего в терминал (Unix).

**os.getpid()** - текущий id процесса.

**os.uname()**

- информация об ОС. возвращает объект с атрибутами: sysname - имя операционной системы, nodename - имя машины в сети (определяется

реализацией), `release` - релиз, `version` - версия, `machine` - идентификатор машины.

**`os.access`**(`path`, `mode`, \*, `dir_fd=None`, `effective_ids=False`, `follow_symlinks=True`) - проверка доступа к объекту у текущего пользователя. Флаги:

**`os.F_OK`** - объект существует, **`os.R_OK`** - доступен на чтение, **`os.W_OK`** - доступен на запись, **`os.X_OK`** - доступен на исполнение.

**`os.chdir`**(`path`) - смена текущей директории.

**`os.chmod`**(`path`, `mode`, \*, `dir_fd=None`, `follow_symlinks=True`) - смена прав доступа к объекту (`mode` - восьмеричное число).

**`os.chown`**(`path`, `uid`, `gid`, \*, `dir_fd=None`, `follow_symlinks=True`) - меняет id владельца и группы (Unix).

**`os.getcwd`**() - текущая рабочая директория.

**`os.link`**(`src`, `dst`, \*, `src_dir_fd=None`, `dst_dir_fd=None`, `follow_symlinks=True`) - создаёт жёсткую ссылку.

**`os.listdir`**(`path="."`) - список файлов и директорий в папке.

**`os.mkdir`**(`path`, `mode=0o777`, \*, `dir_fd=None`) - создаёт директорию. `OSError`, если директория существует.

**`os.makedirs`**(`path`, `mode=0o777`, `exist_ok=False`) - создаёт директорию, создавая при этом промежуточные директории.

**`os.remove`**(`path`, \*, `dir_fd=None`) - удаляет путь к файлу.

**`os.rename`**(`src`, `dst`, \*, `src_dir_fd=None`, `dst_dir_fd=None`) - переименовывает файл или директорию из `src` в `dst`.

**`os.rename`**(`old`, `new`) - переименовывает `old` в `new`, создавая промежуточные директории.

**`os.replace`**(`src`, `dst`, \*, `src_dir_fd=None`, `dst_dir_fd=None`) - переименовывает из `src` в `dst` с принудительной заменой.

**`os.rmdir`**(`path`, \*, `dir_fd=None`) - удаляет пустую директорию.

**`os.removedirs`**(`path`) - удаляет директорию, затем пытается удалить родительские директории, и удаляет их рекурсивно, пока они пусты.

**os.symlink**(source, link\_name, target\_is\_directory=False, \*, dir\_fd=None) - создаёт символическую ссылку на объект.

**os.sync()** - записывает все данные на диск (Unix).

**os.truncate**(path, length) - обрезает файл до длины length.

**os.utime**(path, times=None, \*, ns=None, dir\_fd=None, follow\_symlinks=True) - модификация времени последнего доступа и изменения файла. Либо times - кортеж (время доступа в секундах, время изменения в секундах), либо ns - кортеж (время доступа в наносекундах, время изменения в наносекундах).

**os.walk**(top, topdown=True, onerror=None, followlinks=False) - генерация имён файлов в дереве каталогов, сверху вниз (если topdown равен True), либо снизу вверх (если False). Для каждого каталога функция walk возвращает кортеж (путь к каталогу, список каталогов, список файлов).

**os.system**(command) - исполняет системную команду, возвращает код её завершения (в случае успеха 0).

**os.urandom**(n) - n случайных байт. Возможно использование этой функции в криптографических целях.

os.path - модуль, реализующий некоторые полезные функции на работы с путями.

## SH

Позволяет выполнять терминальные команды через питон

## datetime

Создаёт объекты datetime для обработки даты и времени

```
15
16 import datetime
17 now=datetime.datetime.today()
18 print(now)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
user@shroooms:~/exams$ /bin/python3.11 /home
2024-01-11 01:24:24.793074
user@shroooms:~/exams$
```

## РАБОТА С ФАЙЛАМИ

Питон может работать с файлами

Открыть файл можно с помощью функции `open()`.

```
1. open(file, mode='rt')
```

В функцию в качестве аргументов требуется передать путь файлу (`file`) и выбрать режим работы (`mode`). По умолчанию Python выбирает значение `rt`, но доступны и другие режимы:

Аргумент <code>mode</code>	Как работает
<code>r</code>	Чтение из файла
<code>t</code>	Открыть как текстовый файл
<code>w</code>	Запись в файл и создание файла, если его не существует
<code>x</code>	Запись в файл и вызовы исключения, если файла не существует
<code>b</code>	Открыть как двоичный файл
<code>a</code>	Запись в файл путем добавления новых значений в конец
<code>+</code>	Работа в режиме чтения и записи

```
with open('file.txt', 'r', encoding='utf-8') as f:
    data = f.read(6)
print(data)
```

Если запустить код, то Python выведет в консоль фразу «Привет» — это и есть первые шесть символов строки «Привет, Python!» в file.txt

## JSON, CSV, XML

AAXXAXAXAXAXAXAXAXAXAXAXAXAXAXAXAXA

Ну там import json/csv/xml и уже потом смотреть, что с ними делать

## ОТЛАДКА КОДА И ЛОГИРОВАНИЕ

идёт в попу

Тестирование кода, pytest

Долго писать, я лучше так расскажу

Оформление кода. Docstrings

flake8, линтер на гитхабе

докстринги пишутся в тройных кавычках """ """

А ещё есть вот эта штука

