

Функции в Python

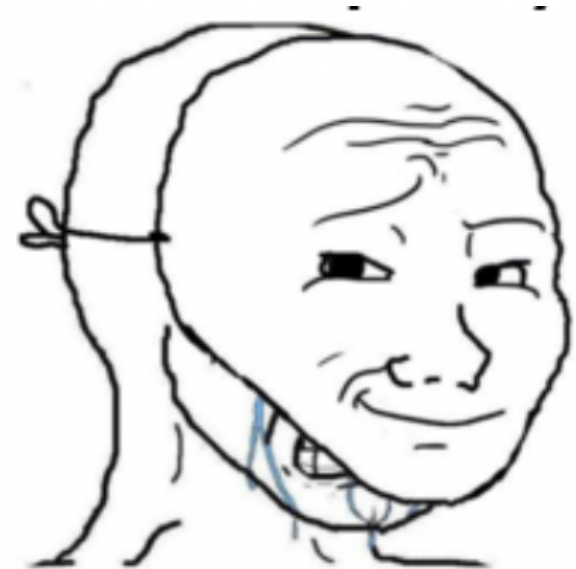
Тёма, лекция на ФиКЛ 21.01.2025

**Когда случайно объявил
переменную float, а не int**



Почему функции в программировании это важно

Ситуация: Иннокентий написал вот такой код



↑
Иннокентий

```
1  s = "Gimme a ticket for an aeroplane"  
2  print("".join(word.upper() for word in s.split()))  
3
```

↗
Код

- Что произойдет при его выполнении?
- Вспоминаем: какую роль здесь играют join и split? Что делает split, если ему не передать никакого параметра?

Почему функции в программировании это важно

История развивается: Иннокентия просят проделать то же самое еще с другой строкой...



```
1  s = "Gimme a ticket for an aeroplane"
2  print("".join(word.upper() for word in s.split()))
3
4  t = "Ain't got time to take a fast train"
5  print("".join(word.upper() for word in t.split()))
```


Почему функции в программировании это важно

...ну и еще с двумя....



```
1  s = "Gimme a ticket for an aeroplane"
2  print("".join(word.upper() for word in s.split()))
3
4  t = "Ain't got time to take a fast train"
5  print("".join(word.upper() for word in t.split()))
6
7  w = "Lonely days are gone, I'm a-goin' home"
8  print("".join(word.upper() for word in t.split()))
9
10 y = "My baby, just a wrote me a letter"
11 print("".join(word.upper() for word in y.split()))
```

Уважаемые знатоки, где облажался
Иннокентий?

Почему функции в программировании это важно

Ну и напоследок, приходит заказчик и говорит, что все плохо и слова надо было склеить через пробел, идем менять код в 4 местах



```
1  s = "Gimme a ticket for an aeroplane"
2  print(" ".join(word.upper() for word in s.split()))
3
4  t = "Ain't got time to take a fast train"
5  print(" ".join(word.upper() for word in t.split()))
6
7  w = "Lonely days are gone, I'm a-goin' home"
8  print(" ".join(word.upper() for word in w.split()))
9
10 y = "My baby, just a wrote me a letter"
11 print(" |".join(word.upper() for word in y.split()))
```

Почему функции в программировании это важно

А могло быть все так



```
1  def capitalize_sentence(original):
2      return " ".join(word.upper() for word in original.split())
3
4  s = "Gimme a ticket for an aeroplane"
5  t = "Ain't got time to take a fast train"
6  w = "Lonely days are gone, I'm a-goin' home"
7  y = "My baby, just a wrote me a letter"
8
9  print(capitalize_sentence(s))
10 print(capitalize_sentence(t))
11 print(capitalize_sentence(w))
12 print(capitalize_sentence(y))
```

А можно еще красивее?

Можно!



Вопрос: чем отличается это

```
def capitalize_sentence(original):  
    return "".join(word.upper() for word in original.split())  
  
for sentence in [s, t, w, y]:  
    print(capitalize_sentence(sentence))
```

ОТ ЭТОГО

```
for sentence in [s, t, w, y]:  
    print("".join(word.upper() for word in sentence.split()))
```

Так зачем нам это все?

- В программировании вам часто придется писать какие-то повторяющиеся куски кода, отличающиеся лишь тем, что находится в переменных, но не логикой
- Если делать копипасту, то есть шанс посадить багу и потом чинить ее в 10 разных местах
- Ну, или даже просто поменять логику будет уже затруднительно
- Кроме того, "простыня" такого скопированного кода вероятно будет не одна, понять это спустя время будет затруднительно
- Читабельность: функция имеет понятное название, не надо в нее вникать, чтобы понять, что она делает

Чуть подробнее

"def" говорит питону,
что это функция

в функцию можно передать
какие-то параметры, ака аргументы

```
def capitalize_sentence(original):  
    capitalized_words = [word.upper() for word in original.split()]  
    return " ".join(capitalized_words)
```

функция может
возвращать что-то.
если ничего не указано,
то вернет None

внутри функции может
находиться какой-то код

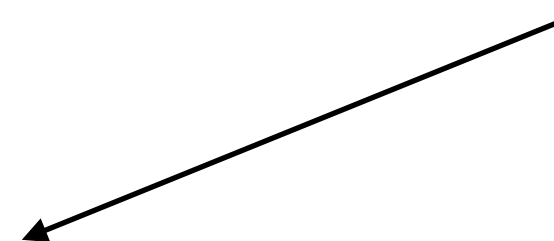
Чуть подробнее

```
1  ✓ def capitalize_sentence(original):  
2      return " ".join(word.upper() for word in original.split())  
3  
4      s = "Gimme a ticket for an aeroplane"  
5  
6      result = capitalize_sentence(s)  
7  
8      print(result.islower())  
9
```

Результат работы функции можно присваивать в переменную

Чуть подробнее

У функции могут быть аргументы
со значением по умолчанию,
если его не указать явно, то возьмется то,
что указано в функции



```
1 def capitalize_and_join(original, separator=" "):
2     | return separator.join(word.upper() for word in original.split())
3
4 print(capitalize_and_join("Над костром")) # склеит через пробел
5 print(capitalize_and_join("пролетает снежинка", separator=".")) # а тут через точку
6 print(capitalize_and_join("как огромный седой вертолет", ",")) # и так тоже будет работать
7
```

Нюансы

- Функция может быть вообще без аргументов
- Может ничего не возвращать
- Может быть аргументом другой функции))

```
1  def foo():  
2      return "hello world"  
3
```

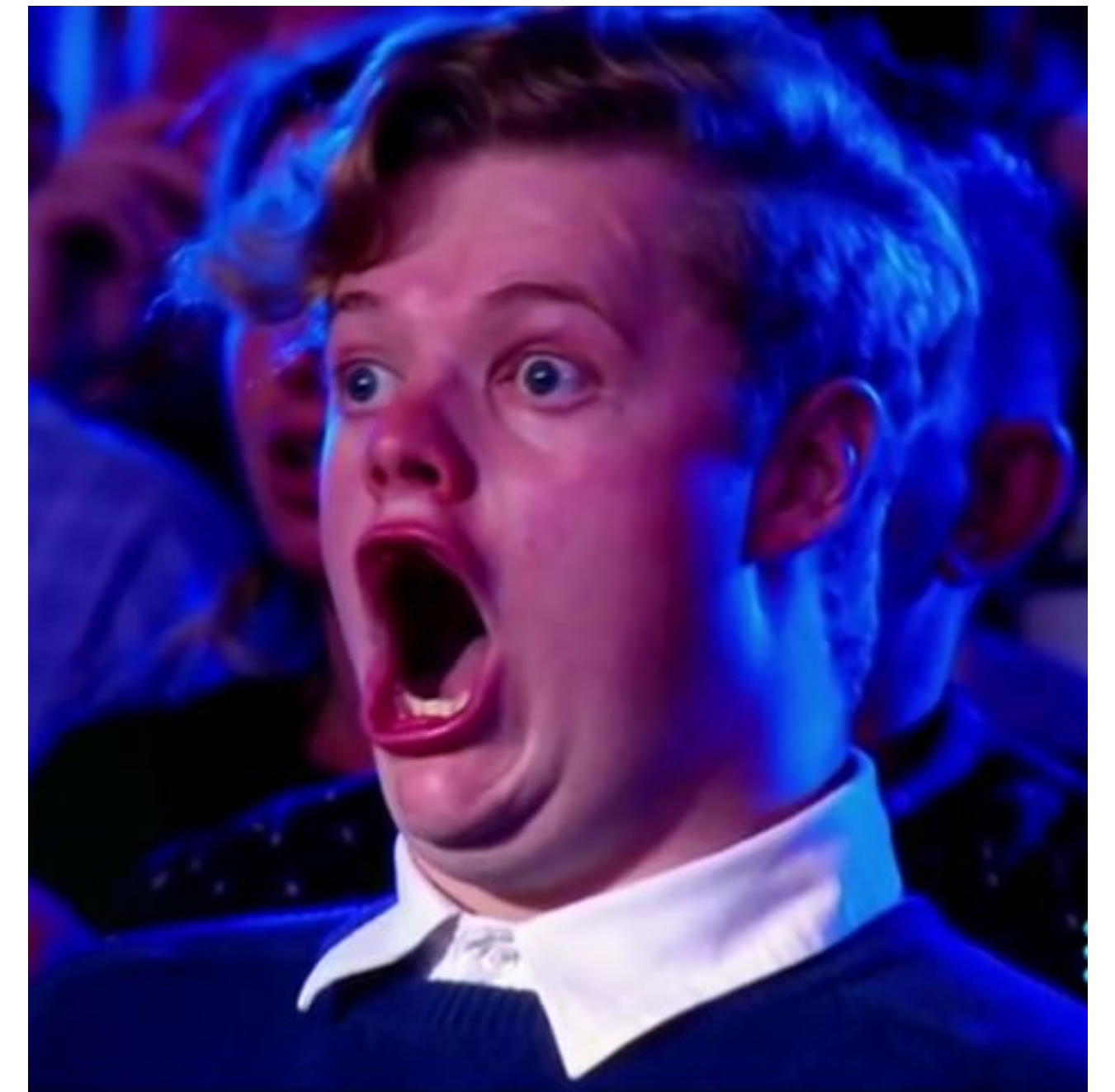
```
1  √ def foo(a):  
2  √      for i in range(len(a)):  
3      |         a[i] += 1  
4
```

```
1  def foo(x):  
2      return x + 1  
3  
4  def bar(func, x):  
5      return func(x)  
6  
7  print(bar(foo, 5))
```


И САМОЕ ВАЖНОЕ!!!

print != return

(Не смейтесь, это правда очень частая ошибка)



Он в глубоком шоке

И САМОЕ ВАЖНОЕ!!!

```
In [1]: def foo():  
...:     print(1)  
...:  
...:     print(foo())
```

```
1  
None
```

```
In [2]:
```

Выведет 1, но вернет None,
которое тоже запринтится

```
In [2]: def foo():  
...:     return 1  
...:  
...:     print(foo())
```

```
1
```

Вернет 1,
которое мы запринтим вызовом
print(foo())



Вопросы?



Позиционные аргументы и именованные

```
1  def foo(a, b):  
2      ...  
3  
4  foo(a=1, b=2) # работает  
5  foo(1, b=2)  # тоже работает  
6  foo(1, 2)    # и так тоже  
7  foo(a=1, 2)  # а так низя
```

Все то же самое верно для
такого объявления функции

```
1  def foo(a, b=3):  
2      ...  
3
```


Позиционные аргументы

```
1 def find_longest(*strings):
2     max_i = 0
3     for i in range(len(strings)):
4         if len(strings[i]) > len(strings[max_i]):
5             max_i = i
6     return strings[max_i]
7
8 print(find_longest("Placebo", "Rammstein", "Arctic Monkeys", "Muse"))
```

Так же, например, с точки зрения аргументов работает функция `max`.
Валидны `max(1, 2)`, `max(6, 8, 2, 3, 4)` и тд

Обычно такие аргументы называют еще `*args`

Keyword аргументы

```
1  def find_longest(**items):
2      max_key = None
3      for key in items.keys():
4          if max_key is None or len(items[key]) > len(items[max_key]):
5              max_key = key
6      return items[max_key]
7
8  print(
9      find_longest(
10         rammstein="Mein herz brennt",
11         placebo="Meds",
12         pleymo="On ne changera rien",
13     ),
14 )
```

Еще их часто называют `**kwargs`.

По сути это возможно передавать
в функцию любые аргументы в виде словаря.

На примере со скриншота `items` -- это как раз словарь.

P.S. Конечно, сделать это можно было проще через `max(items.values())`

Очень приятный бонус

```
check_solution(solution, is_python=True, fail_on_first_test=True, show_failed_test=False)
```

```
check_solution(solution, True, True, False)
```

```
def check_solution(solution, /, *, is_python, fail_on_first_test, show_failed_test)
```

```
1  ✓ def foo(a, /, b, *, c):  
2      ...  
3  
4      foo(1, 2, c=3) # работает  
5      foo(1, b=2, c=3) # работает  
6      foo(a=1, 2, c=3) # ошибка, нельзя key-value после позиционных  
7      foo(a=1, b=2, c=3) # ошибка  
8      foo(a=1, b=2, 3) # ошибка
```

```
def foo(a: str):  
    return a + "b"
```

```
assert isinstance(a, 1)
```

```
foo(1)
```

```
mypy
```

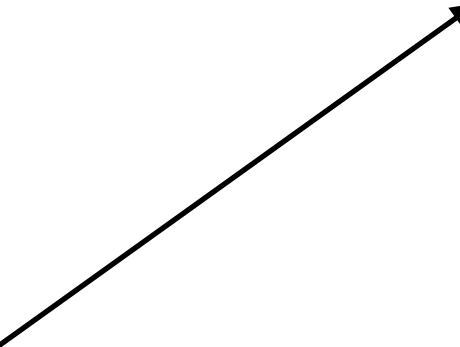


Можно запрещать позиционные или key-value аргументы через спецсимволы:

- 1) / -- до него все аргументы должны быть только позиционные
- 2) * -- после нее все аргументы только key-value
- 3) Между / и * -- как угодно

Глобальные и локальные переменные

переменная x глобальная
для функции foo, поэтому
ее там видно



```
In [7]: x = 5
...:
...: def foo():
...:     print(x)
...:
```

```
In [8]: foo()
5
```

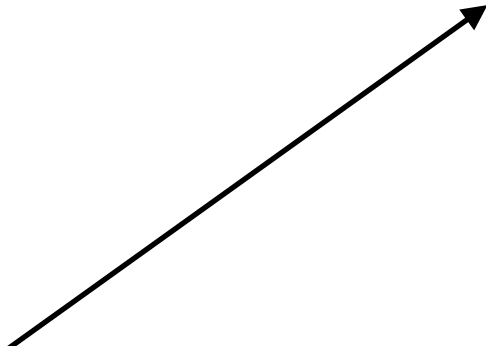
x = 5 # хороший тон

```
def foo():
    print(x)
```

x = 5 # плохой тон
print(globals())
foo()

Глобальные и локальные переменные

НО ПОМЕНЯТЬ ЕЕ ВЫ НЕ
СМОЖЕТЕ



```
In [5]: x = 5
...:
...: def foo():
...:     x += 1 # ошибка
...:     print(x)
...:
```

```
In [6]: foo()
```

```
UnboundLocalError
```

Traceback (most recent call last)

```
Cell In [6], line 1
```

```
----> 1 foo()
```

```
Cell In [5], line 4, in foo()
```

```
      3 def foo():
```

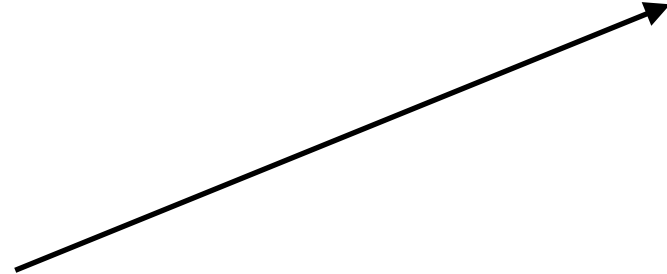
```
----> 4     x += 1 # ошибка
```

```
      5     print(x)
```

```
UnboundLocalError: local variable 'x' referenced before assignment
```

Глобальные и локальные переменные

с помощью `global` вы можете сказать, что вы намеренно хотите менять такую переменную (но много `global` в коде -- это плохой знак)

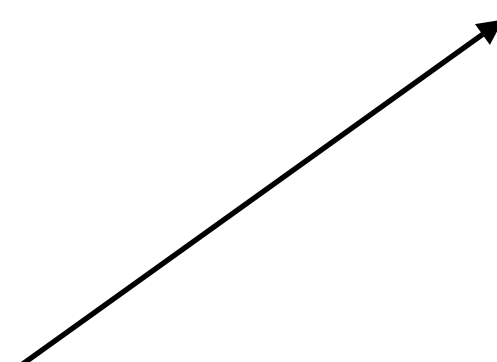


```
In [1]: x = 5
...:
...: def foo():
...:     global x
...:     x += 1 # уже не ошибка
...:     print(x)
...:

In [2]: foo()
6
```

Глобальные и локальные переменные

ВЫНОС МОЗГА: а вот со
списками и почти
любыми другими
коллекциями (множества,
словари и тд) можно без
global, но это тоже
плохой знак, если часто
такое абыюзите

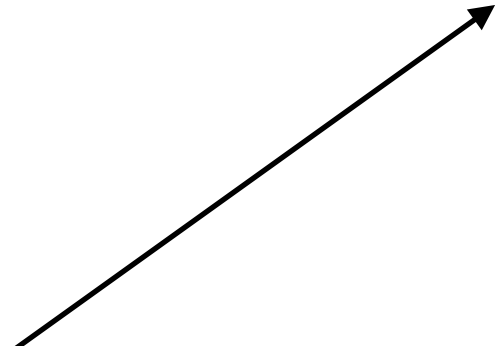


```
In [3]: x = [1, 2, 3]
...:
...: def foo():
...:     x[0] = 5 # тоже не ошибка
...:     print(x)
...:

In [4]: foo()
[5, 2, 3]
```

Глобальные и локальные переменные

а здесь x -- локальная
переменная, поэтому до
нее вы извне достучаться
не сможете



```
In [1]: def foo():  
...:     x = 5  
...:     print(x)  
...:
```

```
In [2]: foo()  
5
```

```
In [3]: print(x)
```

```
NameError
```

Traceback (most recent call last)

```
Cell In [3], line 1
```

```
----> 1 print(x)
```

```
NameError: name 'x' is not defined
```

```
In [4]:
```


Почему так сложно?

- В переменных находятся на самом деле не значения, а ссылки на значения
- Более того, число на самом деле является неизменяемым, как и строка, к примеру
- А вот список может меняться, поэтому поменять его извне -- это значит залезть в какую-то область памяти по ссылке и там что-то поделаться, грубо говоря
- А вот поменять число -- это на самом деле операция вида "поменяй мне ссылку с числа 5 на число 6"

X → 5

Y → [1, 2, 3, 4, 5]

(это не совсем правда,
но для упрощения считаем так)

БОНУС

Функция в функции (в функции в функции...) - это легально

```
1  def foo(x):  
2      def bar(x):  
3          return x + 1  
4  
5      return bar(x) ** 2  
6
```