

# DIRTY PYTHON 1.0

STONE 13TH PRESENTS

ЛЕКЦИЯ 4



DIRTY  
PYTHON

# ЧТО У НАС СЕГОДНЯ?

Функции, функции и еще раз функции

- Виды функций
- type hint
- Значения по умолчанию
- \*args
- \*\*kwargs

функции



memes-arsenal.ru

DIRTY  
PYTHON



# ЧТО ЕСТЬ ФУНКЦИЯ?

- Функция – это кусок кода программы, которая часто повторяется.
- Этот кусок кода можно вынести в функцию, назвать и вызывать при необходимости.
- Зачем использовать функции?
- ✓ Сократить код DRY – don't repeat yourself – не повторяй сам себя

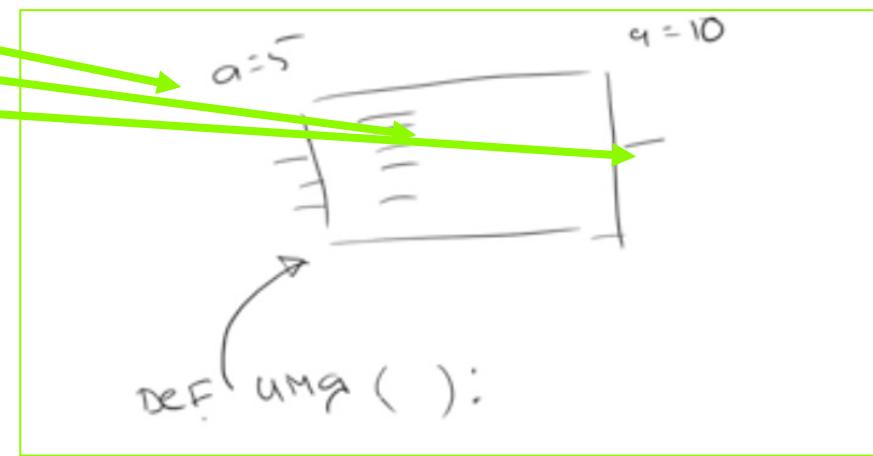
Например ввод числа и проверка его на четность или еще какие-то условия

Лирическое отступление: если не поймете сразу, то вам поможет практика разложить все по полочкам

# ПОПРОБУЕМ ИЗОБРАЗИТЬ РАБОТУ ФУНКЦИИ

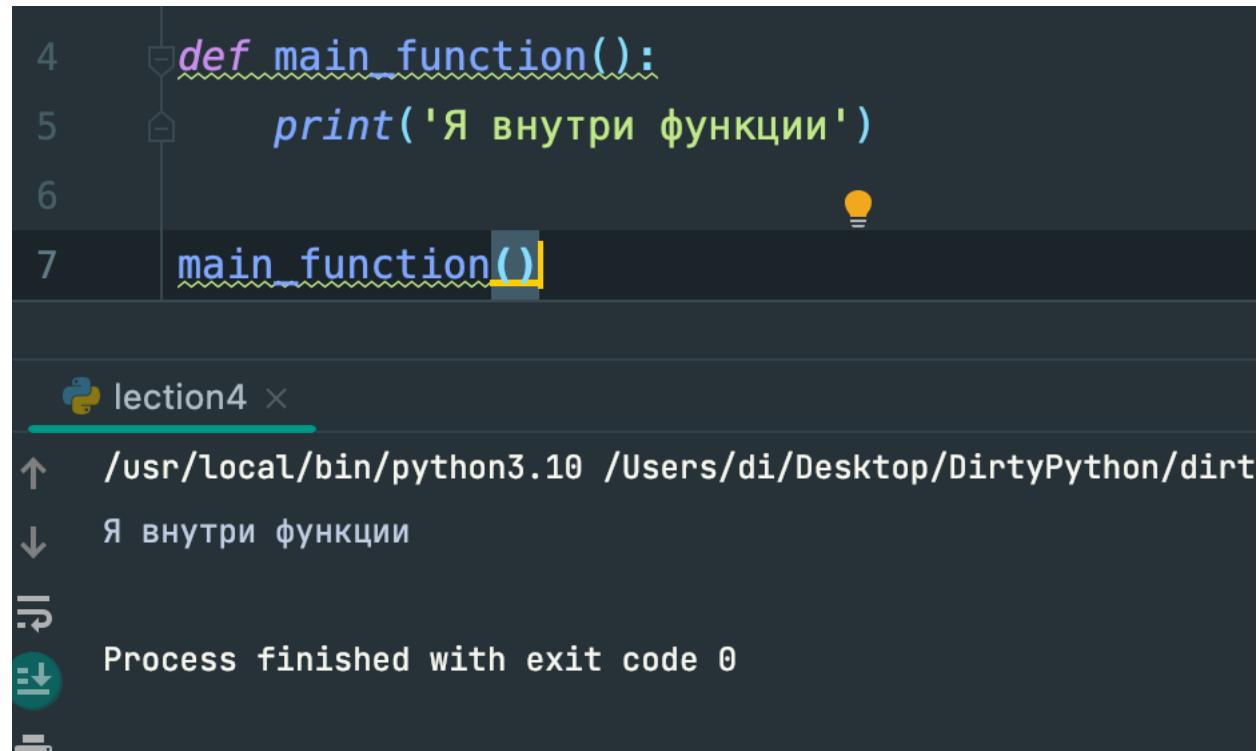


- Функция коробка с кодом
- ✓ В нее что-то заходит
- ✓ Внутри что-то происходит
- ✓ Из нее что-то выходит
- Эдакий кухонный комбайн
- У функции есть специальное слово def (от define) для объявления функции.
- У функции есть имя: те же ограничения в нейминге, что и для переменных (не используем зарезервированные имена)
- В скобках указываем входные данные, они же аргументы (или ничего, но скобки все равно нужны)
- Ставим двоеточие – дальше с отступом идет тело функции, ограниченная табуляцией. Как только закончился отступ – закончилась функция



# А ТЕПЕРЬ КОД

- Написали функцию, которая будет при вызове печатать фразу 'Я внутри функции'
- Как теперь ее вызывать? Просто позовите по имени, но не забудьте поставить скобки (да, даже если они пустые!) Без скобок мы можем посмотреть, где



```
def main_function():
    print('Я внутри функции')
main_function()
```

lection4 ×  
/usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirt  
↑ Я внутри функции  
↓  
Process finished with exit code 0

# ВЫЗОВ ФУНКЦИИ

Это я, орущий на компилятор, медленно  
осознаю, что просто забыл вызвать  
функцию:



DIRTY  
PYTHON



# ФУНКЦИИ: АРГУМЕНТЫ

- Аргументы, они же атрибуты – это то, что мы передаем функции для обработки
- При вызове функции мы должны передать этот аргумент, который будет обработан в теле функции
- Теперь если мы попробуем вызвать функцию, но забудем передать аргумент, то получим ошибку – функция работать не будет.
- А если передадим, то все будет хорошо

The screenshot shows a Python IDE interface with three main sections: a code editor, a terminal, and a status bar.

**Code Editor:**

```
9 def main_function(a):
10     print(f'Я печатаю переменную {a}')
11
12 main_function()
13
```

**Terminal:**

```
lecture4 ×
/usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/Lecture/lection4.py
Traceback (most recent call last):
  File "/Users/di/Desktop/DirtyPython/dirtyPython1/Lecture/lection4.py", line 12, in <module>
    main_function()
TypeError: main_function() missing 1 required positional argument: 'a'
```

**Code Editor (Execution Result):**

```
9 def main_function(a):
10     print(f'Я печатаю переменную {a}')
11
12 main_function('Hi')
13
```

**Terminal (Execution Result):**

```
lecture4 ×
/usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/Lecture/lection4.py
Я печатаю переменную Hi
|
Process finished with exit code 0
```

# ОБЛАСТЬ ВИДИМОСТИ ФУНКЦИИ

- Если вы назовете аргумент функции, так же как переменную внутри кода, то ничего страшного не произойдет. Но вы должны помнить про видимость функции: то что происходит в функции – остается в функции) То есть функция не будет воспринимать свою внутреннюю переменную, как глобальную (если вы это отдельно не укажите).



```
8 a = 78 # это глобальная переменная a, которая равняется 78
9
10 def main_function(a): # а вот тут своя собственная a
11     print(f'Я печатаю переменную {a}')
12
13 main_function('Hi') # теперь для функции a - это Hi
14 print(a) # но наша глобальная a никуда не делась
15
main_function()
File: lection4 ×
/usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/Lecture/lection4.py
↓ Я печатаю переменную Hi
→ 78
```

# КАК РАСШИРИТЬ ВИДИМОСТЬ ФУНКЦИИ?



- Приоритетно работает своя собственная переменная, но если функция внутри аргументов не найдет переменную, то она пойдет ее искать в общем коде.

```
> 8     a = 78      # это глобальная переменная a, которая равняется 78
> 9
> 10    def main_function(b):          # а вот тут своя собственная b
> 11        print(f'Я печатаю переменную {a} и {b}')
> 12        # функция найдет a во внешнем коде и b из переданных аргументов
> 13    main_function(34)      # вот та b, которую мы передаем

run: lection4 ×
> ↑ /usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/Lection/lection4.py
> ↓ Я печатаю переменную 78 и 34
> 
> Process finished with exit code 0
```

# ЕЩЕ ПРО АРГУМЕНТЫ

- Аргументов может быть сколько угодно
- Но если вы написали, что надо передать три аргумента – передаем все три, даже если внутри функции, что-то из аргументов не используется.



# RETURN

- То, что функция оставила после своей работы и есть возвращение: например, функция может вернуть строку или еще что-то
- Если мы вызовем такую функцию просто так, то ничего не увидим, но если мы обернем ее в принт – то уже все распечатается чудесно
- А еще мы можем присвоить результат функции переменной

The screenshot shows a PyCharm interface. In the editor, there is a file named 'main\_function.py' with the following code:

```
a = 78

1 usage
def main_function(a, b, c):
    * return f'я печатаю переменную {a} {b}'

print(main_function(34, 56, 56))


```

In the terminal window, the command 'python main\_function.py' is run, and the output is:

```
D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main_function.py
я печатаю переменную 34 56
```

The screenshot shows a PyCharm interface. In the editor, there is a file named 'main.py' with the following code:

```
a = 78

1 usage
def main_function(a, b, c):
    return f'я печатаю переменную {a} {b}'

text = main_function(34, 56, 56)
print(text)


```

In the terminal window, the command 'python main.py' is run, and the output is:

```
D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py
я печатаю переменную 34 56

Process finished with exit code 0
```

the RETURN





# ВИДЫ ФУНКЦИЙ

- ✓ Что-то принимает – что-то отдает
  - ✓ Ничего не принимает – что-то отдает
  - ✓ Что-то принимает – ничего не отдает
  - ✓ Ничего не принимает – ничего не отдает
- 
- Но на самом деле пайтон в любом случае возвращает None
  - Так что на деле только два вида функций
- ✓ Что-то принимает – что-то отдает
  - ✓ Ничего не принимает – что-то отдает

```
def (g):  
    return
```

```
def ():  
    return
```

```
def (a):
```

```
def ():
```

		да	нет	принимает
да	☒	☒		
нет	☒	☒		

отдаёт



# ФУНКЦИИ И ИХ RETURN

- Функция, в которую мы что-то передали, и из этого что-то получили
- И функция, в которой мы вроде ничего не возвращали, но все равно, кое-что получили - None

```
1 usage
2
3
4 def summ_digits(a, b):
5     c = a + b
6     return c
7
8
9 result = summ_digits(5, 9)
10
11 print(result)
12
```

D:\WorkShopPy\.my\_venv\_3.11.3\Scripts\python.exe D:\WorkShopPy\main.py

```
1 usage
2
3
4 def summ_digits(a, b):
5     c = a + b
6
7
8 result = summ_digits(5, 9)
9
10 print(result)
11
```

D:\WorkShopPy\.my\_venv\_3.11.3\Scripts\main.py

None

# ФУНКЦИЯ, КОТОРАЯ НИЧЕГО НЕ ПРИНИМАЕТ, НО ЧТО-ТО ВОЗВРАЩАЕТ

- И с такими функциями вы будете сталкиваться чаще, чем вам кажется
- Ну вот мы получили функцию, которая ничего не принимает, но возвращает список, в чем мы можем убедиться, если сделаем принт
- Но конечно, в такую функцию просится атрибут – размер и/или еще какие-то аргументы: но сколько объявили, столько надо передать – ни больше, ни меньше!

```
1 usage
4 def create_list():
5     my_list = []
6     for i in range(10):
7         my_list.append(i)
8     return my_list
9
10
11 create_list()
12
13
```

```
1 usage
4 def create_list():
5     my_list = []
6     for i in range(10):
7         my_list.append(i)
8     return my_list
9
10
11 print(create_list())
12
13
```

```
4 def create_list(size):
5     my_list = []
6     for i in range(size):
7         my_list.append(i)
8     return my_list
9
10
11 print(create_list(3))
12 print(create_list(9))
13 print(create_list(22))
14 print(create_list(4))
15
16
```

D:\WorkShopPy\my\_venv\_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py  
[0, 1, 2]  
[0, 1, 2, 3, 4, 5, 6, 7, 8]  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]  
[0, 1, 2, 3]





# МУДРОСТЬ ОТ СТОУНА

ПАЙТОН – ЧЕСТНЫЙ ЯЗЫК:  
ОН ЧУЖОГО, ЛИШНЕГО НЕ ВОЗЬМЕТ!

# ФУНКЦИИ: КОРОТКО О ГЛАВНОМ



- Объявляем функцию def
- Придумываем название
- Передаем или не передаем аргументы
- Делаем тело – сколь угодно сложным, главное, чтобы оно служило выполнению поставленной задачи
- Возвращаем или не возвращаем что-то (если нет return то результат будет None)
- Не забываем вызвать функцию
- Помним, что если мы в функции используем изменяемые объекты, то в принципе можно и без return – объекты изменяться. А если работаем со строкой или числом, то нам надо изменения зафиксировать (сравните две последние картинки)

```
4 def create_list(size):
5     my_list = []
6     for i in range(size):
7         my_list.append(i)
8     for i in range(len(my_list)):
9         if i%2 == 0:
10             my_list[i] *= 10
11
12
13
14 print(create_list(3))
```

0:\WorkShopPy\my\_venv\_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py [0, 1, 20]

The screenshot shows two side-by-side code editors in PyCharm. Both snippets define a function `create_list` that performs operations on its argument.

**Left Snippet:**

```
4 def create_list(num):
5     num *= 10
6     num /= 5
7
8     create_list(number)
9     print(number)
10
11
12
13
14
15
16
17
18
19
20
```

A tooltip for the parameter `num` in the first line of the function definition states: "Parameter num of main.create\_list. num: Any". Below the code, a note says: "Каким number был таким и остался".

**Right Snippet:**

```
4 def create_list(num):
5     num *= 10
6     num /= 5
7     *
8     return num
9
10
11
12
13
14
15
16
17
18
19
20
```

A tooltip for the parameter `num` in the first line of the function definition states: "Parameter num of main.create\_list. num: Any". Below the code, a note says: "Сохранили новый number".

D:\WorkShopPy\my\_venv\_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py 20.0

# TYPE HINTING



- В Python не надо сразу объявлять тип данных, когда мы объявляем новую переменную, он сам разберется.
- То же самое и для аргументов функций – мы можем просто ничего не объясняя написать нейминг.
- Но тем не менее, мы можем оставить подсказку для тех, кто будет читать код (и возможно это мы сами из будущего): какой тип данных мы ждем от аргументов и даже какой тип данных будет возвращать функция. Вот эта подсказка и называется type hint



# TYPE HINT



- Для аргументов type hint указывается непосредственно после наименование через двоеточие.
- Если мы в функцию передадим другой тип данных, то pycharm нам это дело подчеркнет, намекая, что мы по плану должны передать что-то другое.
- При этом, если тип данных позволяет отработать телу функции без ошибок (в данном случае предполагался int, мы отправили float, но его все равно можно умножить и разделить) – функция отработает без ошибок
- Итак, анотация type hinting – это просто подсказка как для нас самих, так и для других программистов (не пишите нечитаемый код)
- Плюс, тайп хинтинг позволяет среди разработки предлагать вам подходящие методы для указанного типа данных



```
4 def change_num(num: int):  
5     num *= 10  
6     num /= 5  
7     return num  
8  
9 print(change_num(5.8))  
10  
11  
change_num()  
Run main x  
D:\WorkShopPy\.my_venv_3.11.3\Scripts\pyth  
11.6
```

# TYPE HINT для РЕЗУЛЬТАТА



- Так же можно обозначить и тип предполагаемого результата
- Сразу после скобок перед двоеточием пишется дефис, знак больше (стрелочка) и тип данных для результата def my\_func (a: int) -> float:
- Если мы укажем какой-то неправильный тип данных, тот который мы не сможем получить исходя из кода, среда разработки это подчеркнет, указывая на возможную ошибку. Но если вы все равно решите выполнить – то вы получите такой же результат, как если бы не использовали type hints вовсе

```
4 def change_num(num: int) -> float:
5     num *= 10
6     num /= 5
7     return num
8
9 print(change_num(5))
10
11
```

change\_num()

Run main

D:\WorkShopPy\my\_venv\_3.11.3\Scripts\python.exe D:\WorkShopPy\p  
10.0

```
4 def change_num(num: int, name: str) -> str:
5     num *= 10
6     num //= 5
7     return f'Новое число {name} - {num}'
8
9 print(change_num(10, 'БОЛЬШОЕ'))
10
11
```

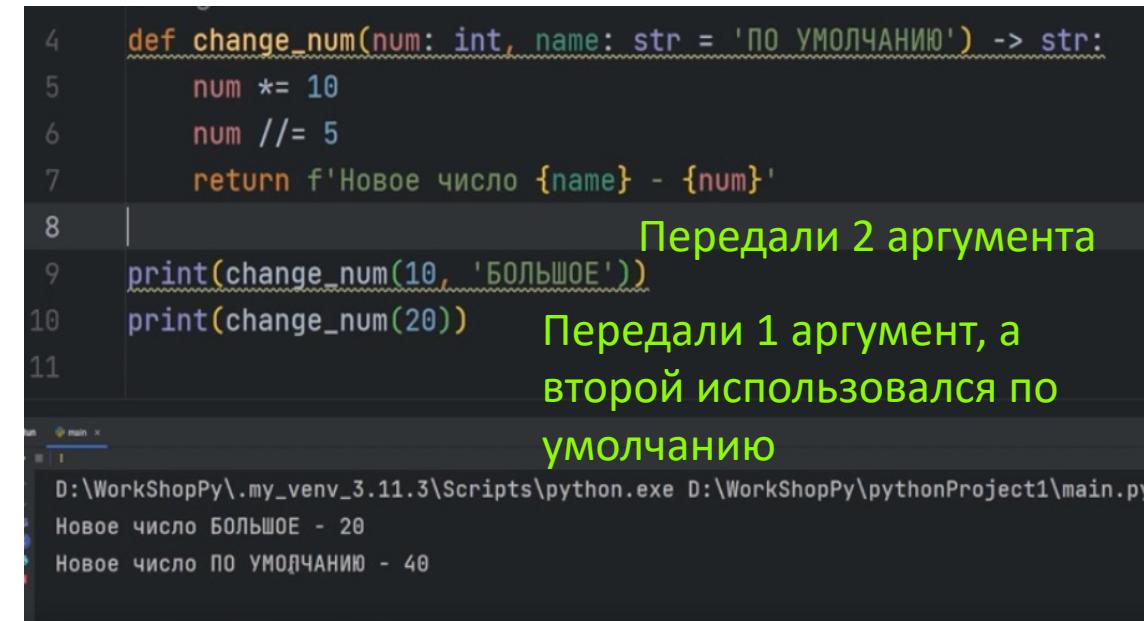
change\_num()

Run main

D:\WorkShopPy\my\_venv\_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py  
Новое число БОЛЬШОЕ - 20

# ЗНАЧЕНИЕ ПО УМОЛЧАНИЮ

- Для аргументов мы сами можем задать значение по умолчанию: если мы при вызове функции передадим значение аргумента – то будет использоваться оно, а если нет – то заданное по умолчанию.
- Помните, что если мы не передадим в функцию нужное количество аргументов, то получим ошибку? Так вот, если будут указаны значения аргументов по умолчанию, такой неприятности не случится (хотя, возможно, вы получите не то, что хотите)
- Чтобы указать значение по умолчанию, нужно просто после нэйма аргумента (и при наличии, type hint написать знак равно и само значение: num: int = 10



```
4 def change_num(num: int, name: str = 'ПО УМОЛЧАНИЮ') -> str:
5     num *= 10
6     num //= 5
7     return f'Новое число {name} - {num}'
8
9 print(change_num(10, 'БОЛЬШОЕ'))
10 print(change_num(20))  
Передали 2 аргумента  
Передали 1 аргумент, а
второй использовался по
умолчанию  
D:\WorkShopPy\my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py
Новое число БОЛЬШОЕ - 20
Новое число ПО УМОЛЧАНИЮ - 40
```



# по умолчанию

- Аргументы у нас вообще-то позиционные: значения надо передавать по очереди, как указано в функции. Поэтому у нас есть ограничение на написание аргументов со значением по умолчанию – они пишутся после позиционных аргументов. Либо же все аргументы должны иметь значение по умолчанию. Тогда можно и вовсе ничего не передавать в функцию.

```
4 def change_num(num: int = 10, name: str = 'BIG') -> str:
5     num *= 10
6     num //= 5
7     return f'Новое число {name} - {num}'
8
9 print(change_num(10, 'БОЛЬШОЕ'))
10 print(change_num())
11
```

```
D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py
Новое число БОЛЬШОЕ - 20
Новое число BIG - 20
```

Process finished with exit code 0

```
4 def change_num(num: int, name: str = 'BIG', devine: int = 10) -> str:
5     num *= 10
6     num //= devine
7     return f'Новое число {name} - {num}'
8
9 print(change_num(10, 'НОВОЕ'))
10 print(change_num(10, 'БОЛЬШОЕ', 1))
11 print(change_num(100))
```

```
D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py
Новое число НОВОЕ - 10
Новое число БОЛЬШОЕ - 100
Новое число BIG - 100
```

Все три вызова  
нормально  
отработают, ведь  
у двух из трех  
аргументов есть  
значение по  
умолчанию

# ОПИСАНИЕ ФУНКЦИИ

- В тройных кавычках можно расписать что и зачем делает функция
- Тогда написав `help(your_function)`, вы получите это самое описание. Обратите внимание – вот тут мы не вызываем функцию, поэтому скобки не нужны
- Но пока сильно не заморачивайтесь – это на будущее

```
15 def make_big_num(num: int, name: str = 'BIG', devine: int = 10) -> float:  
16     '''Функция принимает число, умножает его на 100,  
17     делит на другое либо на 10 по умолчанию  
18     и возвращает строку типа "Новое число BIG – результат"'''  
19     num *= 100  
20     num /= devine  
21     return f'Новое число {name} – {num}'  
22  
23 print(make_big_num(100, 'большое', 1))  
24 print(help(make_big_num))  
25  
26
```

лекция4 ×

/usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/Lecture/lection4.py

↑ Новое число большое - 10000.0  
↓ Help on function make\_big\_num in module \_\_main\_\_:  
≡  
make\_big\_num(num: int, name: str = 'BIG', devine: int = 10) -> float  
 Функция принимает число, умножает его на 100,  
 делит на другое либо на 10 по умолчанию  
 и возвращает строку типа "Новое число BIG – результат"



# \*ARGS И \*\*Kwargs РЕЗИНОВЫЕ ПЕРЕМЕННЫЕ

- Количество аргументов для однотипного действия может меняться... Что же делать? Писать для каждого случая отдельные функции? Запихивать все в какой-нибудь список и переписывать функцию под список, да еще дополнительно собирать аргументы в этот самый список? Все несколько проще!
- Помните у класса могут быть необязательные параметры? Так и функции они тоже могут быть! Называют их args и пишут с одной звездочкой. Вот так: \*args. В данном случае это не умножение, а распаковка объекта (ну да, как со списком при печати)

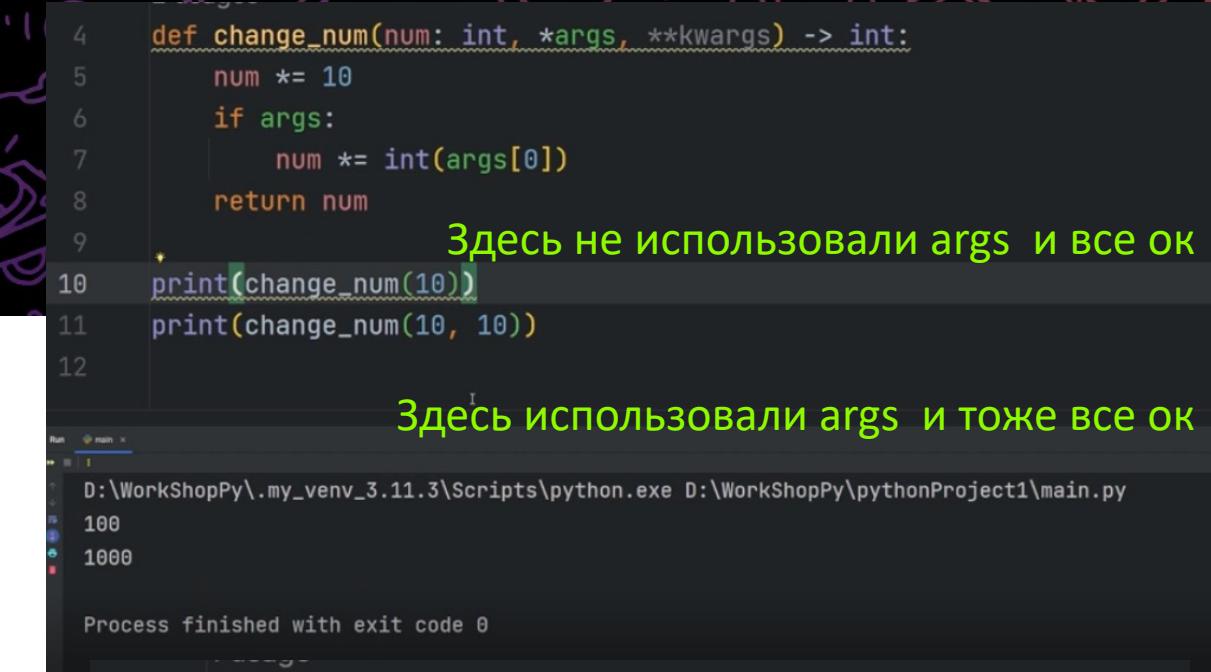
# \*ARGS

- В args может передаваться множество значений, но они передаются кортежем, что нужно учитывать

```
4 def change_num(num: int, *args, **kwargs) -> int:  
5     num *= 10  
6     if args:  
7         print(args) Просто распечатаем наши args  
8     return num  
9  
10    print(change_num(10))  
11    print(change_num(10, 10))  
12    print(change_num(10, 10, 'adafs', 6.9, [5, 6, 7]))  
13
```

Передаем сколько угодно и любые типы

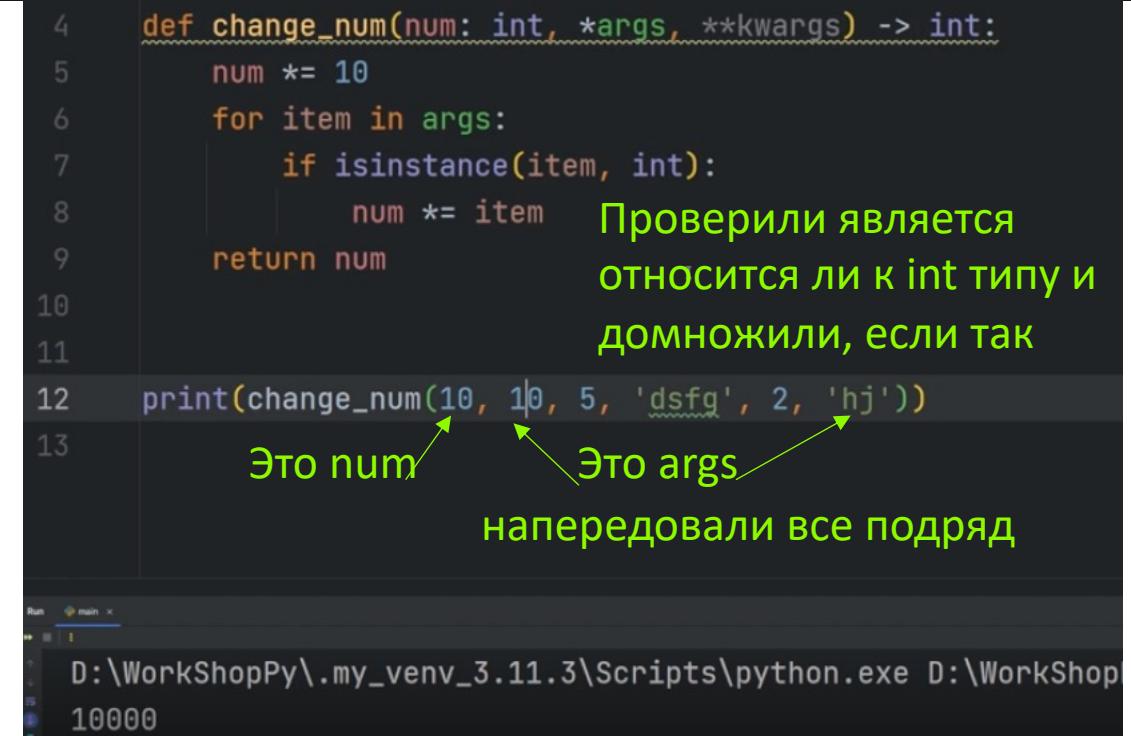
```
D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py  
100  
(10,)  
100  
(10, 'adafs', 6.9, [5, 6, 7])  
100
```



```
4 def change_num(num: int, *args, **kwargs) -> int:  
5     num *= 10  
6     if args:  
7         num *= int(args[0])  
8     return num  
9  
10    print(change_num(10))  
11    print(change_num(10, 10))  
12
```

Здесь не использовали args и все ок

```
D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py  
100  
1000  
Process finished with exit code 0
```



```
4 def change_num(num: int, *args, **kwargs) -> int:  
5     num *= 10  
6     for item in args:  
7         if isinstance(item, int):  
8             num *= item  
9     return num  
10  
11  
12    print(change_num(10, 10, 5, 'dsfg', 2, 'hj'))  
13
```

Проверили является относится ли к int типу и домножили, если так

Это num      Это args

напередовали все подряд

```
D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py  
10000
```

# \*\*Kwargs

- Как вы уже поняли это будет что-то похожее на \*args только с дополнительным условием.
- Итак, если \*args от arguments, то \*\*kwargs – от keys with arguments
- То есть это не просто дополнительные аргументы, а аргумент связанный с каким-то значением (что-то вроде словаря)! Поэтому и две звездочки нужны, ведь функции придется распаковывать это дважды!
- Собственно очень похоже на \*args, только больше значений
- Ключ и значение передаем через равно
- Да и если ничего не передать, то ничего и не будет
- Вот статья на хабре про это вот все <https://habr.com/ru/company/rvuds/blog/482464/>

```
def change_num(num: int, *args, **kwargs) -> int:  
    num *= 10  
    for item in args:  
        if isinstance(item, int):  
            num *= item  
    print(kwargs)  
    return num  
  
print(change_num(10, 10, 5, 'dsfg', 2, name='hj', age=90))
```

Это num

Это args

Это kwargs, они с ключом

D:\WorkShopPy\.my\_venv\_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject1\main.py  
{'name': 'hj', 'age': 90}  
10000

# ПОРЯДОК ПЕРЕДАЧИ АРГУМЕНТОВ



- Вообще-то аргументы передаем в том порядке, как записаны в функции
- Но если мы используем их имена – то уже можно передавать в каком угодно порядке.
- Но либо мы все подписываем, либо не подписываем вообще

```
def change_num (num: int, devine: int, sum_: int) -> float:  
    num /= devine  
    num += sum_  
    return num  
  
print(change_num(num=10, sum_= 4, devine = 2))
```

Run: lection4

/usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/Lection/lection4.py

9.0

# А ЕСЛИ У НАС ПРИЕМЛЕМО НЕСКОЛЬКО ТИПОВ ДАННЫХ ДЛЯ АРГУМЕНТОВ?



- Например, мы вполне можем умножать не только цифры, но и строки. И делить можно как на int, так и на float
- И в type hints это можно указать при помощи значка |

```
26     def change_num (num: int | str, mult: int | float, sum_: int | float ) -> str | float:
27         num *= mult
28         if isinstance(num, int):
29             num += sum_
30         return num
31
32     print(change_num('a', 4, 2))
33     print(change_num(10, 4, 2.5))
34
```

File: lection4 ×  
↑ /usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/Lecture/lection4.py  
↓ aaaa  
= 42.5

ИЧТО ДАЛЬШЕ?

TOO MANY CHOICES

TOO MANY OPTIONS



# И ЧТО ДАЛЬШЕ?

- СДЕЛАЕМ ДОМАШЕЧКУ ЛЕГКУЮ, ПРОСТО,  
ЧТОБЫ ПРОРАБОТАТЬ СИНТАКСИС
- А НА СЛЕДУЮЩЕЙ ЛЕКЦИИ РАЗБЕРЕМ  
МОДУЛЬНОСТЬ И РЕКУРСИЮ
- В ВОСКРЕСЕНЬЕ ОПЯТЬ ВСЕ РАЗБЕРЕМ



D1RTY  
РУТНОН