

# Лекция 4. Функции

Погружение в Python



# Оглавление

На этой лекции мы	3
Дополнительные материалы к лекции	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	4
<b>1. Создание своих функции</b>	<b>4</b>
Определение функции	5
Что такое pass	6
Аргументы функции	7
Изменяемые и неизменяемые аргументы	8
Возврат значения	9
Значения по умолчанию	11
Изменяемый объект как значение по умолчанию	11
Позиционные и ключевые параметры	12
Параметры args и kwargs	14
Области видимости: global и nonlocal	15
Доступ к константам	17
Анонимная функция lambda	18
Документирование кода функций	19
Задание	20
<b>2. Функции “из коробки”</b>	<b>21</b>
Повторяем map(), filter(), zip()	22
Функция map()	22
Функция filter()	22
Функция zip()	22
Функции max(), min(), sum()	23

Функция max()	23
Функция min()	24
Функция sum()	24
Функции all(), any()	24
Функция all()	24
Функция any()	25
Функции chr(), ord()	26
Функция chr()	26
Функция ord()	26
Функции locals(), globals(), vars()	26
Функция locals()	26
Функция globals()	27
Функция vars()	28
Задание	28
<b>3. Вывод</b>	<b>28</b>

## На этой лекции мы

1. Разберёмся с созданием собственных функций в Python.
2. Изучим работу встроенных функций “из коробки”.

## Дополнительные материалы к лекции

Встроенные в Python функции: <https://docs.python.org/3/library/functions.html>

# Краткая выжимка, о чём говорилось в предыдущей лекции

## На прошлой лекции мы:

1. Разобрали что такое коллекция и какие коллекции есть в Python.
2. Изучили работу со списками, как с самой популярной коллекцией.
3. Узнали как работать со строкой в ключе коллекции.
4. Разобрали работу с кортежами.
5. Узнали что такое словари и как с ними работать.
6. Изучили множества и особенности работы с ними.
7. Познакомились с классами байт и массив байт.

## Термины лекции

- **Функция в программировании, или подпрограмма** — фрагмент программного кода, к которому можно обратиться из другого места программы.
- **Функции высшего порядка** — это функции, которые работают с другими функциями, либо принимая их в виде параметров, либо возвращая их.

## Подробный текст лекции

### Введение

**Функция** — фрагмент программного кода, к которому можно обратиться из другого места программы. Функцию стоит представлять как чёрный ящик. В ящик попадают данные, обрабатываются внутри (для пользователя функции не важно как они обрабатываются) и ящик возвращает готовый результат. Подобные функции обеспечивают простоту использования и возможности переносимости и переиспользования в других частях проекта и даже в других проектах.

В Python можно создавать свои функции, использовать встроенные функции интерпретатора, а также работать с функциями из модулей и пакетов стандартной библиотеки и из устанавливаемых дополнений. На этой лекции подробно поговорим про самописные функции и функции “из коробки”.

# 1. Создание своих функции

Как и всё в Python функция является объектом. Можно сказать, что питоновская функция это функция высшего порядка. Она может работать с другими функциями, либо принимая их в виде параметров, либо возвращая их. Проще говоря, функцией высшего порядка называется такая функция, которая принимает функцию-объект как аргумент или возвращает функцию-объект в виде выходного значения.

Разберёмся в понятиях “вызываем” и “передаём” на уже знакомом примере кода.

```
a = 42
print(type(a), id(a))
print(type(id))
```

Функция `print` вызывается с двумя аргументами - функциями. Каждая из переданных в качестве аргументов функций: `type` и `id` так же вызываются с переменной `a` в качестве аргумента.

Во втором случае функция `type` вызывается с функцией `id` в качестве аргумента. При этом у `id` отсутствуют круглые скобки после имени. Мы не вызываем её, а передаём как объект.

Ещё один пример передачи функции ниже.

```
very_bad_programming_style = sum
print(very_bad_programming_style([1, 2, 3]))
```

Передали в переменную встроенную функцию `sum`. Теперь переменную можно вызывать как функцию суммирования.

Итого. Наличие круглых скобок после имени функции с аргументами или без них внутри скобок — **вызов** функции. Имя функции без скобок — **передача** функции как объекта.

## Определение функции

Для определения собственной функции используется зарезервированное слово `def`. Далее указывается имя функции, круглые скобки с параметрами при необходимости и двоеточием. Со следующей строки описывается тело функции как

вложенный блок, т.е. с 4 отступами для каждой строки тела.

```
def my_func():  
    pass
```

Определили функцию под именем my\_func, которая не принимает аргументы и ничего не делает.



**PEP-8!** Имена функций записываются в стиле snake\_case как и имена переменных

## Что такое pass

Похоже мы впервые встретили слово pass. Это зарезервированное слово, которое ничего не делает. Используется в тех местах, где должен быть код для верной работы программы, но его пока нет. Например мы не можем создать функцию без тела. Нужна как минимум одна строка. В ней мы и написали pass.



**Внимание!** Не злоупотребляйте использованием pass. Делать десятки заготовок функций и классов с pass на будущее - не лучшая привычка. Создавайте код по мере того как он вам нужен.

И сразу пару антипримеров использования pass которые выдают в программисте новичка, радостно вставляющего новое слово повсюду в коде.

Плохо:

```
if a != 5:  
    pass  
else:  
    a += 1
```

Уже лучше:

```
if a == 5:  
    a += 1  
else:  
    pass
```

Отлично. Ничего лишнего:

```
if a == 5:
    a += 1
```

## Аргументы функции

Рассмотрим функцию с параметрами, т.е. принимающую на вход значения. Вспомним как в школе решали квадратные уравнения. Заодно разберём особенности создания функций в Python.

```
def quadratic_equations(a: int | float, b: int | float, c: int | float) -> tuple[float, float] | float | str:
    d = b ** 2 - 4 * a * c
    if d > 0:
        return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 * a)
    elif d == 0:
        return -b / (2 * a)
    else:
        return 'Нет решений'

print(quadratic_equations(2, -3, -9))
```

Во первых воспользовались аннотацией типов. Указали, что на вход ожидаем три значения целого или вещественного типа. Далее после стрелки указано, что функция может вернуть кортеж с парой вещественных значений, т.е. два корня уравнения, либо одно число когда дискриминант равен нулю, либо строку — фразу нет решений.



**Внимание!** Обратите внимание, что смешивать текст и числа в выводе — не лучшая идея. Тот, кто будет использовать функцию будет вынужден делать проверки на тип возвращаемого значения. Логичнее возвращать None, если уравнение не имеет решений. В таком случае первая строка будет записана как:


```
def quadratic_equations(a: int | float, b: int | float, c: int | float) -> tuple[float, float] | float | None:
```

А в последней строке вернём ничего:

```
return None
```

Кстати без указания типов первая строка могла быть записана как:

```
def quadratic_equations(a, b, c):
```

 **PEP-8!** Обратите внимание на пустые строки между функцией и её вызовом. Рекомендуется оставлять по 2 пустых строки как после, так и до функции.

В нашем примере мы указали 3 позиционных параметра. При вызове функции значения попадают в соответствующие позиции переменные. При попытке передать отличное от указанного в описании функции число аргументов получим ошибку `TypeError`.

```
print(quadratic_equations(2, -3)) # TypeError:
quadratic_equations() missing 1 required positional argument: 'c'
```

## Изменяемые и неизменяемые аргументы

При передаче аргументов в функцию стоит помнить изменяемого типа объект или нет. От этого зависит поменяем мы оригинал или он останется неизменным. Пример работы с неизменяемыми переменными.

```
def no_mutable(a: int) -> int:
    a += 1
    print(f'In func {a = }') # Для демонстрации работы, но не
    # для привычки принтить из функции
    return a

a = 42
print(f'In main {a = }')
z = no_mutable(a)
print(f'{a = }\t{z = }')
```



Попытка изменить содержимое переменной внутри функции не привела к изменению одноимённой переменной вне функции. Подробнее об областях видимости далее в лекции.

А пока пример работы с изменяемыми объектами.

```
def mutable(data: list[int]) -> list[int]:
    for i, item in enumerate(data):
        data[i] = item + 1
    print(f'In func {data = }') # Для демонстрации работы, но не
    # для привычки принтить из функции
    return data

my_list = [2, 4, 6, 8]
print(f'In main {my_list = }')
new_list = mutable(my_list)
print(f'{my_list = }\t{new_list = }')
```

Изменение списка внутри функции привело к изменению списка вне функции.



**Важно!** В Python аргументы передаются внутри функции по ссылке на объект. Следовательно изменяемый объект можно поменять. А при попытке изменить неизменяемый создадим новый объект либо получим ошибку.

## Возврат значения

Как вы уже заметили внутри функции на печать ничего не выводится. Мы не используем `print` в теле, а выводим на печать результат работы функции. Такой подход предпочтительнее. Ведь результат можно сохранить в переменную для дальнейшей работы с ним.

Для возврата результатов используется зарезервированное слово `return`. Указанные после слова объекты возвращаются как результат работы функции.

- Если указан один объект — возвращается именно этот объект.
- Если указано несколько значений через запятую, возвращается кортеж с перечисленными значениями
- Если ничего не указано после `return` — возвращается `None`

После выполнения строки с командой `return` работа функции завершается, даже если это не последняя строка в теле функции.

```
def quadratic_equations(a, b, c):
    d = b ** 2 - 4 * a * c
    if d > 0:
        return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 *
a)
    if d == 0:
        return -b / (2 * a)
    return None
```

Мы убрали else, т.к. последний возврат будет выполнен только в том случае, когда не сработала ни одно условие выше и следовательно не получилось выйти из функции через другой return. По этой же причине заменили elif на if.

## Неявный return

Иногда функции не возвращают значения. Точнее нет явного возврата. В этом случае Python “мысленно” дописывает в качестве последней строки функции return None. Немного изменим прошлый пример и посмотрим на новый результат.

```
def no_return(data: list[int]):
    for i, item in enumerate(data):
        data[i] = item + 1
    print(f'In func {data = }') # Для демонстрации работы, но не
для привычки принтить из функции

my_list = [2, 4, 6, 8]
print(f'In main {my_list = }')
new_list = no_return(my_list)
print(f'{my_list = }\t{new_list = }')
```

Функция состоит из 4 строк кода. Можно представить, что в 5-й автоматически дописался возврат None. Именно по этой причине в переменной new\_list вместо списка содержится None.



**Важно!** Если функция ничего не возвращает, значит она возвращает ничего — None. Теперь можно изменить функцию поиска корней квадратного уравнения и сделать короче.

```
def quadratic_equations(a, b, c):
```

```

d = b ** 2 - 4 * a * c
if d > 0:
    return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 *
a)
if d == 0:
    return -b / (2 * a)

```

## Значения по умолчанию

Функция может содержать значения по умолчанию для своих параметров. Например:


```

def quadratic_equations(a, b=0, c=0):
    d = b ** 2 - 4 * a * c
    if d > 0:
        return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 *
a)
    if d == 0:
        return -b / (2 * a)

print(quadratic_equations(2, -9))

```

Переменная `a` должна быть передана в обязательном порядке. Если не передать 2-й и/или 3-й аргумент, в переменные попадут нули как значения по умолчанию.

 **PEP-8!** Для указания значения по умолчанию используется знак равенства. До и после такого равно пробелы не ставятся.

## Изменяемый объект как значение по умолчанию

В качестве значения по умолчанию нельзя указывать изменяемые типы: списки, словари и т.п. Это приведёт к неожиданным результатам:

```
def from_one_to_n(n, data=[]):
    for i in range(1, n + 1):
        data.append(i)
    return data

new_list = from_one_to_n(5)
print(f'{new_list = }')
other_list = from_one_to_n(7)
print(f'{other_list = }')
```

`other_list` содержит в себе и новую последовательность и ту, которая была в списке `new_list`. Связано это с тем, что значение по умолчанию задаётся один раз при создании функции. Каждый вызов — работа со списком `data` и его очередное изменение.

Если в качестве значения по умолчанию нужен изменяемый тип данных, используют особый приём с `None`

```
def from_one_to_n(n, data=None):
    if data is None:
        data = []
    for i in range(1, n + 1):
        data.append(i)
    return data
```

Если изменяемый объект не передан, он создаётся по новой для каждого вызова функции.

## Позиционные и ключевые параметры

Пришло время поговорить о позиционных и ключевых параметрах функции. Начнём с общего синтаксиса.

```
def func(positional_only_parameters, /, positional_or_keyword_parameters, *,
keyword_only_parameters):
    pass
```

При указании параметров функции вначале идут позиционные параметры. При вызове функции передаются значения без указания имени переменной-аргумента. Косая черта не является переменной. Это символ разделитель. После неё могут

идти как позиционные, так и ключевые параметры. Далее символ разделитель звёздочка указывает только на ключевые параметры.



**Важно!** Косая черта и звёздочка одновременно или по отдельности могут отсутствовать при определении функции.

Разберем передачу аргументов по позиции и по имени на примерах.

- **Пример обычной функции:**

```
def standard_arg(arg):  
    print(arg)    # Принтим для примера, а не для привычки  
  
standard_arg(42)  
standard_arg(arg=42)
```

Функция может принимать значения по позиции и по ключу. Мы явно указали имя переменной.

- **Пример только позиционной функции:**

```
def pos_only_arg(arg, /):  
    print(arg)    # Принтим для примера, а не для привычки  
  
pos_only_arg(42)  
pos_only_arg(arg=42)    # TypeError: pos_only_arg() got some  
positional-only arguments passed as keyword arguments: 'arg'
```

Теперь функция принимает только позиционные параметры.

- **Пример только ключевой функции:**

```
def kwd_only_arg(*, arg):  
    print(arg)    # Принтим для примера, а не для привычки  
  
kwd_only_arg(42)  
kwd_only_arg(arg=42)
```

А теперь наоборот, можем принимать только значения по ключу.

- **Пример функции со всеми вариантами параметров:**

```
def combined_example(pos_only, /, standard, *, kwd_only):
```


```

    print(pos_only, standard, kwd_only)  # Принтим для примера, а
не для привычки

combined_example(1, 2, 3)  # TypeError: combined_example() takes
2 positional arguments but 3 were given
combined_example(1, 2, kwd_only=3)
combined_example(1, standard=2, kwd_only=3)
combined_example(pos_only=1, standard=2, kwd_only=3)  #
TypeError: combined_example() got some positional-only arguments
passed as keyword arguments: 'pos_only'

```

И наконец пример со всеми возможными вариантами параметров.

 **Важно!** Если функция принимает несколько ключевых параметров, порядок передачи аргументов может отличаться.

```

def triangulation(*, x, y, z):
    pass

triangulation(y=5, z=2, x=11)

```

## Параметры args и kwargs

Отдельно разберём пару особых параметров. Звёздочка args и две звёздочки kwargs.

Важно! Python в первую очередь смотрит на звёздочки, а не на имя переменной. Но среди разработчиков приняты имена args и kwargs. Они делают код привычным, т.е. повышают читаемость. Не используйте другие переменные.

Начнём с \*args:

```

def mean(args):
    return sum(args) / len(args)

print(mean([1, 2, 3]))
print(mean(1, 2, 3))  # TypeError: mean() takes 1 positional


```

*argument but 3 were given*

Функция может принять лишь один аргумент. В случае со списком проблем нет. Но если перечислить элементы через запятую и не указать скобки - не передать кортеж, получим ошибку.

```
def mean(*args):  
    return sum(args) / len(args)  
  
print(mean(*[1, 2, 3]))  
print(mean(1, 2, 3))
```


Теперь функция принимает любое число позиционных аргументов. Переменная args превращается в кортеж. Можно сказать, что звёздочка упаковала все позиционные аргументы в один кортеж.

 **Внимание!** При вызове функции со списком перед квадратными скобками добавили звёздочку. Смысл её противоположный. Мы распаковали список. Каждый элемент был передан в функцию по отдельности.

Параметр **\*\*kwargs** работает аналогично, но принимает ключевые параметры и возвращает словарь.

```
def school_print(**kwargs):  
    for key, value in kwargs.items():  
        print(f'По предмету "{key}" получена оценка {value}')
```

```
school_print(химия=5, физика=4, математика=5, физра=5)
```

 **Важно!** Благодаря кодировке utf-8 мы смогли передать в функцию переменные на русском языке.

При написании своих функций стоит помнить о возможности сочетать позиционные и ключевые переменные, специальные символы разделители / и \*, а также переменные \*args и \*\*kwargs.

# Области видимости: global и nonlocal

Хорошая функция работает как чёрный ящик. Использует только переданные ей значения и возвращает ответ. Но в Python функции могут обращаться к внешним переменным без явной передачи в качестве аргумента.

В Python есть несколько областей видимости:

- локальная — код внутри самой функции, т.е. переменные заданные в теле функции.
- глобальная — код модуля, т.е. переменные заданные в файле py содержащем функцию.
- не локальная — код внешней функции, исключающий доступ к глобальным переменным.

Разберем на примерах.

- **Локальные переменные:**

```
def func(y: int) -> int:
    x = 100
    print(f'In func {x = }')  # Для демонстрации работы, но не
    для привычки принтить из функции
    return y + 1

x = 42
print(f'In main {x = }')
z = func(x)
print(f'{x = }\t{z = }')
```

Переменная x в теле функции и переменная x в основном коде - две разные переменные. Локальная область видимости функции создала свою переменную. Попробуем для эксперимента заменить строку с иском на `x += 100` В результате получаем ошибку `UnboundLocalError: local variable 'x' referenced before assignment`. Функция не смогла увеличить 42 на 100, т.к. переменные лишь для нас выглядят одинаково. Чёрный ящик не увидел x без его явной передачи в функцию.

- **Глобальные переменные:**

```
def func(y: int) -> int:
    global x
    x += 100
    print(f'In func {x = }')  # Для демонстрации работы, но не
    для привычки принтить из функции
    return y + 1
```



```
x = 42
print(f'In main {x = }')
z = func(x)
print(f'{x = }\t{z = }')
```

Теперь переменная `x` в теле функции объявлена как глобальная. Мы получили доступ к внешнему `иксу` со значением 42 и смогли его увеличить. Изменение затронуло как внешний, так и внутренний `икс`.

Важно! Не стоит злоупотреблять командой `global`. В 9 из 10 случаев переменную стоит передать как аргумент в функцию и вернуть ответ.

- **Не локальные переменные:**

```
def main(a):
    x = 1

    def func(y):
        nonlocal x
        x += 100
        print(f'In func {x = }')  # Для демонстрации работы, но
        не для привычки принтить из функции
        return y + 1

    return x + func(a)

x = 42
print(f'In main {x = }')
z = main(x)
print(f'{x = }\t{z = }')
```

Функция `func` вложена в функцию `main`. Благодаря команде `nonlocal` мы смогли получить доступ к `x = 1`. В результате внутри `func` `x` увеличился до 101. В отличие от команды `global`, мы не смогли увидеть внешний `x = 42`. `nonlocal` позволяет заглянуть на верхний уровень вложенности, но не выходить на глобальные переменные модуля.

# Доступ к константам

Один из случаев когда обращение из тела функции к глобальной переменной считается нормальным — доступ к константам.

```
LIMIT = 1_000

def func(x, y):
    result = x ** y % LIMIT
    return result

print(func(42, 73))
```

Константа `LIMIT` является глобальной. При обращении к ней из функции производится поиск в локальной области, т.е. в теле функции. Далее поиск переходит на уровень выше, в глобальную область видимости модуля. Чтение значений констант внутри функции будет работать без ошибок.

## Анонимная функция `lambda`

Анонимные функции, они же лямбда функции — синтаксический сахар для обычных питоновских функций с рядом ограничений. Они позволяют задать функцию в одну строку кода без использования других ключевых слов.

Рассмотрим пример обычной функции и её аналог в виде лямбды для проведения параллели.

```
def add_two_def(a, b):
    return a + b

add_two_lambda = lambda a, b: a + b

print(add_two_def(42, 3.14))
print(add_two_lambda(42, 3.14))
```

После зарезервированного слова `lambda` перечисляются параметры функции через запятую. Далее ставят двоеточие и указывают значение, которое необходимо вернуть без добавления `return`. Функция записывается в одну строку.



**Важно!** С точки зрения разработки присваивание анонимной функции имени является неверным. Лямбды не должны использоваться для подобных программных решений. Аналогичное предупреждение есть и в PEP-8.

Обычно лямбды используют там, где однократно нужна функция и нет смысла заводить определение через `def`

```
my_dict = {'two': 2, 'one': 1, 'four': 4, 'three': 3, 'ten': 10}
s_key = sorted(my_dict.items())
s_value = sorted(my_dict.items(), key=lambda x: x[1])
print(f'{s_key = }\n{s_value = }')
```

В первом случае словарь сортируется по ключам, т.е. по алфавиту. Во втором благодаря лямбде указали сортировку по второму (индексация начинается с нуля) элементу, т.е. по значению.

## Документирование кода функций

Несколько слов о документировании функций. Начнём с того, что документация обычно пишется на английском языке, как универсальном для программистов из любой страны. Вполне допустим и родной язык, если проект локальный. Но лучше воспользоваться онлайн переводчиком и сразу привыкнуть к документированию на английском.

```
def max_before_hundred(*args):
    """Return the maximum number not exceeding 100."""
    m = float('-inf')
    for item in args:
        if m < item < 100:
            m = item
    return m

print(max_before_hundred(-42, 73, 256, 0))
```

## Пояснения к однострочной строке документации

- Тройные кавычки используются, даже если строка помещается на одной строке. Это позволяет легко расширить его позже.
- Закрывающие кавычки находятся на той же строке, что и открывающие. Это выглядит лучше для однострочников.
- Нет пустой строки ни до, ни после строки документации.
- Строка документации — это фраза, заканчивающаяся точкой. Он описывает действие функции или метода как команду
- Однострочная строка документации не должна повторять параметры функции.



**Внимание!** В программе использована переменная, а точнее константа “минус бесконечность” `float('-inf')`. Это особая форма представления бесконечности в памяти интерпретатора, аналогичная бесконечности из модуля `math`.

Если описание функции подразумевает больше подробностей, после первой строки документации оставляют одну пустую. Далее в несколько строк даётся всё необходимое описание. Закрывающие кавычки ставятся на отдельной строке, без текста.

```
def max_before_hundred(*args):  
    """Return the maximum number not exceeding 100.  
  
    :param args: tuple of int or float numbers  
    :return: int or float number from the tuple args  
    """  
    ...
```

Подобная запись автоматически помещает текст в переменную `__doc__`. Описание функции можно будет получить через вызов справки `help` с передачей функции в качестве аргумента.

```
help(max_before_hundred)
```

# Задание

Перед вами функция и её вызов. Найдите три ошибки и напишите о них в чат. У вас три минуты.

```
def func(a=0.0, /, b=0.0, *, c=0.0):  
    """func(a=0.0: int | float, /, b=0.0: int | float, *, c=0.0:  
    int | float) -> : int | float"""  
    if a > c:  
        return a  
    if a < c:  
        return c  
    return  
  
print(func(c=1, b=2, a=3))
```

## 2. Функции “из коробки”

В Python есть ряд встроенных функций. Они доступны всегда, без импортов и других подготовительных операций. Перечислим их в алфавитном порядке:

abs(), aiter(), all(), any(), anext(), ascii(), bin(), bool(), breakpoint(), bytearray(), bytes(), callable(), chr(), classmethod(), compile(), complex(), delattr(), dict(), dir(), divmod(), enumerate(), eval(), exec(), filter(), float(), format(), frozenset(), getattr(), globals(), hasattr(), hash(), help(), hex(), id(), input(), int(), isinstance(), issubclass(), iter(), len(), list(), locals(), map(), max(), memoryview(), min(), next(), object(), oct(), open(), ord(), pow(), print(), property(), range(), repr(), reversed(), round(), set(), setattr(), slice(), sorted(), staticmethod(), str(), sum(), super(), tuple(), type(), vars(), zip().

Часть функций мы уже разбирали на прошлых лекциях. Ещё часть разберём сегодня. О некоторых функциях поговорим на следующих лекциях курса.



**Важно!** Не используйте имена встроенных функций в качестве имён переменных.



**PEP-8!** Если очень хочется, добавляйте к имени переменной символ подчёркивания. Не `sum`, а `sum_`. Не `max`, а `max_`

## Повторяем `map()`, `filter()`, `zip()`

Повторим уже знакомые вам по прошлым курсам функции для обработки последовательностей.

### Функция `map()`

**`map(function, iterable)`** — принимает на вход функцию и последовательность. Функция применяется к каждому элементу последовательности и возвращает `map` итератор.

```
texts = ["Привет", "ЗДОРОВА", "привеТствую"]
res = map(lambda x: x.lower(), texts)
print(*res)
```

В качестве функции использовали лямбду для вызова метода `lower` у каждого из переданных объектов. Объект итератор `res` был распакован в функцию `print` через символ “звёздочка”.

- **Функция `filter()`**

**`filter(function, iterable)`** — принимает на вход функцию и последовательность. Если функция возвращает истину, элемент остаётся в последовательности. Как и `map` возвращает объект итератор.

```
numbers = [42, -73, 1024]
res = tuple(filter(lambda x: x > 0, numbers))
print(res)
```

Лямбда фильтрует элементы больше нуля. Функция `tuple` преобразует итератор к кортежу с положительными числами.

- **Функция zip()**

**zip(\*iterables, strict=False)** — принимает несколько последовательностей и итерируется по ним параллельно.

Если передать ключевой аргумент `strict=True`, вызовет ошибку `ValueError` в случае разного числа элементов в каждой из последовательностей.

```
names = ["Иван", "Николай", "Пётр"]
salaries = [125_000, 96_000, 109_000]
awards = [0.1, 0.25, 0.13, 0.99]
for name, salary, award in zip(names, salaries, awards):
    print(f'{name} заработал {salary:.2f} денег и премию {salary * award:.2f}')
```

Последовательно получаем имена, зарплату и процент премии из каждого списка. Итерация идёт слева направо.

## Функции max(), min(), sum()

Несколько слов о функциях поиска максимума, минимума и подсчёта суммы.

- **Функция max()**

`max(iterable, *, key, default)` или `max(arg1, arg2, *args[, key])`

Функция принимает на вход итерируемую последовательность или несколько позиционных элементов и ищет максимальное из них. Ключевой параметр `key` указывает на то, какие элементы необходимо сравнить, если объект является сложной структурой. Отдельно параметр `default` используется для возврата значения, если на вход передана пустой итератор.

```
lst_1 = []
lst_2 = [42, 256, 73]
lst_3 = [("Иван", 125_000), ("Николай", 96_000), ("Пётр", 109_000)]
print(max(lst_1, default='empty'))
print(max(*lst_2))
print(max(lst_3, key=lambda x: x[1]))
```

В первом случае передана пустая последовательность и функция вернула строку `empty`.

Во втором — распаковали список и нашли максимальное число.

В третьем ищем максимальное среди трёх кортежей по элементу с индексом один, т.е. по числу.

- **Функция min()**

`min(iterable, *, key, default)]` или `min(arg1, arg2, *args[, key])`

Функция работает аналогично `max`, но ищет минимальный элемент.

```
lst_1 = []
lst_2 = [42, 256, 73]
lst_3 = [("Иван", 125_000), ("Николай", 96_000), ("Пётр", 109_000)]
print(min(lst_1, default='empty'))
print(min(*lst_2))
print(min(lst_3, key=lambda x: x[1]))
```

- **Функция sum()**

`sum(iterable, /, start=0)`

Функция принимает объект итератор и подсчитывает сумму всех элементов. Ключевой аргумент `start` задаёт начальное значение для суммирования.

```
my_list = [42, 256, 73]
print(sum(my_list))
print(sum(my_list, start=1024))
```

## Функции `all()`, `any()`

- **Функция all()**

`all(iterable)`

Функция возвращает истину, если все элементы последовательности являются истиной. На Python создание функции `all` выглядело бы так:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```



Функция `all` обычно применяется с результатами каких-то вычислений, которые должны быть истинными или ложными.

```
numbers = [42, -73, 1024]
if all(map(lambda x: x > 0, numbers)):
    print('Все элементы положительные')
else:
    print('В последовательности есть отрицательные и/или нулевые элементы')
```

Функция `map` заменила числа на `True` и `False`, далее `all` проверила все ли элементы больше нуля или есть как минимум один не более нуля.

- **Функция `any()`**

`any(iterable)`

Функция возвращает истину, если хотя бы один элемент последовательности является истиной. На Python создание функции `any` выглядело бы так:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

Функция `any` работает аналогично `all`. Но если `all` можно представить как `if` с цепочкой `and`, то `any` — это `if` с цепочкой `or`.

Функция `map` заменила числа на `True` и `False`, далее `all` проверила все ли элементы больше нуля или есть как минимум один не более нуля.

```
numbers = [42, -73, 1024]
if any(map(lambda x: x > 0, numbers)):
    print('Хотя бы один элемент положительный')
else:
    print('Все элементы не больше нуля')
```



**Важно!** Все перечисленные выше функции имеют линейную асимптотику  $O(n)$ , т.е. функция проходит последовательность от начала до конца прежде чем вернуть результат.

# Функции chr(), ord()

Рассмотрим пару функций для работе со символами и их кодами в Юникод.

- **Функция chr()**

chr(integer)

Функция возвращает строковый символ из таблицы Юникод по его номеру. Номер - целое число от 0 до 1\_114\_111.

```
print(chr(97))
print(chr(1105))
print(chr(128519))
```

- **Функция ord()**

ord(char)

Функция принимает один символ и возвращает его код в таблице Юникод.

```
print(ord('a'))
print(ord('a'))
print(ord('😊'))
```

Функции ord и chr выполняют противоположные действия.

# Функции locals(), globals(), vars()

И в финале несколько функций о переменных и областях видимости.

- **Функция locals()**


Функция возвращает словарь переменных из локальной области видимости на момент вызова функции.

```
SIZE = 10

def func(a, b, c):
    x = a + b
    print(locals())
    z = x + c
    return z
```

```
func(1, 2, 3)
```

Функция вернула словарь с переменными a, b, c, x и их значениями. Константа SIZE не попала в вывод, т.к. не входит в локальную область функции. Так же в словаре отсутствует переменная z. Она была впервые создана после вызова функции locals.

 **Важно!** Python игнорирует попытки обновления словаря locals. Для изменения значений переменных надо обращаться к ним напрямую.

- **Функция globals()**

Функция возвращает словарь переменных из глобальной области видимости, т.е. из пространства модуля.


```
SIZE = 10

def func(a, b, c):
    x = a + b
    print(globals())
    z = x + c
    return z

print(globals())
print(func(1, 2, 3))
```

Функция не сохраняет в словаре локальные переменные функций, даже если будет вызвана из тела функции.

В словаре от globals содержатся и дандер переменные модуля. Они нужны Python для правильной работы кода.

 **Внимание!** Если вызвать функцию locals() из основного кода модуля, а не из функции, результат будет аналогичен работе функции globals()

```
x = 42
glob_dict = globals()
glob_dict['x'] = 73
print(x)
```

В отличие от locals словарь globals позволяет изменить значение переменной.

- **Функция vars()**

Функция без аргументов работает аналогично функции locals(). Если передать в vars объект, функция возвращает его атрибут \_\_dict\_\_. А если такого атрибута нет у объекта, вызывает ошибку TypeError.

```
print(vars(int))
```

Получили все дандер методы класса int

## Задание

Перед вами список и три операции с ним. Напишите что выведет каждая из строк по вашему мнению, не запуская код. У вас 3 минуты.

```
data = [25, -42, 146, 73, -100, 12]
print(list(map(str, data)))
print(max(data, key=lambda x: -x))
print(*filter(lambda x: not x[0].startswith('__'),
globals().items()))
```

## 3. Вывод

**На этой лекции мы:**

1. Разобрались с созданием собственных функций в Python
2. Изучили работу встроенных функций “из коробки”.

## Краткий анонс следующей лекции

1. Разобрать решения задач в одну строку
2. Изучить итераторы и особенности их работы
3. Узнать о генераторных выражениях и генераторах списков, словарей, множеств

4. Разобрать создание собственных функций генераторов.

## Домашнее задание

1. Поработайте со справочной информацией в Python, функцией `help()`. Попробуйте найти информацию об изученных на уроке функциях “из коробки”. Почитайте описание тех функций, которые не рассматривались на уроке. Если вы плохо читаете на английском, воспользуйтесь любым онлайн переводчиком.