Функция

Функция – именованный фрагмент кода, к которому можно обратиться из другого места программы.

Основная **цель** использование функций – **повторное использование** некоторого кода

С функцией можно сделать 2 вещи:

- Определить
- Вызвать

```python
def factorial(n):
    if n == 0:
        return 1
    return factorial(n-1) * n

print(factorial(5))
```

Как определить функцию

Порядок определения функции:

1. Написать ключевое слово «**def**»
2. Написать **имя функции** (имя функции выбираем по тем же правилам, что и имена переменных)
3. **В круглых скобках** указываем **аргументы**, которые должна принимать функция (от 0 до …)
4. Не забываем в конце добавить символ «:»

```
1  def sum_pow(num_1, num_2):
2      return num_1 ** 2 + num_2 ** 2
```

Как вызвать функцию

Вызов функции – выполнение тела с переданными ей аргументами.

Для того, чтобы вызвать функцию из другого участка кода, нужно написать ее имя и круглые скобки (если функции нужны какие-то данные, то пишем их в круглых скобках)

```
1  def count_positive(data: list):
2      counter = 0
3      for item in data:
4          if item > 0:
5              counter += 1
6      return counter
7
8
9  print(count_positive([1, 2, -3, 4, -2]))
10
```

Как вернуть значение из функции

Чтобы вернуть из функции какое-либо значение, используется ключевое слово **return**.

- Функция может возвращать любое количество значений (вернуться внутри tuple)
- Возвращаемое значение можно присвоить переменной для дальнейшего использования
- Если функция ничего не возвращает (мы не написали return), то результат ее выполнения будет None

```
1  def some_func():
2      return 1, 2, 3
3
4
5  def none_func():
6      pass
7
8
9  # 'a': 1, 'b': 2, 'c': 3
10 a, b, c = some_func()
11
12 # 'is_none': None
13 is_none = none_func()
14
```

Чистые функции

Функция с побочным эффектом (side effect) – функция, которая изменяет какие-то внешние данные

Чистая функция – функция, которая получает аргументы, проводит действия, возвращает результат, не затрагивая данные вне себя. Такая функция будет работать каждый раз одинаково при одних и тех же входящих параметрах. Данные функции легче тестировать

```python
from copy import deepcopy
from datetime import datetime

some_data = {
    'username': 'potato',
    'age': 25
}


def function(data: dict):
    data['last_visit'] = datetime.now()


def clear_function(data: dict):
    data_copy = deepcopy(data)
    data_copy['last_visit'] = datetime.now()
    return data_copy


clear_function(some_data)
function(some_data)

```

Внутренние функции

В теле некоторой функции можно определить другую функцию.

- Используется, если необходимо внутри тела выполнять какие то повторяющиеся действия

```
1  def sqrt_all(*args):
2      def sqrt_numb(numb):
3          return numb ** 0.5
4      return [sqrt_numb(item) for item in args]
5
6
7  print(sqrt_all(1, 2, 3, 4, 5))
8
```

Замыкания

Внутренняя функция может действовать как замыкание.

Замыкание – функция, которая динамически генерируется другой функцией и они обе могут изменять и запоминать значения переменных, которые были созданы вне функции.

Замыкания используются в декораторах.

```
1  def multiply(x):
2      def power(y):
3          return x * y
4      return power
5
6
7  x_3 = multiply(3)
8  print(x_3(6))  # 18
9
```

Анонимные функции lambda

В Python **лямбда-функция** – это анонимная функция, выраженная одним выражением.

- Ее можно использовать вместо некоторой маленькой функции
- Лямбда-функции регулярно используются со встроенными функциями map() и filter(), а также functools.reduce(), представленными в модуле functools

```
>>> easy_sqrt = lambda x: x ** 0.5
>>> easy_sqrt(4)
2.0
>>> list(map(lambda x: x.upper(), ['cat', 'dog', 'cow']))
['CAT', 'DOG', 'COW']
>>> list(filter(lambda x: 'o' in x, ['cat', 'dog', 'cow']))
['dog', 'cow']

```

Встроенные функции

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

Аргументы

передача данных в функцию,
разновидности, значения по умолчанию

Аргументы функции

Аргументы функции – значения, которые передаются в функцию.

- Функция может принимать любое количество аргументов (даже неопределенное)
- В Python есть 2 типа передачи аргументов:
  - Позиционный
  - С помощью ключевых слов
- Сперва необходимо определять позиционным ме
  ключевыми словами

```
1  def func(a1, a2, a3, a4, a5=None, a6=None):
2      print(locals())
3
4
5  func(1, 3, a4=2, a3=6, a6=7)
6  # {'a1': 1, 'a2': 3, 'a3': 6,
7  # 'a4': 2, 'a5': None, 'a6': 7}
8
```

Значения по умолчанию

Иногда необходимо, чтобы некоторый аргумент функции был необязательным. Тогда для него указывается значение по умолчанию. Если указано такое значение, то этот аргумент можно не передавать, а вместо него при выполнении будет использовано значение по умолчанию.

- Значение по умолчанию высчитывается, когда функция определяется, а не выполняется. Типичная ошибка – использование изменяемого типа данных

```
1  def say_phrase(text, repeat=1):
2      for i in range(repeat):
3          print(text)
4
5
6  say_phrase('I\'m blue')
7  say_phrase('Da ba dee da ba di', 7)
8
```

Произвольное количество аргументов

Иногда необходимо, чтобы функция могла принимать произвольное количество аргументов. Есть 2 способа это сделать. Первый:

- Указать одним из аргументов «*args». Данный аргумент будет представлен в виде tuple. Если не передать значения, то будет пустой tuple. Не обязательно называть данный аргумент «args», но так принято, так что настоятельно рекомендуется

```
1  def sum_all(*args):
2      print(locals())
3      return sum(args)
4
5
6  result = sum_all(1, 3, 2)
7
```

Произвольное количество аргументов

Второй способ:

- Указать одним из аргументов «**kwargs». Данный аргумент будет представлен в виде dict. Если не передать параметры в виде ключ-значение, то будет пустой dict. Не обязательно называть данный аргумент «kwargs», но так принято, так что настоятельно рекомендуется

```
1  def sum_all(*args):
2      print(locals())
3      return sum(args)
4
5
6  result = sum_all(1, 3, 2)
7
```

Порядок указания аргументов

При объявлении функции:

- Позиционные аргументы
- Аргументы со значением по умолчанию (имя=значение)
- *args
- Аргументы, которые должны передаваться только по ключевым словам (имя=значение)
- **kwargs

```
1  def some_function(arg_1, arg_2=2, *args, arg_3=None, **kwargs):
2      pass
```

Порядок указания аргументов

При вызове функции:

- Любые позиционные аргументы значение
- *итерируемый_объект
- Комбинация любых ключевых аргументов имя=значение
- **словарь

```
1  def some_function(arg_1, arg_2=2, *args, arg_3=None, **kwargs):
2      pass
3
4
5  some_function(1, 2, *(1, 2, 3), arg_3=123, **{'1': 100})
6
```

Области видимости

В Python действует правило **LEGB**, которым интерпретатор пользуется при поиске переменных:

- **L (Local)** – в локальной (внутри функции)
- **E (Enclosing)** – в локальной области объемлющих функций
- **G (Global)** – в глобальной области видимости
- **B (Built-in)** – во встроенной (Зарезервированные значения Python)

```python
num_1 = 11
num_2 = 15


def sqrt_sum(number_1, number_2):
    pow_n = 2
    print(f"Возводим числа в степень {pow_n}")

    def sqrt(some_number):
        return some_number ** 0.5

    return sqrt(number_1) + sqrt(number_2)


if __name__ == '__main__':
    print(f"Результат: {sqrt_sum(num_1, num_2)}")

```

Пространства имен

Пространства имен – своего рода разделы, внутри которых определенное имя уникально и не связано с такими же именами в других пространствах имен.

В Python есть два пространства имен:

- Локальное
- Глобальное

```python
some_val = 123
print(locals())
print(globals())


def some_func(a=1):
    some_val = 234
    loc_var = 111
    print(locals())
    print(globals())

    def inner_func(c=2):
        some_val = 234
        inner_var = 222
        print(locals())
        print(globals())
        return some_val

    inner_func()

    return some_val

some_func()
```

Локальное и глобальное пространство имен

В Python есть 2 функции, с помощью которых можно посмотреть, что в локальном и глобальном пространствах имен:

- **locals()** – возвращает словарь, который содержит имена и значения локального пространства имен
- **globals()** – возвращает словарь, который содержит имена и значения глобального пространства имен

```
1  global_var = 123
2
3
4  def some_function():
5      local_var = 321
6      print(globals())
7      print(locals())
8
```

Ключевое слово global

Если у нас есть 2 переменные, одна в глобальном пространстве, а вторая в локальном, то

- При присвоении переменной в локальной области – будет создана переменная в локальной области, значение в глобальной области не будет изменено
- Если нам нужно изменять значение из глобальной области, то нужно воспользоваться ключевым словом **global**

Если не использовать global, то Python берет имя из локального пространства имен, работает с локальной переменной и после того, как функция выполнит свою работу, **значение пропадает**

```
1  some_val = 123
2
3
4  def local_val():
5      some_val = 24
6      return some_val
7
8
9  local_val()
10 print(some_val)
11
12
13 def global_val():
14     global some_val
15     some_val = 35
16     return some_val
17
18
19 global_val()
20 print(some_val)
21
```

Генераторы, декораторы и рекурсия

итерируемся по бесконечной последовательности, изменяем поведение нашего кода, вызываем функцию из себя же

Итерация и итерируемый объект

Итерация - это общий термин, который описывает процедуру взятия элементов чего-либо по очереди.

В более общем смысле, это последовательность инструкций, которая повторяется определенное количество раз или до выполнения указанного условия.

Итерируемый объект (iterable) - это объект, который способен возвращать элементы по одному. Кроме того, это объект, из которого можно получить итератор.

Примеры итерируемых объектов:

- все последовательности: список, строка, кортеж, множество, словарь, frozenset
- range()
- enumerate()
- файлы

```python
some_list = [1, 2, 3]

for i in some_list:
    print(i)


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


for i, val in enumerate(some_list):
    print(f"list[{i}]: {val}")
```

Генераторы

Генераторы – это специальный класс функций, который позволяет легко создавать свои итераторы (последовательности)

- Генератор может запоминать некоторое промежуточное состояние
- Напоминает функцию, но для генерации значения (возврата) используется не return, а yield
- По генератору можно проитерироваться только один раз, после возникает исключение StopIteration

```
1  def odd_numbers(n):
2      for i in range(n):
3          yield 2 * i + 1
4
5
6  for item in odd_numbers(4):
7      print(item)
8
```

Вопрос-ответ

? Зачем мне писать генераторы, если я могу написать функцию?

✔ В отличие от функций, с помощью генератора можно проитерироваться («пройтись») по последовательности, не создавая ее сразу, тем самым экономя ресурсы компьютера

Как работать с генераторами

Для работы с генераторами, выполняем следующие действия:

- Пишем генератор
- Присваиваем результат его выполнения переменной
- Итерируемся с помощью функции next

Либо итерируемся с помощью цикла

```python
def odd_numbers(n):
    for i in range(n):
        yield 2 * i + 1


for item in odd_numbers(4):
    print(item)


odd = odd_numbers(2)

next(odd)  # 1
next(odd)  # 3
next(odd)  # StopIteration

```

Может быть несколько yield

В одном генераторе может быть несколько инструкций yield.

Код будет выполняться последовательно, от одной инструкции к другой

```
1  def some_generator():
2      print('Start')
3      yield 1
4      print('Between')
5      yield 2
6      print('After')
7
8
9  some_gen = some_generator()
10 next(some_gen)
11 next(some_gen)
12 next(some_gen)
13
```

Конструкция yield from

- Используется для передачи (делегирования) управления между генераторами
- До Python 3.5 также использовалось, чтобы дождаться ответа от корутины, но когда добавили asyncio в стандартную библиотеку, используется только для генераторов. Для корутин «async def», «await»

```
1  def positive(n):
2      number = 1
3      while number <= n:
4          yield number
5          number += 1
6
7
8  def negative(n):
9      number = -1
10     while number >= -n:
11         yield number
12         number -= 1
13
14
15 def some_generator():
16     yield from positive(3)
17     yield from negative(3)
18
19
20 for item in some_generator():
21     print(item)
```

Функция – объект первого класса

Всё в Python – **объект**. **Функция** – тоже **объект**. Это дает питону такую гибкость, которая многим языкам не под силу (например генерация кода «на лету»).

Из этого следует:

- Функцию можно присвоить переменной (не результат, а именно саму функцию)
- Функцию можно передать в качестве аргумента другой функции (используется при написании декораторов)

```
1  def print_1():
2      print('1')
3
4
5  def print_2():
6      print('2')
7
8
9  func_switch = {
10     1: print_1,
11     2: print_2
12 }
13
14 some_func = func_switch.get(1)
15
16 if some_func is not None:
17     some_func()
18
```

Декоратор

Иногда нужно изменить поведение функции не меняя ее тела.

Декоратор – это функция, которая позволяет обернуть другую функцию для расширения ее функционала без непосредственного изменения ее кода.

```
1  def decorator(func):
2      def wrapper():
3          print("зашли в функцию-обертку")
4          print("делаем какие-то действия до")
5          result = func()
6          print("делаем что-то после")
7          print("выходим из обертки...")
8          return result
9      return wrapper
10
11 @decorator
12 def some_func():
13     print("выполняется функция")
14
```

Что потребуется, чтобы писать декораторы

При написании декоратора применяются следующие понятия и инструменты:

- *args, **kwargs
- Передача функции в качестве аргумента
- Внутренние функции и замыкания

Существует два вида декораторов:

- Обычный
- С пробросом аргументов

Пишем декоратор

1. Определяем функцию – декоратор, которая принимает декорируемую функцию
2. Внутри нее определяем функцию-обертку (функция-обертка принимает те же аргументы, что и декорируемая функция)
3. Внутри функции обертки присваиваем переменной результат выполнения декорируемой функции
4. Возвращаем из обертки результат
5. Из декоратора возвращаем

```python
def decorator(func):
    def wrapper(*args, **kwargs):
        print('Выполняем действия до функции')
        result = func(*args, **kwargs)
        print('Выполняем действия после функции')
        return result
    return wrapper


@decorator
def hello(name):
    print(f"Hello, {name}")

```

Пишем декоратор с пробросом аргументов

1. Определяем функцию, которая принимает параметры из декоратора
2. Внутри этой функции пишем декоратор, как на предыдущем слайде
3. Возвращаем декоратор из функции (без круглых скобок)

```python
import time


def delay(seconds=1):
    def decorator(func):
        def wrapper(*args, **kwargs):
            print('Выполняем действия до функции')
            time.sleep(seconds)
            result = func(*args, **kwargs)
            print('Выполняем действия после функции')
            return result
        return wrapper
    return decorator


@delay(4)
def hello(name):
    print(f"Hello, {name}")

```

Порядок выполнения декораторов

Функция может быть обернута несколькими декораторами.

Выполняются в таком же порядке, как они написаны.

```python
def make_bold(fn):
    def wrapped():
        return f"<b>{fn()}</b>"
    return wrapped


def make_italic(fn):
    def wrapped():
        return f"<i>{fn()}</i>"
    return wrapped


@make_bold
@make_italic
def hello():
    return "Hello World"


# выведет <b><i>Hello World</i></b>
print(hello())

```

P.S.

- Декоратор можно написать не только с помощью функций, но и также с помощью классов, но запись будет длиннее, а выполняется он дольше. В подавляющем числе случаев вы будете писать через функции
- Если декоратор ничего не возвращает, либо возвращает None – скорее всего что-то не так в архитекторе и декоратор используется не по назначению. Декоратор, возвращающий функцию без изменения – нормально, если возвращает None – обескураживает. Избегайте такого кода и старайтесь всегда из декоратора возвращать функцию. Такое поведение также будет более очевидным для других разработчиков

Рекурсия

Из одной функции можно вызвать другую. Но помимо этого, из функции можно вызвать эту же функцию. Процесс, когда в своем теле функция вызывает себя же, называется рекурсией

```
1  def decode_utf(data):
2      if isinstance(data, bytes) or isinstance(data, bytearray):
3          data = data.decode('utf-8')
4      elif isinstance(data, list):
5          for key, value in enumerate(data):
6              data[key] = decode_utf(value)
7      elif isinstance(data, dict):
8          data = {
9              decode_utf(key): decode_utf(value) for key, value in data.items()
10         }
11     return data
12
```