

Python

Занятие #3. Функции

О чем это занятие:

- ☐ Как функция определяется и вызывается
- ☐ Виды параметров и способы их задания/передачи
- ☐ Области видимости переменных
- ☐ Анонимные функции
- ☐ Замыкания

Теги:

определение функции, type hinting, вызов функции, параметры функции, области видимости, *args, **kwargs, global, local, lambda-функции, замыкания

Краткий обзор темы

Определение функции: **def + имя функции**

Спецификация параметров с **type hinting** – можно аннотировать типы переменных, полей класса, аргументов и возвращаемых значений функций

docstring – справочное описание функции и параметров

Вызов функции: имя, параметры

Результат вызова: return со значением и неявный return

Параметры функции

- ❑ записываются в формате snake_case
- ❑ могут быть любого типа, в том числе функцией, объектом класса или типом
- ❑ передаются по ссылке

Параметры функции: обязательные и опциональные; позиционные и именованные

***args** — для неопределенного количества позиционных параметров, собирая их в один кортеж

****kwargs** — преобразует некоторое количество пар название=значение именованных параметров в dict

Области видимости:

Глобальная (global) – уровень модуля

Локальная (local) – уровень функции/класса

Анонимные функции (lambda-функции)


Замыкания – это функции, внутри себя содержащие другую функцию

Определение и вызов функций

Определение

Для определения функции используем:

- ❑ ключевое слово **def** + **имя функции** в формате snake_case
- ❑ при желании включаем спецификацию формальных параметров с [type hinting](#)
- ❑ при желании оформляем **docstring** в тройных кавычках (документационная строка для описания функции, ее параметров и т.д.)
- ❑ **Тело функции** – последовательность выражений

 **snake_case** – стиль написания составных слов, при котором несколько слов разделяются символом подчеркивания (_), и не имеют пробелов в записи, причём каждое слово обычно пишется с маленькой буквы

Пример полного определения функции:

```
1 def the_fullest(first_param: int, second_param: str) -> str:
2     """Make some useful stuff."""
3     return first_param * second_param
```

 Хороший тон программирования: имена переменных должны отражать их суть

Вызов

Для вызова функции:

- задаем имя функции
- указываем фактические параметры в круглых скобках
- получаем результат

Результатом может быть:

- то, что указано после ключевого слова `return` в теле функции. Если после `return` ничего не указано, то вернется значение `None`
- значение `None`, если в теле нет `return` (т.н. неявный `return`)

Пример с неявным return

```
1 def implicit_return(good_number: int):
2     print('The good number is', good_number)
3
4 result = implicit_return(42)
5 Out: The good number is 42
6
7 print(result)
8 Out: None
```

Пример return без значения

```
1 def return_without_value(bad_number: int):  
2     print('The bad number is', bad_number)  
3     return  
4
```

 Пример return со значением

```
5 result = return_without_value(666)  
6 Out: The bad number is 666  
7  
8 print(result)  
None
```

```
1 def return_with_value(any_number: int):  
2     print('The specified number is', any_number)  
3     return any_number  
4  
5 result = return_with_value(1024)  
6 Out: The specified number is 1024  
7  
8 print(result)  
Out: 1024
```

Type hinting

Type Hinting — это механизм, который позволяет явно указывать типы параметров.

Python — язык с динамической типизацией и позволяет не указывать тип переменных. Но версии Python начиная с 3.5 поддерживают аннотации типов переменных, полей класса, аргументов и возвращаемых значений функций.

Для функции можно указать типы параметров и возвращаемого значения:

- ❑ встроенные (`str`, `int`, `tuple`, `dict` и т.д.)
- ❑ из модуля `typing` (позволяет встроить любые типы)

Аннотации типов используются в статических анализаторах (например, `mypy`), при этом поведение разных анализаторов может отличаться.

И если в Pycharm будет только указано на несоответствие ожидаемого и передаваемого типов, то, например, в `mypy` несоответствие типов может быть причиной ошибки.

👉 Использование `type hinting` позволяет сделать код более прозрачным и понятным для других (и получить плюсики в карму).

Пример

Реализуем функцию, удаляющую из полученного списка (или кортежа) дубликаты, используя type hinting:

```
1 from typing import Iterable
2 def remove_duplicates(source_list:Iterable) -> Iterable:
3     """Removes duplicates from the source_list."""
4     return list(set(source_list))
```

Встроенного типа `Iterable` в Python нет, но его можно импортировать из модуля `typing`.

В качестве параметра функции можно задать один из встроенных типов, это делается с помощью конструкции `Union`:

```
1 from typing import Union, List, Tuple
2 def remove_duplicates(source_list:Union[List,Tuple])
```

Параметры функции

Синтаксис

Параметры при описании функции:

- ❑ записываются в формате `snake_case`

- ❑ могут быть любого типа, в том числе функцией, объектом класса или типом
- ❑ передаются по ссылке (кроме простых неизменяемых типов `int`, `str` и т.д., которые передаются по значению)

👉 В Python есть возможность менять порядок параметров при указании имен, для этого необходимо указывать все имена

Виды параметров

Параметры функции по необходимости задания при вызове можно разделить на

- ❑ Обязательные
- ❑ Опциональные – значения заданы по умолчанию

По способу указания параметры можно разделить на

- ❑ Позиционные – порядок соответствует порядку параметров в определении функции
- ❑ Именованные – порядок задается парами **название=значение**

"Специальные" варианты – с помощью т.н. `*args` (сокращение от arguments) и `**kwargs` (сокращение от keyword arguments)

👉 Названия после символов `*` и `**` — стандартные, но не жестко закреплены, и их можно менять под свои конкретные задачи

Порядок определения параметров


1. Позиционные

2. Опциональные
3. *args
4. **kwargs

Пример

```
1 def say_hello(name: str, border: str = '-') -> str:
2     """Wraps the `name` with the specified `border`."""
3     return f'{border} Hello, {name} {border}'
4
5 print(say_hello('dvshulyak'))
6 Out: - Hello, dvshulyak -
7
8 print(say_hello('dvshulyak', '=='))
9 Out: == Hello, dvshulyak =
10
11 print(say_hello(border='*', name='dvshulyak'))
12 Out: * Hello, dvshulyak *
13
```

Здесь: username – позиционный обязательный параметр, border – опциональный с заданным значением по умолчанию.

 В первом приведенном примере используется значение параметра border по умолчанию, во втором – оба параметра передаются в исходном порядке, в третьем – порядок параметров изменен за счет того, что для каждого параметра указано значение. Однако, **третий вариант наименее предпочтителен**.

Параметры.*args

- ❑ Представляет собой **tuple** (кортеж)
- ❑ В определении функции "упаковывает" последовательность значений позиционных параметров
Допустим, функция принимает заранее неизвестное число однотипных параметров. В этом случае в определении функции параметр *args собирает их в один кортеж.
- ❑ В вызове функции "распаковывает" **tuple** в последовательность значений позиционных параметров
Можно указать после * любой итерируемый объект, и этот объект преобразуется в несколько значений через запятую.

Пример:

```
1 def sum_squares(initial_value: int = 0, *other_values) -> int:
2     """Calculates the sum of squares of specified values."""
3     all_values = (initial_value, ) + other_values
4     return sum(v ** 2 for v in all_values)
5
6 print(sum_squares())
7 Out: 0
8
9 print(sum_squares(1, 3, 10))
10 Out: 110
11
12 print(sum_squares(*[8, 10, 11]))
13 Out: 285
```


Пример:

Реализуем функцию `users_list`, получающую на вход список пользователей и выдающую некоторую информацию о них

```
1 def show_stats(user: dict):  
2     print (f'Some kind of magic with {user}')
```



```
3  
4 def users_list(first_user: dict, *users):  
5     some_users_iterator = (first_user, ) + users  
6     for one_user in some_users_iterator:  
7         show_stats(one_user)
```

 Здесь параметр `first_user` введен для того, чтобы в функцию точно был передан хотя бы один элемент. Количество остальных не регламентируется, их может не быть совсем.

 Чаще всего `*args` используется в функциях для необязательных параметров

Параметры. **kwargs

- ❑ Представляет собой dict (словарь)
- ❑ В определении "упаковывает" пары **название=значение** именованных параметров
- ❑ В вызове "распаковывает" dict в пары **название=значение** именованных параметров

Пример:

Передаем в функцию список пользователей, которых хотим экспортировать в файл какого-то формата (например, .csv).

```
1 def format_users(users: list, delimiter: str = ';', **kwargs) -> str:
2     """Formats the `users` with specified options."""
3     prefix = kwargs.get('prefix', '-')
4     num_spaces = kwargs.get('num_spaces', 1)
5     spaces = ' ' * num_spaces
6     lines = (
7         f'{prefix}{spaces}{user["name"]}{delimiter}{spaces}{user["age"]}'
8         for user in users
9     )
10    return '\n'.join(lines)
11 sample = [
12     {'name': 'aatest', 'age': 25},
13     {'name': 'bbtest', 'age': 36},
```

```
14 ]
15
16 print(format_users(sample, num_spaces=2, prefix='**'))
17 Out:
18 ** aatest; 25
19 ** bbtest; 36
20
```

💬 Здесь в `**kwargs` собраны параметры для форматирования списка, которые не вынесены в отдельные параметры функции, в данном случае это `users` и `delimiter`.

👉 В идеале функция в Python не должна иметь более 7 отдельных параметров, поэтому `**kwargs` очень помогают.

👉 Если в базе данных некоторые поля не заполнены, то корректно обработать её также помогут функции с `**kwargs`.

Области видимости

При разработке встречаются ошибки доступа к имени, неизвестному на данный момент, хотя формально в коде имя присутствует. Это происходит из-за выхода из зоны видимости переменной.

Области видимости задают синтаксический контекст — определяют, какие функции/классы/модули имеют доступ к конкретному имени.

Область видимости бывает:

- ☐ Глобальная (global) – уровень модуля
- ☐ Локальная (local) – уровень функции/класса
- ☐ Нелокальная (nonlocal) – уровень функции внутри другой функции (замыкания)

Глобальная область видимости

Имя “видно” всем функциям/классам текущего модуля, а также другим модулям. Глобальными могут быть не только имена переменных, но и имена функций, классов и констант.

👉 Глобальными часто объявляются константы – в этом случае имя пишется в формате *SNAKE_CASE*

✗ Объявлять глобальные переменные крайне нежелательно, т.к. при их изменении в любой части кода могут произойти нежелательные изменения.

Локальная область видимости

Имя “видно” в функции/классе, в котором определено. Это относится в том числе и к параметрам функций.

👉 При совпадении имен локальной и глобальной переменных Python переопределяет глобальную переменную внутри функции, но на выходе она перестает быть видимой

✗ Локальные переменные не могут быть импортированы из другого модуля

💻 Пример использования глобальной переменной

```
1 program_name = 'python course'
2 def show_login(username: str) -> str:
3     return f'Logged in to {program_name} as {username}'
4
5 print(show_login('dvshulyak'))
6
7 Out: Logged in to python course as dvshulyak
```

Пример использования локальной переменной

```
1 program_name = 'python course'
2 def show_message(name: str):
3     program_name = 'ml course'
4     return f'Hey {name}, you had better learn {program_name}'
5
6 print(show_message('aatest'))
7 Out: Hey aatest, you had better learn ml course
8
9 print(program_name)
10 Out: python course
11
12 print(name)
13 Out:
14 -----
15 NameError                                Traceback (most recent call last)
16 <ipython-input-23-a6df61a0978b> in <module>
17     6 print(show_message('aatest'))
18     7 print(program_name)
19 ----> 8 print(name)
20
21 NameError: name 'name' is not defined
```

 Внутри функции у переменной `program_name` значение поменялось, но за её пределами – осталось прежним

👉 Если в теле функции нет переопределения глобальной функции, Python обращается к глобальной области видимости, если же переопределение есть, то без ключевого слова `global` перед именем глобальной переменной код работать не будет

```
1 program_name = 'python course'
2
3 def show_message():
4     global program_name
5     a=program_name
6     program_name='ml course'
7     a=program_name
8     print(f'A:{a}')
9     print(f'Program name:{program_name}')
10
11 show_message()
12
13 Out:
14 A: python course
15 Program name: ml course
```

Анонимные функции

- ❑ Не имеют имени
- ❑ Определяются с помощью ключевого слова `lambda`
- ❑ Тело функции состоит из единственного выражения
- ❑ Имеют доступ к внешнему синтаксическому контексту

- ❑ Удобны для одноразового использования
- ❑ Полезны там, где в качестве параметра – функция
- ❑ Привносят функциональный стиль в царство Python
- ❑ Нетестопригодны

Замыкания

Функция, которая внутри себя имеет другую вложенную функцию.

Вложенная функция

- Имеет доступ к параметрам внешней функции
- Может получить доступ к внешнему контексту через `nonlocal`

Замыкания активно используются для **декораторов**