# Advanced Python для сетевых инженеров

**Natasha Samoylenko**

**мая 30, 2023**

# Оглавление

В книге рассматриваются продвинутые возможности Python на примерах для сетевых инженеров.

# 1

# I. Полезные модули

# 1. Основы pytest

Pytest - фреймворк для тестирования кода

Тестирование кода позволяет проверить:

- работает ли код так как нужно
- как ведет себя код в нестандартных ситуациях
- пользовательский интерфейс
- ...

Уровни тестирования

- unit - тестирование отдельных функций/классов
- intergration - тестирование взаимодействия разных частей софта друг с другом
- system - тестируется вся система, для web, например, это может быть тестирование от логина пользователя до выхода

Альтернативы pytest:

- unittest
- doctest
- nose

Пример тестов с unittest:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
if __name__ == '__main__':
    unittest.main()
```

Аналогичный тест с pytest:

```
def test_upper():
    assert 'foo'.upper() == 'FOO'


def test_isupper():
    assert 'FOO'.isupper() == True
    assert 'Foo'.isupper() == False


def test_split():
    s = 'hello world'
    assert s.split() == ['hello', 'world']
    # check that s.split fails when the separator is not a string
    with pytest.raises(TypeError):
        s.split(2)
```

## Основы pytest

Для начала надо установить pytest:

```
pip install pytest
```

Например, есть следующий код с функцией check_ip:

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False


if __name__ == "__main__":
    result = check_ip('10.1.1.1')
    print('Function result:', result)
```

Функция check_ip проверяет является ли аргумент, который ей передали, IP-адресом. Пример вызова функции с разными аргументами:

```
In [1]: import ipaddress
   ...:
   ...:
   ...: def check_ip(ip):
   ...:     try:
   ...:         ipaddress.ip_address(ip)
   ...:         return True
   ...:     except ValueError as err:
   ...:         return False
   ...:

In [2]: check_ip('10.1.1.1')
Out[2]: True

In [3]: check_ip('10.1.')
Out[3]: False

In [4]: check_ip('a.a.a.a')
Out[4]: False

In [5]: check_ip('500.1.1.1')
Out[5]: False
```

Теперь необходимо написать тест для функции check_ip. Тест должен проверять, что при передаче корректного адреса, функция возвращает True, а при передаче неправильного аргумента - False.

Чтобы упростить задачу, тест можно написать в том же файле. В pytest, тестом может быть обычная функция, с именем, которое начинается на `test_`. Внутри функции надо написать условия, которые проверяются. В pytest это делается с помощью assert.

### assert

assert ничего не делает, если выражение, которое написано после него истинное и генерирует исключение, если выражение ложное:

```
In [6]: assert 5 > 1

In [7]: a = 4

In [8]: assert a in [1,2,3,4]

In [9]: assert a not in [1,2,3,4]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-9-1956288e2d8e> in <module>
----> 1 assert a not in [1,2,3,4]

AssertionError:

In [10]: assert 5 < 1
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-10-b224d03aab2f> in <module>
----> 1 assert 5 < 1

AssertionError:
```

После assert и выражения можно писать сообщение. Если сообщение есть, оно выводится в исключении:

```
In [11]: assert a not in [1,2,3,4], "а нет в списке"
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-11-7a8f87272a54> in <module>
----> 1 assert a not in [1,2,3,4], "а нет в списке"

AssertionError: а нет в списке
```

### Пример теста

pytest использует assert, чтобы указать какие условия должны выполняться, чтобы тест считался пройденным.

В pytest тест можно написать как обычную функцию, но имя функции должно начинаться с `test_`. Ниже написан тест test_check_ip, который проверяет работу функции check_ip, передав ей два значения: правильный адрес и неправильный, а также после каждой проверки написано сообщение:

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        return False


def test_check_ip():
    assert check_ip('10.1.1.1') == True, 'При правильном IP, функция должна возвращать␣
↪True'
    assert check_ip('500.1.1.1') == False, 'Если адрес неправильный, функция должна␣
↪возвращать False'


if __name__ == "__main__":
    result = check_ip('10.1.1.1')
    print('Function result:', result)
```

Код записан в файл check_ip_functions.py. Теперь надо разобраться как вызывать тесты. Самый простой вариант, написать слово pytest. В этом случае, pytest автоматически обнаружит тесты в текущем каталоге. Однако, у pytest есть определенные правила, не только по названию функцию, но и по названию файлов с тестами - имена файлов также должны начинаться на test_. Если правила соблюдаются, pytest автоматически найдет тесты, если нет - надо указать файл с тестами.

В случае с примером выше, надо будет вызвать такую команду:

```
$ pytest check_ip_functions.py
========================== test session starts ==========================
platform linux -- Python 3.7.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/general/pyneng.github.io/code_examples/pytest
collected 1 item

check_ip_functions.py .                                            [100%]

======================== 1 passed in 0.02 seconds ========================
```

По умолчанию, если тесты проходят, каждый тест (функция test_check_ip) отмечается точкой. Так как в данном случае тест только один - функция test_check_ip, после имени check_ip_functions.py стоит точка, а также ниже написано, что 1 тест прошел.

Теперь, допустим, что функция работает неправильно и всегда возвращает False (напишите return False в самом начале функции). В этом случае, выполнение теста будет выглядеть так:

```
$ pytest check_ip_functions.py
========================== test session starts ==========================
platform linux -- Python 3.6.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/general/pyneng.github.io/code_examples/pytest
collected 1 item

```

(continues on next page)

(продолжение с предыдущей страницы)

```
check_ip_functions.py F                                                  [100%]

============================== FAILURES ==============================
_____________________________ test_check_ip _____________________________

    def test_check_ip():
>       assert check_ip('10.1.1.1') == True, 'При правильном IP, функция должна␣
↪возвращать True'
E       AssertionError: При правильном IP, функция должна возвращать True
E       assert False == True
E        +  where False = check_ip('10.1.1.1')

check_ip_functions.py:14: AssertionError
======================== 1 failed in 0.06 seconds ========================
```

Если тест не проходит, pytest выводит более подробную информацию и показывает в каком месте что-то пошло не так. В данном случае, при выполении строки `assert check_ip('10.1.1.1') == True`, выражение не дало истинный результат, поэтому было сгенерировано исключение.

Ниже, pytest показывает, что именно он сравнивал: `assert False == True` и уточняет, что False - это `check_ip('10.1.1.1')`. Посмотрев на вывод, можно заподозрить, что с функцией check_ip что-то не так, так как она возвращает False на правильном адресе.

Чаще всего, тесты пишутся в отдельных файлах. Для данного примера тест всего один, но он все равно вынесен в отдельный файл.

Файл test_check_ip_function.py:

```
from check_ip_functions import check_ip


def test_check_ip():
    assert check_ip('10.1.1.1') == True, 'При правильном IP, функция должна возвращать␣
↪True'
    assert check_ip('500.1.1.1') == False, 'Если адрес неправильный, функция должна␣
↪возвращать False'
```

Файл check_ip_functions.py:

```
import ipaddress


def check_ip(ip):
    #return False
    try:
        ipaddress.ip_address(ip)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        return True
    except ValueError as err:
        return False


if __name__ == "__main__":
    result = check_ip('10.1.1.1')
    print('Function result:', result)
```

В таком случае, тест можно запустить не указывая файл:

```
$ pytest
================= test session starts ======================
platform linux -- Python 3.6.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/general/pyneng.github.io/code_examples/pytest
collected 1 item

test_check_ip_function.py .                               [100%]

================= 1 passed in 0.02 seconds ====================
```

## Примеры тестов

Тут примеры тестов еще не используют fixture и параметризацию.

### Тест функции

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError:
        return False
```

Тесты

```
from basics_01_check_ip import check_ip


def test_check_ip_correct_10_1_1_1():
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    assert (
        check_ip("10.1.1.1") == True
    ), "При правильном IP, функция должна возвращать True"


def test_check_ip_correct_180_10_1_1():
    assert (
        check_ip("180.10.1.1") == True
    ), "При правильном IP, функция должна возвращать True"


def test_check_ip_wrong_octet():
    assert (
        check_ip("10.400.1.1") == False
    ), "При неправильном IP, функция должна возвращать False"


def test_check_ip_wrong_number_of_octets():
    assert (
        check_ip("10.1.1") == False
    ), "При неправильном IP, функция должна возвращать False"
```

### Тест класса

```
import ipaddress


class IPAddress:
    def __init__(self, ip, mask):
        self.ip = ip
        self.mask = mask

    def __int__(self):
        int_ip = int(ipaddress.ip_address(self.ip))
        return int_ip

    def __str__(self):
        return f"{self.ip}/{self.mask}"

    def __repr__(self):
        return f"IPAddress('{self.ip}', {self.mask})"

    def __lt__(self, second_ip):
        if type(second_ip) != IPAddress:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            raise TypeError(f"'<' not supported between instances of 'IPAddress'"
                            f" and '{type(second_ip).__name__}'")
        return (int(self), self.mask) < (int(second_ip), second_ip.mask)

    def __le__(self, second_ip):
        if type(second_ip) != IPAddress:
            raise TypeError(f"'<=' not supported between instances of 'IPAddress'"
                            f" and '{type(second_ip).__name__}'")
        return (int(self), self.mask) <= (int(second_ip), second_ip.mask)

    def __eq__(self, second_ip):
        # print("eq", self, second_ip)
        if type(second_ip) != IPAddress:
            raise TypeError(f"'==' not supported between instances of 'IPAddress'"
                            f" and '{type(second_ip).__name__}'")
        return (int(self), self.mask) == (int(second_ip), second_ip.mask)
```

Тесты:

```
from class_ipaddress import IPAddress
import pytest


def test_ipaddress_attrs():
    ip1 = IPAddress("10.1.1.1", 25)
    assert ip1.ip == "10.1.1.1"
    assert ip1.mask == 25


def test_ipaddress_str_repr():
    ip1 = IPAddress("10.1.1.1", 25)
    assert str(ip1) == "10.1.1.1/25"
    assert repr(ip1) == "IPAddress('10.1.1.1', 25)"


def test_ipaddress_int():
    ip1 = IPAddress("10.1.1.1", 25)
    assert int(ip1) == 167837953


def test_ipaddress_cmp_basic():
    ip1 = IPAddress("10.2.1.1", 25)
    ip2 = IPAddress("10.10.1.1", 25)
    assert ip1 < ip2
    assert ip2 > ip1
    assert ip1 != ip2
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    assert not ip1 == ip2
    assert ip1 <= ip2
    assert ip2 >= ip1


def test_ipaddress_cmp_mask():
    ip1 = IPAddress("10.2.1.1", 24)
    ip2 = IPAddress("10.2.1.1", 25)
    assert ip1 < ip2
    assert ip2 > ip1
    assert ip1 != ip2
    assert not ip1 == ip2
    assert ip1 <= ip2
    assert ip2 >= ip1


def test_ipaddress_cmp_equal():
    ip1 = IPAddress("10.2.1.1", 24)
    ip2 = IPAddress("10.2.1.1", 24)
    assert ip1 == ip2


def test_ipaddress_cmp_raise():
    ip1 = IPAddress("10.2.1.1", 24)
    ip2 = 100
    with pytest.raises(TypeError):
        ip1 == ip2
```

## Запуск тестов

Запуск тестов из конкретного файла:

```
$ pytest test_check_password.py
========================== test session starts ==========================
platform linux -- Python 3.7.3, pytest-5.2.0, py-1.8.0, pluggy-0.12.0
rootdir: /home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_
↪pytest_basics
collected 3 items

test_check_password.py ...                                          [100%]

=========================== 3 passed in 0.02s ===========================
```

Запуск всех тестов:

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.7.3, pytest-5.2.0, py-1.8.0, pluggy-0.12.0
rootdir: /home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_
↪pytest_basics
collected 9 items

test_check_ip_function.py .                                        [ 11%]
test_check_password.py ...                                         [ 44%]
test_ipv4_network.py .....                                         [100%]

=========================== 9 passed in 0.07s ===========================
```

Запуск тестов с более подробной информацией:

```
$ pytest -v
========================== test session starts ==========================
platform linux -- Python 3.7.3, pytest-5.2.0, py-1.8.0, pluggy-0.12.0 -- /home/vagrant/
↪venv/pyneng-py3-7/bin/python3.7
cachedir: .pytest_cache
rootdir: /home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_
↪pytest_basics
collected 9 items

test_check_ip_function.py::test_check_ip PASSED                    [ 11%]
test_check_password.py::test_password_min_length PASSED            [ 22%]
test_check_password.py::test_password_contains_username PASSED     [ 33%]
test_check_password.py::test_password_default_values PASSED        [ 44%]
test_ipv4_network.py::test_class_created PASSED                    [ 55%]
test_ipv4_network.py::test_attributes_created PASSED               [ 66%]
test_ipv4_network.py::test_methods_created PASSED                  [ 77%]
test_ipv4_network.py::test_return_types PASSED                     [ 88%]
test_ipv4_network.py::test_address_allocation PASSED               [100%]

=========================== 9 passed in 0.08s ===========================
```

Запуск одного теста

```
$ pytest test_check_password.py::test_password_min_length -v
========================== test session starts ==========================
platform linux -- Python 3.7.3, pytest-5.2.0, py-1.8.0, pluggy-0.12.0 -- /home/vagrant/
↪venv/pyneng-py3-7/bin/python3.7
cachedir: .pytest_cache
rootdir: /home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_
↪pytest_basics
collected 1 item
```

(continues on next page)

(продолжение с предыдущей страницы)

```
test_check_password.py::test_password_min_length PASSED           [100%]

============================ 1 passed in 0.01s =============================
```

Отображение вывода на stdout:

```
$ pytest test_check_password.py -s
============================ test session starts ============================
platform linux -- Python 3.7.3, pytest-5.2.0, py-1.8.0, pluggy-0.12.0
rootdir: /home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_
↪pytest_basics
collected 3 items

test_check_password.py Проверяем пароль
Пароль для пользователя nata прошел все проверки
Пароль содержит имя пользователя
.Пароль для пользователя nata прошел все проверки
Пароль содержит имя пользователя
.Пароль слишком короткий
Пароль содержит имя пользователя
Пароль для пользователя nata прошел все проверки
.

============================ 3 passed in 0.02s =============================
```

Аналогично с verbose:

```
$ pytest test_check_password.py -v -s
============================ test session starts ============================
platform linux -- Python 3.7.3, pytest-5.2.0, py-1.8.0, pluggy-0.12.0 -- /home/vagrant/
↪venv/pyneng-py3-7/bin/python3.7
cachedir: .pytest_cache
rootdir: /home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_
↪pytest_basics
collected 3 items

test_check_password.py::test_password_min_length Проверяем пароль
Пароль для пользователя nata прошел все проверки
Пароль содержит имя пользователя
PASSED
test_check_password.py::test_password_contains_username Пароль для пользователя nata␣
↪прошел все проверки
Пароль содержит имя пользователя
PASSED
test_check_password.py::test_password_default_values Пароль слишком короткий
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Пароль содержит имя пользователя
Пароль для пользователя nata прошел все проверки
PASSED

============================ 3 passed in 0.02s ==============================
```

### Когда тесты не проходят

Вывод когда тесты не проходят

```
$ pytest test_check_password.py
========================== test session starts ==========================
platform linux -- Python 3.7.3, pytest-5.2.0, py-1.8.0, pluggy-0.12.0
rootdir: /home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_
↪pytest_basics
collected 3 items

test_check_password.py .F.                                          [100%]

=============================== FAILURES ================================
_____________________ test_password_contains_username _____________________

    def test_password_contains_username():
        assert check_passwd('nata', '12345nata', min_length=3, check_username=False)
        assert not check_passwd('nata', '12345nata', min_length=3, check_username=True)
>       assert not check_passwd('nata', '12345NATA', min_length=3, check_username=True),
↪"Если в пароле присутствует имя пользователя в любом регистре, проверка не должна пройти
↪"
E       AssertionError: Если в пароле присутствует имя пользователя в любом регистре,␣
↪проверка не должна пройти
E       assert not True
E        +  where True = check_passwd('nata', '12345NATA', min_length=3, check_
↪username=True)

test_check_password.py:12: AssertionError
------------------------- Captured stdout call --------------------------
Пароль для пользователя nata прошел все проверки
Пароль содержит имя пользователя
Пароль для пользователя nata прошел все проверки
```

Короткий вывод traceback:

```
$ pytest test_check_password.py --tb=line
========================== test session starts ==========================
```

(continues on next page)

(продолжение с предыдущей страницы)

```
platform linux -- Python 3.7.3, pytest-5.2.0, py-1.8.0, pluggy-0.12.0
rootdir: /home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_
↪pytest_basics
collected 3 items

test_check_password.py .F.                                               [100%]

================================= FAILURES =================================
/home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_pytest_
↪basics/test_check_password.py:12: AssertionError: Если в пароле присутствует имя␣
↪пользователя в любом регистре, проверка не должна пройти
====================== 1 failed, 2 passed in 0.07s =======================
```

### Остановиться после первого неудачного теста

```
$ pytest test_check_password.py --tb=line -x
========================== test session starts ===========================
platform linux -- Python 3.7.3, pytest-5.2.0, py-1.8.0, pluggy-0.12.0
rootdir: /home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_
↪pytest_basics
collected 3 items

test_check_password.py .F

================================= FAILURES =================================
/home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_pytest_
↪basics/test_check_password.py:12: AssertionError: Если в пароле присутствует имя␣
↪пользователя в любом регистре, проверка не должна пройти
====================== 1 failed, 1 passed in 0.06s =======================
```

### Показать какие тесты есть, но не запускать их

```
$ pytest --collect-only
========================== test session starts ===========================
platform linux -- Python 3.7.3, pytest-5.2.0, py-1.8.0, pluggy-0.12.0
rootdir: /home/vagrant/repos/advanced-pyneng-1/advpyneng-online-oct-nov-2019/examples/14_
↪pytest_basics
collected 9 items
<Module test_check_ip_function.py>
  <Function test_check_ip>
<Module test_check_password.py>
  <Function test_password_min_length>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  <Function test_password_contains_username>
  <Function test_password_default_values>
<Module test_ipv4_network.py>
  <Function test_class_created>
  <Function test_attributes_created>
  <Function test_methods_created>
  <Function test_return_types>
  <Function test_address_allocation>

======================== no tests ran in 0.05s =========================
```

## Параметризация теста

Очень часто в тестах нужно проверять функцию/класс/метод на разных входящих данных. Один вариант, в этом случае, будет написать несколько assert в одном тесте.

```
def check_passwd(username, password, min_length=8, check_username=True):
    if len(password) < min_length:
        print('Пароль слишком короткий')
        return False
    elif check_username and username in password:
        print('Пароль содержит имя пользователя')
        return False
    else:
        print(f'Пароль для пользователя {username} прошел все проверки')
        return True


def test_password_min_length():
    assert check_passwd('user', '12345', min_length=3) == True
    assert check_passwd('user', '123456', min_length=5) == False
    assert check_passwd('user', 'userpass', min_length=5) == False
```

Этот вариант плох тем, что теперь все три проверки считаются одним тестом и если одна из проверок не проходит, следующие не проверяются.

Параметризация тестов позволяет указать несколько наборов данных, на которых надо проверить тест:

```
import pytest


@pytest.mark.parametrize(
    ("user", "passwd", "min_len", "result"),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    [
        ("user1", "123456", 4, True),
        ("user1", "123456", 8, False),
        ("user1", "123456", 6, True),
    ],
)
def test_min_len_param(user, passwd, min_len, result):
    assert check_passwd(user, passwd, min_length=min_len) == result
```

и, что особенно удобно, каждый набор данных срабатывает как отдельный запуск теста:

```
$ pytest test_check_password.py -v
============================= test session starts ==============================
...
collected 3 items

test_check_password.py::test_min_len_param[user1-123456-4-True] PASSED   [ 33%]
test_check_password.py::test_min_len_param[user1-123456-8-False] PASSED  [ 66%]
test_check_password.py::test_min_len_param[user1-123456-6-True] PASSED   [100%]

============================== 3 passed in 0.03s ===============================
```

Пример использования параметризации теста для одного параметра:

```
import ipaddress


def is_ip_address(ip):
    if not isinstance(ip, str):
        return False
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError:
        return False


@pytest.mark.parametrize("ip", ["10.1.1.1", "224.1.1.1", "0.0.0.0"])
def test_is_ip_address_correct(ip):
    assert is_ip_address(ip) == True


@pytest.mark.parametrize("ip", ["500.1.1.1", "50.1.1", "a", 100])
def test_is_ip_address_wrong(ip):
    assert is_ip_address(ip) == False
```

Запуск теста:

```
$ pytest test_check_ip.py
======================== test session starts ======================
...
collected 9 items

test_check_ip.py::test_check_ip PASSED                          [ 11%]
test_check_ip.py::test_check_ip_correct[10.1.1.1] PASSED        [ 22%]
test_check_ip.py::test_check_ip_correct[224.1.1.1] PASSED       [ 33%]
test_check_ip.py::test_check_ip_correct[0.0.0.0] PASSED         [ 44%]
test_check_ip.py::test_check_ip_wrong[500.1.1.1] PASSED         [ 55%]
test_check_ip.py::test_check_ip_wrong[50.1.1] PASSED            [ 66%]
test_check_ip.py::test_check_ip_wrong[a] PASSED                 [ 77%]
test_check_ip.py::test_check_ip_wrong[100] PASSED               [ 88%]
test_check_ip.py::test_check_ip_wrong[ip4] PASSED               [100%]
```

## Fixture

Fixtures это функции, которые выполняют что-то до теста и, при необходимости, после.

Два самых распространенных применения fixture:

- для передачи каких-то данных для теста
- setup and teardown

Fixture scope - контролирует как часто запускается fixture:

- function (значение по умолчанию) - fixture запускается до и после каждого теста, который использует это fixture
- class
- module - fixture запускается один раз до и после тестов в модуле, который использует это fixture
- package
- session - fixture запускается один раз в начале сессии и в конце

Запуск «после» актуален только для fixture с yield.

Полезные команды для работы с fixture:

- `pytest --fixtures` - показывает все доступные fixture (встроенные, из плагинов и найденные в тестах и conftest.py). Добавление `-v` показывает в каких файлах находятся fixture и на какой строке определена функция
- `pytest --setup-show` - показывает какие fixture запускаются и когда

### Создание fixture

```
import pytest


@pytest.fixture
def topology_with_dupl_links():
    topology = {('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
                ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
                ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
                ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
                ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
                ('R3', 'Eth0/2'): ('R5', 'Eth0/0'),
                ('SW1', 'Eth0/1'): ('R1', 'Eth0/0'),
                ('SW1', 'Eth0/2'): ('R2', 'Eth0/0'),
                ('SW1', 'Eth0/3'): ('R3', 'Eth0/0')}
    return topology


@pytest.fixture
def normalized_topology_example():
    normalized_topology = {('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
                           ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
                           ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
                           ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
                           ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
                           ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
    return normalized_topology
```

```
import yaml
import pytest
from netmiko import ConnectHandler


@pytest.fixture(scope='module')
def first_router_from_devices_yaml():
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
        r1 = devices[0]
    return r1


@pytest.fixture(scope='module')
def first_router_wrong_pass(first_router_from_devices_yaml):
    r1 = first_router_from_devices_yaml.copy()
    r1['password'] = 'wrong'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    return r1


@pytest.fixture(scope='module')
def first_router_wrong_ip(first_router_from_devices_yaml):
    r1 = first_router_from_devices_yaml.copy()
    r1['ip'] = 'unreachable'
    return r1
```

Fixture r1_test_connection создает подключение netmiko до тестов в одном файле и закрывает его после:

```
@pytest.fixture(scope='module')
def r1_test_connection(first_router_from_devices_yaml):
    with ConnectHandler(**first_router_from_devices_yaml) as r1:
        r1.enable()
        yield r1
```

### Встроенные fixture

- capsys
- monkeypatch
- tmp_path
- и другие

### capsys

```
from netmiko import ConnectHandler
from paramiko.ssh_exception import AuthenticationException


def send_show_command(device, command):
    try:
        with ConnectHandler(**device) as ssh:
            ssh.enable()
            result = ssh.send_command(command)
            return result
    except AuthenticationException as error:
        print(error)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def test_function_return_value(capsys, first_router_wrong_pass):
    return_value = send_show_command(first_router_wrong_pass, "sh ip int br")
    correct_stdout = "Authentication fail"
    out, err = capsys.readouterr()
    assert out != "", "Сообщение об ошибке не выведено на stdout"
    assert correct_stdout in out, "Выведено неправильное сообщение об ошибке"
```

### monkeypatch

```
def check_passwd(username, password, min_length=8):
    if len(password) < min_length:
        print('Пароль слишком короткий')
        return False
    elif username in password:
        print('Пароль содержит имя пользователя')
        return False
    else:
        print(f'Пароль для пользователя {username} прошел все проверки')
        return True


def test_password_min_length(monkeypatch):
    monkeypatch.setattr('builtins.input', lambda x=None: 'user')
    monkeypatch.setattr('getpass.getpass', lambda x=None: '12345')
    assert check_passwd(min_length=3) == True


@pytest.mark.parametrize(
    "username,password,result",
    [
        ('user', '12345', True),
        ('user', '12345user', False)
    ],
)
def test_check_passwd_function(monkeypatch, username, password, result):
    monkeypatch.setattr('builtins.input', lambda x=None: username)
    monkeypatch.setattr('getpass.getpass', lambda x=None: password)
    assert check_passwd(min_length=3) == result
```

### conftest

conftest.py это файл в котором хранятся fixture для разных тестов. Этих файлов может быть много, например, может быть такая структура:

```
├── conftest.py
├── pytest.ini
└── tests
    ├── conftest.py
    ├── helper_test_functions.py
    ├── network
    │   ├── conftest.py
    │   └── test_11_network_fixture_params.py
    └── unit
        ├── conftest.py
        ├── test_01_check_ip.py
        ├── test_02_send_command.py
        ├── test_03_check_password.py
        ├── test_04_get_interfaces.py
        ├── test_05_class_topology.py
        └── test_06_class_ipv4network.py
```

Conftest.py также добавляет каталог в котором он находится в sys.path. Поэтому часто можно встретить пустые файлы conftest.py. Например в этом случае conftest.py в текущем каталоге может быть пустым, но он нужен чтобы тесты из каталога tests могли импортировать функции из файлов в текущем каталоге:

```
$ tree
.
├── check_ip_functions.py
├── check_password_function_input.py
├── check_password_function.py
├── class_ipv4_network.py
├── common_functions.py
├── conftest.py
└── tests
    ├── conftest.py
    ├── test_check_ip_function.py
    ├── test_check_password_input.py
    ├── test_check_password_parametrize.py
    ├── test_check_password.py
    └── test_ipv4_network.py
```

### Возможности fixture

Fixture может смотреть в тест/модуль, который «вызвал» fixture.

conftest.py

```
@pytest.fixture(scope="module")
def smtp_connection(request):
    server = getattr(request.module, "smtpserver", "smtp.gmail.com")

    smtp_connection = smtplib.SMTP(server, 587, timeout=5)
    yield smtp_connection
    print("finalizing {} ({})".format(smtp_connection, server))
    smtp_connection.close()
```

test_smtp.py

```
smtpserver = "mail.python.org"  # will be read by smtp fixture


def test_showhelo(smtp_connection):
    assert 0, smtp_connection.helo()
```

factory as fixture

```
@pytest.fixture
def make_customer_record():
    def _make_customer_record(name):
        return {"name": name, "orders": []}

    return _make_customer_record


def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

### Параметризация fixture

```
with open("devices.yaml") as f:
    devices_params = yaml.safe_load(f)
ip_list = [d["host"] for d in devices_params]


@pytest.fixture(params=devices_params, scope="session", ids=ip_list)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def ssh_connection(request):
    with ConnectHandler(**request.param) as ssh:
        ssh.enable()
        yield ssh


def test_ospf_enabled(ssh_connection):
    output = ssh_connection.send_command("sh ip ospf")
    assert "routing process" in output.lower()


def test_loopback(ssh_connection):
    output = ssh_connection.send_command("sh ip int br | i up +up")
    assert "Loopback0" in output
```

Параметризация fixture и теста:

```
with open("devices.yaml") as f:
    devices_params = yaml.safe_load(f)
ip_list = [d["host"] for d in devices_params]


@pytest.fixture(params=devices_params, scope="session", ids=ip_list)
def ssh_connection(request):
    ssh = ConnectHandler(**request.param)
    ssh.enable()
    yield ssh
    ssh.disconnect()


@pytest.mark.parametrize(
    "ip",
    ["192.168.100.100", "192.168.100.2", "192.168.100.3"],
    ids=["ISP1", "ISP2", "FW"],
)
def test_ping(ssh_connection, ip):
    output = ssh_connection.send_command(f"ping {ip}")
    assert "success rate is 100 percent" in output.lower()
```

## Дополнительные возможности

### pytest.raises

```
import ipaddress
import pytest


def check_ip(ip):
    if type(ip) != str:
        raise TypeError("Function only works with strings")
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError:
        return False


def test_check_ip_raises_1():
    with pytest.raises(TypeError):
        check_ip(100)


def test_check_ip_raises_3():
    with pytest.raises(TypeError) as error:
        check_ip(100)
    assert "strings" in str(error.value)


def test_check_ip_raises_4():
    with pytest.raises(TypeError, match="st.+ngs"):
        check_ip(100)
```

## Использование pytest для тестирования сети

conftest.py

```
import pytest
from netmiko import Netmiko
import yaml


with open("devices.yaml") as f:
    DEVICES = yaml.safe_load(f)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
DEVICES_IP = [dev["host"] for dev in DEVICES]


def get_host(device):
    return device["host"]


@pytest.fixture(params=DEVICES, ids=get_host, scope="session")
def ssh_connection(request):
    with Netmiko(**request.param) as ssh:
        ssh.enable()
        yield ssh
```

Тесты:

```
import pytest


@pytest.mark.parametrize(
    "command,check_output",
    [
        ("sh ip ospf", "routing process"),
        ("sh ip int br", "up"),
    ],
)
def test_ospf(ssh_connection, command, check_output):
    output = ssh_connection.send_command(command)
    assert check_output in output.lower()


@pytest.mark.parametrize(
    "ip_address", ["192.168.100.1", "192.168.100.100"], ids=["ISP1", "ISP2"]
)
def test_ping(ssh_connection, ip_address):
    output = ssh_connection.send_command(f"ping {ip_address}")
    assert "success rate is 100" in output.lower()


def test_loopback(ssh_connection):
    loopback = "Loopback0"
    output = ssh_connection.send_command("sh ip int br")
    assert loopback in output


def test_intf(ssh_connection):
    output = ssh_connection.send_command("sh ip int br | i up +up")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    assert output.count("up") >= 4
```

## Рекомендации/нюансы по тестам

### Много assert в тесте

Чем плохо то, что в тесте много assert - тест остановится на первом assert, который не прошел и не будет проверять другие.

---

**Примечание:** Есть плагины, которые позволяют в каком-то виде проверять все assert (или выражения, которые используются вместо assert в плагине).

---

В идеале было бы хорошо, чтобы в каждом тесте был только один assert, но так далеко не всегда есть смысл делать.

Иногда по одному результату есть много проверок. И вам может подходить то, что если одна проверка не прошла, другие не выполняются. Пример теста scrapli в котором много assert:

```
@pytest.mark.parametrize(
    "test_data",
    [(Scrapli, "system", NetworkDriver), (AsyncScrapli, "asyncssh", AsyncNetworkDriver)],
    ids=["sync_factory", "async_factory"],
)
def test_factory_community_platform_variant(test_data):
    Factory, transport, expected_driver = test_data
    driver = Factory(
        platform="scrapli_networkdriver",
        host="localhost",
        variant="test_variant1",
        transport=transport,
    )
    assert isinstance(driver, expected_driver)
    assert driver.transport_name == transport
    assert driver.failed_when_contains == [
        "% Ambiguous command",
        "% Incomplete command",
        "% Invalid input detected",
        "% Unknown command",
    ]
    assert driver.textfsm_platform == "cisco_iosxe"
    assert driver.genie_platform == "iosxe"
    assert driver.default_desired_privilege_level == "configuration"
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    assert callable(driver.on_open)
    assert callable(driver.on_close)
    for actual_priv_level, expected_priv_level in zip(
        driver.privilege_levels.values(), TEST_COMMUNITY_PRIV_LEVELS.values()
    ):
        assert actual_priv_level.name == expected_priv_level.name
        assert actual_priv_level.pattern == expected_priv_level.pattern
```

### Циклы в тестах

Циклы в тестах во многом попадают в ту же категорию, что и много assert. Однако стоит учитывать, что, если, например, в тесте проверяется подключение к нескольким устройствам в цикле и assert стоит именно в цикле по устройствам, то достаточно одному устройству не пройти assert и к остальным тест подключаться не будет. Иногда это может быть то, что нужно от теста, иногда нет.

### Правильный результат в fixture

Правильный результат, который должна была вернуть функция/метод/класс, очень часто пишется прямо в тесте (correct_access_dict, correct_trunk_dict).

```
import pytest


@pytest.fixture
def cfg_example_1():
    cfg = (
        "!\n"
        "interface FastEthernet0/0\n"
        " switchport mode access\n"
        " switchport access vlan 10\n"
        "!\n"
        "interface FastEthernet0/1\n"
        " switchport trunk encapsulation dot1q\n"
        " switchport trunk allowed vlan 100,200\n"
        " switchport mode trunk\n"
        "!\n"
        "interface FastEthernet0/2\n"
        " switchport mode access\n"
        " switchport access vlan 20\n"
        "!\n"
    )
    return cfg
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def test_cfg_1(cfg_example_1):
    correct_access_dict = {"FastEthernet0/0": 10, "FastEthernet0/2": 20}
    correct_trunk_dict = {"FastEthernet0/1": [100, 200]}
    access_dict, trunk_dict = get_int_vlan_map(cfg_example_1)
    assert access_dict == correct_access_dict and trunk_dict == correct_trunk_dict
```

Иногда результат слишком большой чтобы писать в тесте, тогда можно встретить варианты с записью параметров и результатов в файлах/структурах данных. Также часто наборы входящих параметров и результатов пишут в parametrize:

```
@pytest.mark.parametrize(
    "network,correct_net_len",
    [
        ("10.1.1.192/30", 2),
        ("10.1.1.0/28", 14),
        ("10.1.1.0/24", 254),
    ],
)
def test_len_method(network, correct_net_len):
    network = Network(network)
    assert hasattr(network, "__len__")
    assert len(network) == correct_net_len, "Метод __len__ возвращает неверное значение"
```

Fixture, как правило, используются только для подготовки данных или подготовки до теста и удаления после теста, но не для передачи правильного результата в тест.

### Проверка типов данных

В тестах можно проверять типы данных, которые возвращает функция/метод/класс, но обычно это делают не от и до, например, по всем данным словаря, а только проверяют что это словарь. Как правило, проверка типа делается чтобы ошибка была понятной, что возвращается не тот тип данных. При этом, например, не нужно проверять каждый элемент словаря, потому что при сравнении словарь == правильный словарь, все отличия покажет pytest

```
def test_1():
    correct_dict = {"FastEthernet0/0": 10, "FastEthernet0/2": 20}
    result_dict = {"FastEthernet0/0": "10", "FastEthernet0/2": "20"}
    assert result_dict == correct_dict
```

Пример вывода:

```
    def test_1():
        correct_dict = {"FastEthernet0/0": 10, "FastEthernet0/2": 20}
        result_dict = {"FastEthernet0/0": "10", "FastEthernet0/2": "20"}
>       assert result_dict == correct_dict
E       AssertionError: assert {'FastEtherne...net0/2': '20'} == {'FastEtherne...ernet0/2
↪': 20}
E         Differing items:
E         {'FastEthernet0/0': '10'} != {'FastEthernet0/0': 10}
E         {'FastEthernet0/2': '20'} != {'FastEthernet0/2': 20}
E         Use -v to get the full diff
```

Вывод с -v:

```
    def test_1():
        correct_dict = {"FastEthernet0/0": 10, "FastEthernet0/2": 20}
        result_dict = {"FastEthernet0/0": "10", "FastEthernet0/2": "20"}
>       assert result_dict == correct_dict
E       AssertionError: assert {'FastEtherne...net0/2': '20'} == {'FastEtherne...ernet0/2
↪': 20}
E         Differing items:
E         {'FastEthernet0/2': '20'} != {'FastEthernet0/2': 20}
E         {'FastEthernet0/0': '10'} != {'FastEthernet0/0': 10}
E         Full diff:
E         - {'FastEthernet0/0': 10, 'FastEthernet0/2': 20}
E         + {'FastEthernet0/0': '10', 'FastEthernet0/2': '20'}
E         ?                     +  +                     +  +
```

### Проверка True/False

Стоит ли писать в тестах `if value == True` вместо `if value`?

Если это проверка типа isinstance, например, то не надо:

```
assert isinstance(value, str)
```

Если это проверка именно того что возвращает функция, которую мы тестируем, то `== True` более явно говорит, что тут должен быть результат именно True, а не любой истинный результат

```
assert function(value) == True
```

---

**Примечание:** Речь только о тестах, для кода в целом рекомендация писать `if value` не `if value == True`

---

### Закрытие сессий/файлов в тесте

Два примера кода. Первый - сессия закрывается close:

```
def test_telnet_class(reachable_device):
    r1 = CiscoTelnet(**reachable_device)
    assert r1.prompt == ">"
    r1.close()
```

Второй - сессия закрывает в менеджере контекста:

```
def test_telnet_class(reachable_device):
    with CiscoTelnet(**reachable_device) as r1:
        assert r1.prompt == ">"
```

Очень важная разница этих вариантов в том, что менеджер контекста закроет сессию даже если assert не прошел, а close НЕ сработает.

При этом первый пример можно переделать так и тогда сессия закроется (но лучше конечно использовать менеджер контекста где возможно или fixture):

```
def test_telnet_class(reachable_device):
    r1 = CiscoTelnet(**reachable_device)
    r1_prompt = r1.prompt
    r1.close()
    assert r1_prompt == ">"
```

### Структура теста

AAA (Arrange, Act, Assert).

Тесты, как правило, можно разбить на несколько этапов:

- Arrange
- Act
- Assert
- Cleanup

При этом тест можно состоять только из первых трех шагов, если стадия cleanup не нужна.

Как правило, в pytest стадии Arrange и Cleanup делаются в fixture, а остальное в тесте.

## Дополнительные материалы

### Документация:

- pytest
- Good Integration Practices

### Статьи:

- Getting Started With Testing in Python
- Effective Python Testing With Pytest
- tox
- pytest and tox

### Альтернативы pytest

- unittest
- doctest
- nose

Полезные ссылки по сравнению фреймворков:

- Test & code podcast: 2: Pytest vs Unittest vs Nose
- The Cleaning Hand of Pytest

### Тестирование сети

- Automating «Network Ready for Use» Testing - полезное выступление об использовании pytest для тестирования сети

### Примеры тестов в модулях

Scrapli:

- Тесты scrapli.
- Пример относительно простых тестов scrapli
- Тесты Response
- Тесты со skipif

Netmiko:

- Тесты netmiko.
- Пример относительно простых тестов netmiko

# 2. Основы аннотации типов

Аннотация типов - это дополнительное описание в классах, функциях, переменных, которое указывает какой тип данных должен быть в этом месте.

При этом указанные типы не проверяются и не форсируются самим Python. То есть, при выполнении кода, несоответствие реального типа данных тому, что написано в аннотациях, не вызывает ошибок или предупреждений. Для проверки типов данных используются отдельные модули, например, mypy. Муру выполняет статический анализ кода - проверяет соответствие типов данных без выполнения кода.

---

**Примечание:** Аннотация типов добавлялась постепенно в Python 3.x. Начиная с версии Python 3.0 была доступна аннотация функций, а в Python 3.6 была добавлена аннотация для переменных.

---

Преимущества:

- при создании объектов сразу описаны типы данных
- можно проверять правильность указанных типов с помощью отдельных модулей
- IDE могут делать подсказки, указывать на ошибки на основании аннотации типов

Нюансы:

- как и с тестами, надо потратить время на написание аннотаций (хотя есть софт, который может в этом помочь)
- на данный момент, надо делать довольно большое количество импортов
- желательно использовать Python 3.6+ чтобы были доступны все возможности, в идеале, последнюю версию Python.

## Синтаксис

### Аннотация переменных

---

**Примечание:** Аннотацию не нужно будет писать абсолютно для всех переменных, так как многие типы будут автоматически вычисляться mypy.

---

Пример переменной:

```
username: str = 'user1'
```

Пример аннотации для разных встроенных типов данных:

```
length: int = 5
summ: float = 5.5
skip_line: bool = True
line: str = "switchport mode access"
```

Списки, множества, словари:

```
from typing import List, Set, Dict, Tuple, Union

vlans: List[int] = [10, 20, 100]
unique_vlans: Set[int] = {1, 6, 10}

book_price_map: Dict[str, float] = {'Good Omens': 22.0}
```

---

**Примечание:** Начиная с Python 3.9 вместо List, Set, Dict, Tuple из модуля typing, можно будет использовать встроенные объекты list, set, dict, tuple.

---

Кортеж с фиксированным количеством элементов:

```
sw_info: Tuple[str, str, int] = ("sw1", "15.1(3)", 24)
```

Кортеж с произвольным количеством элементов:

```
vlans: Tuple[int, ...] = (1, 2, 3)
```

Список с элементами разного типа:

```
sw_info: List[Union[str, int]] = ["sw1", "15.1(3)", 24]
```

Также можно создавать аннотацию переменной без значения:

```
In [1]: username: str

In [2]: username
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-407fefd38331> in <module>
----> 1 username

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

Например, этот функционал используется в Data classes чтобы указать какие атрибуты будут у экземпляров:

```
In [11]: @dataclass
    ...: class IPAddress:
    ...:     ip: str
    ...:     mask: int
    ...:

In [12]: ip1 = IPAddress('10.1.1.1', 28)

In [13]: ip1
Out[13]: IPAddress(ip='10.1.1.1', mask=28)
```

### Аннотация функции

Для параметров функции, аннотация пишется так же как для переменных, плюс добавляется возвращаемое значение:

```
import ipaddress


def check_ip(ip: str) -> bool:
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False
```

Пример аннотации функции со значениями по умолчанию:

```
def check_passwd(username: str, password: str,
                 min_length: int = 8, check_username: bool = True) -> bool:
    if len(password) < min_length:
        print('Пароль слишком короткий')
        return False
    elif check_username and username in password:
        print('Пароль содержит имя пользователя')
        return False
    else:
        print(f'Пароль для пользователя {username} прошел все проверки')
        return True
```

### Аннотация функции с исключениями

**Примечание:** https://www.python.org/dev/peps/pep-0484/#exceptions

Если функция может возвращать какой-то результат или исключения, аннотация пишется только для результата:

```
from typing import Dict, Union, List, Any


def summ(x: int, y: int) -> int:
    if isinstance(x, int) and isinstance(y, int):
        return x + y
    else:
        raise ValueError('Аргументы должны быть числами')
```

### Аннотация классов

Аннотация методов пишется так же как аннотация функций. Единственный нюанс методов - self пишется без аннотации.

```
class IPAddress:
    def __init__(self, ip: str, mask: int) -> None:
        self.ip = ip
        self.mask = mask

    def __repr__(self) -> str:
        return f"IPAddress({self.ip}/{self.mask})"
```

### Аннотация типов и наследование

Дочерний класс должен поддерживать те же типы данных, что и родительский:

```
import time
from typing import Union, List


class BaseSSH:
    def __init__(self, ip: str, username: str, password: str) -> None:
        self.ip = ip
        self.username = username
        self.password = password
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def send_config_commands(self, commands: Union[str, List[str]]) -> str:
        if isinstance(commands, str):
            commands = [commands]
        for command in commands:
            time.sleep(0.5)
        return 'result'


class CiscoSSH(BaseSSH):
    def __init__(self, ip: str, username: str, password: str,
                 enable_password: str = None, disable_paging: bool = True) -> None:
        super().__init__(ip, username, password)

    def send_config_commands(self, commands: List[str]) -> str:
        return 'result'
```

В этом случае будет ошибка:

```
$ mypy example_07_class_inheritance.py
example_07_class_inheritance.py:25: error: Argument 1 of "send_config_commands" is␣
↪incompatible with supertype "BaseSSH"; supertype defines the argument type as
↪"Union[str, List[str]]"
Found 1 error in 1 file (checked 1 source file)
```

### Протоколы

Так как в Python достаточно часто будут встречаться функции с аргументами не конкретного типа, а поддерживающие протокол, в typing есть объекты типа Iterable, Iterator и другие:

```
import ipaddress
from typing import Iterable, Iterator, List

def convert_int_to_str(integers: Iterable[int]) -> List[str]:
    return [str(x) for x in integers]

class Network:
    def __init__(self, network: str) -> None:
        self.network = network
        subnet = ipaddress.ip_network(self.network)
        self.addresses = [str(ip) for ip in subnet.hosts()]

    def __iter__(self) -> Iterator[str]:
        return iter(self.addresses)
```

### Атрибут __annotations__

Атрибут __annotations__ содержит словарь с описанием аннотации.

Для глобальных переменных, он появляется как только есть аннотация хотя бы в одной переменной:

```
In [1]: username: str

In [2]: username
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-407fefd38331> in <module>
----> 1 username

NameError: name 'username' is not defined

In [3]: __annotations__
Out[3]: {'username': str}
```

Атрибут __annotations__ в функции:

```
def check_passwd(username: str, password: str,
                 min_length: int = 8, check_username: bool = True) -> bool:
    pass

In [2]: check_passwd.__annotations__
Out[2]:
{'username': str,
 'password': str,
 'min_length': int,
 'check_username': bool,
 'return': bool}
```

В классе атрибут __annotations__ появляется в методах и в самом классе, если были созданы переменные класса:

```
class IPAddress:
        address_type: int = 4

        def __init__(self, ip: str, mask: int) -> None:
            self.ip = ip
            self.mask = mask

        def __repr__(self) -> str:
            return f"IPAddress({self.ip}/{self.mask})"
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [9]: IPAddress.__annotations__
Out[9]: {'address_type': int}

In [10]: IPAddress.__repr__.__annotations__
Out[10]: {'return': str}
```

### Справка при работе в консоли

Болшая часть объектов модуля typing имеет встроенную справку, например:

```
In [21]: ?Iterable
Signature:   Iterable(*args, **kwargs)
Type:        _GenericAlias
String form: typing.Iterable
File:        /usr/lib/python3.8/typing.py
Docstring:
The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'.
There are two kind of these aliases: user defined and special. The special ones
are wrappers around builtin collections and ABCs in collections.abc. These must
have 'name' always set. If 'inst' is False, then the alias can't be instantiated,
this is used by e.g. typing.List and typing.Dict.

In [22]: help(Union)
```

Но некоторые объекты дополнительно описаны методом _doc, например:

```
In [32]: print(Optional._doc)
Optional type.

    Optional[X] is equivalent to Union[X, None].


In [33]: print(Union._doc)
Union type; Union[X, Y] means either X or Y.

    To define a union, use e.g. Union[int, str].  Details:
    - The arguments must be types and there must be at least one.
    - None as an argument is a special case and is replaced by
      type(None).
    - Unions of unions are flattened, e.g.::
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    Union[Union[int, str], float] == Union[int, str, float]

- Unions of a single argument vanish, e.g.::

    Union[int] == int  # The constructor actually returns int

- Redundant arguments are skipped, e.g.::

    Union[int, str, int] == Union[int, str]

- When comparing unions, the argument order is ignored, e.g.::

    Union[int, str] == Union[str, int]

- You cannot subclass or instantiate a union.
- You can use Optional[X] as a shorthand for Union[X, None].
```

## Основы mypy

Так как сам Python никак не проверяет указанные типы данных, надо использовать какой-то дополнительный модуль для проверки. Один из таких модулей - mypy.

Муру выполняет статический анализ кода - проверяет соответствие типов данных без выполнения кода.

---

**Примечание:** Муру не единственный проект такого типа. Другие модули: pyre, pytype.

---

Пример запуска скрипта с помощью mypy:

```
$ mypy example_01_function_check_ip.py
example_01_function_check_ip.py:13: error: Argument 1 to "check_ip" has incompatible type
↪"int"; expected "str"
Found 1 error in 1 file (checked 1 source file)
```

Писать аннотацию для переменных нужно далеко не всегда. Как правило, того типа который «угадал» mypy достаточно. Например, в этом случае mypy понимает, что ip это строка:

```
ip = '10.1.1.1'
```

И не будет выводить никаких ошибок:

```
$ mypy example_03_variable.py

Success: no issues found in 1 source file
```

Однако, если переменная может быть и строкой и числом:

```
ip = '10.1.1.1'
ip = 3
```

mypy посчитает это ошибкой:

```
example_03_variable.py:2: error: Incompatible types in assignment (expression has type
↪"int", variable has type "str")
Found 1 error in 1 file (checked 1 source file)
```

В таком случае надо явно указать, что переменная может быть числом или строкой:

```
from typing import Union

ip: Union[int, str] = '10.1.1.1'
ip = 3
```

### strict

```
def func1(a: str, b: str) -> str:
    return a + b

def func2(c, d):
    result = func1(4, 6)
    return c + d
```

По умолчанию, mypy игнорирует функции без аннотации типов:

```
$ mypy testme.py
Success: no issues found in 1 source file
```

С параметром strict mypy проверяет эти функции и их работу с другими объектами:

```
$ mypy testme.py --strict
testme.py:4: error: Function is missing a type annotation
testme.py:5: error: Argument 1 to "func1" has incompatible type "int"; expected "str"
testme.py:5: error: Argument 2 to "func1" has incompatible type "int"; expected "str"
Found 3 errors in 1 file (checked 1 source file)
```

### reveal

reveal_type

reveal_locals:

```
def check_passwd(username: str, password: str,
                 min_length: int = 8, check_username: bool = True) -> bool:
    reveal_locals()
    if len(password) < min_length:
        print('Пароль слишком короткий')
        return False
    elif check_username and username in password:
        print('Пароль содержит имя пользователя')
        return False
    else:
        print(f'Пароль для пользователя {username} прошел все проверки')
        return True
```

```
example_02_function_check_passwd.py:4: note: Revealed local types are:
example_02_function_check_passwd.py:4: note:     check_username: builtins.bool
example_02_function_check_passwd.py:4: note:     min_length: builtins.int
example_02_function_check_passwd.py:4: note:     password: builtins.str
example_02_function_check_passwd.py:4: note:     username: builtins.str
```

## Примеры использования аннотации типов

### ignore-missing-imports

```
mypy --ignore-missing-imports example_04_class_basessh.py
```

### Отложенное вычисление аннотаций типов

---

**Примечание:** Работает в Python 3.7+ с импортом __future__

---

Использование имени класса в аннотации внутри этого же класса:

```
from __future__ import annotations
import ipaddress


```

(continues on next page)

(продолжение с предыдущей страницы)

```
class IPAddress:
    def __init__(self, ip: str) -> None:
        self.ip = ip

    def __add__(self, other: int) -> IPAddress:
        ip_int = int(ipaddress.ip_address(self.ip))
        sum_ip_str = str(ipaddress.ip_address(ip_int + other))
        return IPAddress(sum_ip_str)
```

### Опциональный аргумент

```
from typing import Union, List, Optional


def check_passwd(username: str, password: str, min_length: int = 8,
                 check_username: bool = True,
                 forbidden_symbols: Union[List, None] = None) -> bool:
                 #forbidden_symbols: Optional[List] = None) -> bool:
    if len(password) < min_length:
        print('Пароль слишком короткий')
        return False
    elif check_username and username in password:
        print('Пароль содержит имя пользователя')
        return False
    else:
        print(f'Пароль для пользователя {username} прошел все проверки')
        return True
```

## Ошибки и решения

### Ошибки при использовании type вместо isinstance

**Примечание:** Это известная проблема https://github.com/python/mypy/issues/4445

При использовании type для создания условия по типу данных таким образом:

```
from netmiko import ConnectHandler
from typing import Dict, Union, Iterable, Any
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def send_show_commands(
    device_params: Dict[str, Any], commands: Union[str, Iterable[str]]
) -> Dict[str, str]:
    result = {}
    if type(commands) == str:
        commands = [commands]
    with ConnectHandler(**device_params) as ssh:
        ssh.enable()
        for command in commands:
            output = ssh.send_command(command)
            result[command] = output
    return result
```

Муру выдаст такую ошибку:

```
$ mypy example_16_isinstance.py
example_16_isinstance.py:11: error: List item 0 has incompatible type "Union[str,␣
↪Iterable[str]]"; expected "str"
Found 1 error in 1 file (checked 1 source file)
```

Исправить ситуацию можно используя isinstance вместо type:

```
def send_show_commands(
    device_params: Dict[str, Any], commands: Union[str, Iterable[str]]
) -> Dict[str, str]:
    result = {}
    if isinstance(commands, str):
        commands = [commands]
    with ConnectHandler(**device_params) as ssh:
        ssh.enable()
        for command in commands:
            output = ssh.send_command(command)
            result[command] = output
    return result
```

### Отсутствие проверки None

Пример кода:

```
import re
from typing import List, Tuple


def parse_sh_cdp_neighbors(command_output: str) -> List[Tuple[str, ...]]:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    regex = re.compile(
        r"(?P<r_dev>\w+) +(?P<l_intf>\S+ \S+)"
        r" +\d+ +[\w ]+ +\S+ +(?P<r_intf>\S+ \S+)"
    )
    connect_list = []
    match_l_dev = re.search(r"(\S+)[>#]", command_output)
    l_dev = match_l_dev.group(1)
    for match in regex.finditer(command_output):
        neighbor = (l_dev, *match.group("l_intf", "r_dev", "r_intf"))
        connect_list.append(neighbor)
    return connect_list
```

При таком коде mypy выдаст ошибку:

```
$ mypy example_17_re_search.py
example_17_re_search.py:13: error: Item "None" of "Optional[Match[str]]" has no attribute
↪"group"
Found 1 error in 1 file (checked 1 source file)
```

Ошибка возникает потому что выражение `match_l_dev = re.search(r"(\S+)[>#]", command_output)` может возвращать None или объект Match. Без проверки что возвращается именно истинное значение, будет ошибка. Код надо исправить таким образом:

```
def parse_sh_cdp_neighbors(command_output: str) -> List[Tuple[str, ...]]:
    regex = re.compile(
        r"(?P<r_dev>\w+) +(?P<l_intf>\S+ \S+)"
        r" +\d+ +[\w ]+ +\S+ +(?P<r_intf>\S+ \S+)"
    )
    connect_list = []
    match_l_dev = re.search(r"(\S+)[>#]", command_output)
    if match_l_dev:
        l_dev = match_l_dev.group(1)
    for match in regex.finditer(command_output):
        neighbor = (l_dev, *match.group("l_intf", "r_dev", "r_intf"))
        connect_list.append(neighbor)
    return connect_list
```

### Особенности работы с Union

Пример кода, в котором в значении словаря типы указаны как Union[str, int, bool] (полный пример в файле example_14_dict_multiple_types_wrong.py):

```
def send_show_command_to_devices(
    devices: List[Dict[str, Union[str, int, bool]]], command: str
) -> Dict[str, str]:
    data = {}
    for device in devices:
        output = send_show_command(device, command)
        data[device["host"]] = output
    return data
```

В этом случае возникнет такая ошибка:

```
$ mypy example_14_dict_multiple_types.py
example_14_dict_multiple_types_wrong.py:24: error: Incompatible return value type (got
↪"Dict[Union[str, int, bool], str]", expected "Dict[str, str]")
Found 1 error in 1 file (checked 1 source file)
```

Проблема связана с тем, что если в значении словаря указан `Union[str, int, bool]`, то mypy это воспринимает как то, что любое значение может быть любым из этих типов. Указав что результатом будет словарь `Dict[str, str]`. Мы как бы уточняем, что `device["host"]` соответствует именно строка, но при работе с Union это будет ошибкой. Исправить ошибку можно либо указав, что возвращаемый словарь будет содержать в ключе `Union[str, int, bool]`, или указав в словаре в devices тип значения Any (полный пример в example_14_dict_multiple_types.py):

```
def send_show_command_to_devices(
    devices: List[Dict[str, Any]], command: str
) -> Dict[str, str]:
    data = {}
    for device in devices:
        output = send_show_command(device, command)
        data[device["host"]] = output
    return data
```

## Дополнительные материалы

- Шпаргалка по аннотации типов
- Модуль typing
- Type hinting in PyCharm
- PEP 563 – Postponed Evaluation of Annotations
- Annotations Best Practices

Статьи:

- the state of type hints in Python
- Введение в аннотации типов Python
- Python Type Checking (Guide)

Видео:

- Bernat Gabor - Type hinting (and mypy) - PyCon 2019
- Carl Meyer - Type-checked Python in the real world - PyCon 2018
- Michael Sullivan - Getting to Three Million Lines of Type-Annotated Python - PyCon 2019

Связанные проекты:

- pydantic
- typeshed
- MonkeyType

# 3. Code formatters

## Автоматическое форматирование кода с Black

Black - модуль для автоматического форматирования кода Python.

Установка

` pip install black `

Использование:

` black somefile_or_dir `

### Правила

```
# in:

j = [1,
     2,
     3
]

# out:

j = [1, 2, 3]
```

```
# in:

ImportantClass.important_method(exc, limit, lookup_lines, capture_locals, extra_argument)

# out:

ImportantClass.important_method(
    exc, limit, lookup_lines, capture_locals, extra_argument
)
```

### Волшебная запятая

### Исключение кода из форматирования

## Дополнительные материалы

Документация:

- Black

Линтеры и другие утилиты для анализа кода

- [pycodestyle](https://github.com/PyCQA/pycodestyle)
- [pylint](https://github.com/PyCQA/pylint)
- [isort - сортировка строк import](https://github.com/PyCQA/isort)
- [Pyflakes](https://github.com/PyCQA/pyflakes)
- [Flake8](https://github.com/PyCQA/flake8) - объединяет pycodestyle, pyflakes, mccabe

Модули для автоматического форматирования кода:

- [black](https://github.com/psf/black)
- [blue](https://github.com/grantjenks/blue)
- [yapf](https://github.com/google/yapf)
- [autopep8](https://github.com/hhatto/autopep8)

# 4. Модуль click

Click это модуль, который позволяет создавать интерфейс командной строки. Примеры того, что позволяет делать модуль:

- создавать аргументы и опции, с которыми может вызываться скрипт
- указывать типы аргументов, значения по умолчанию
- отображать сообщения с подсказками по использованию скрипта

Click не единственный модуль для обработки аргументов командной строки.

## Основы click

В click описание интерфейса командной строки (CLI) построено на декораторах:

- аргументы создаются с помощью декоратора click.argument
- опции с помощью декоратора click.option

Пример скрипта, который пингует только один IP-адрес (ping_ip.py):

```
import subprocess


def ping_ip(ip_address, count):
    """
    Ping IP address and return True/False
    """
    reply = subprocess.run(
        f"ping -c {count} -n {ip_address}",
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
    )
    if reply.returncode == 0:
        return True
    else:
        return False


if __name__ == "__main__":
    ip = "8.8.8.8"
    if ping_ip(ip, count=3):
        print(f"IP-адрес {ip:15} пингуется")
    else:
        print(f"IP-адрес {ip:15} не пингуется")
```

Первое, что нужно сделать, чтобы добавить CLI к этому скрипту - перенести код из блока `if __name__ == "__main__"` в функцию, так как click применяет декораторы к функции:

```
def main():
    ip = "8.8.8.8"
    if ping_ip(ip, count=3):
        print(f"IP-адрес {ip:15} пингуется")
    else:
        print(f"IP-адрес {ip:15} не пингуется")


if __name__ == "__main__":
    main()
```

Следующий шаг - превратить функцию main в команду click. Для этого надо применить декоратор click.command и импортировать click:

```
import click

@click.command()
def main():
    ip = "8.8.8.8"
    if ping_ip(ip, count=3):
        print(f"IP-адрес {ip:15} пингуется")
    else:
        print(f"IP-адрес {ip:15} не пингуется")


if __name__ == "__main__":
    main()
```

Теперь вызов скрипта отработает так же, но у скрипта появилась опция –help:

```
$ python ping_ip_click.py --help
Usage: ping_ip_click.py [OPTIONS]

Options:
  --help  Show this message and exit.
```

Так как для выполнения скрипта надо указать IP-адрес, надо добавить соответствующий параметр в CLI. Без IP-адреса скрипт запускать нет смысла, поэтому IP-адрес будут указываться с помощью обязательного параметра - аргумента. Он также указывается декоратором:

```
@click.command()
@click.argument("ip_address")
def main(ip_address):
    if ping_ip(ip_address, count=3):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        print(f"IP-адрес {ip_address:15} пингуется")
    else:
        print(f"IP-адрес {ip_address:15} не пингуется")


if __name__ == "__main__":
    main()
```

Строка `@click.argument("ip_address")` указывает, что теперь скрипт ожидает один обязательный параметр - ip_address, а также функция main должна принимать аргумент с таким именем, так как click автоматически передаст значение, которое передается при вызове скрипта, как ключевой аргумент функции, используя имя аргумента.

Теперь опция –help отображает такой вывод:

```
$ python ping_ip_click.py --help
Usage: ping_ip_click.py [OPTIONS] IP_ADDRESS

Options:
  --help  Show this message and exit.
```

И при вызове скрипта обязательно надо передать IP-адрес:

```
$ python ping_ip_click.py
Usage: ping_ip_click.py [OPTIONS] IP_ADDRESS
Try "ping_ip_click.py --help" for help.

Error: Missing argument "IP_ADDRESS".


$ python ping_ip_click.py 8.8.8.8
IP-адрес 8.8.8.8         пингуется
```

Так как функция зависит от еще одного значения - count, надо добавить еще один параметр click, в этот раз - опцию. Опции создаются с помощью декоратора click.option:

```
@click.command()
@click.argument("ip_address")
@click.option("--count", "-c", default=3)
def main(ip_address, count):
    if ping_ip(ip_address, count):
        print(f"IP-адрес {ip_address:15} пингуется")
    else:
        print(f"IP-адрес {ip_address:15} не пингуется")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
if __name__ == "__main__":
    main()
```

Так же как с аргументом, click будет передавать как ключевой аргумент имя опции и значение, которое было указано при вызове скрипта. Так как в данном случае у опции есть значение по умолчанию, если опция не указана передается значение 3. Еще одно следствие задания значения по умолчанию - click теперь считает, что count обязательно должен быть числом. Это поведение можно менять, указав тип параметра явно, но в данном случае, он подходит.

Запуск скрипта с вводом данных неправильного типа:

```
$ python ping_ip_click.py 8.8.8.8
IP-адрес 8.8.8.8         пингуется


$ python ping_ip_click.py 8.8.8.8 -c a
Usage: ping_ip_click.py [OPTIONS] IP_ADDRESS
Try "ping_ip_click.py --help" for help.

Error: Invalid value for "--count" / "-c": a is not a valid integer


$ python ping_ip_click.py 8.8.8.8 -c 1
IP-адрес 8.8.8.8         пингуется
```

И help для текущей версии скрипта:

```
$ python ping_ip_click.py --help
Usage: ping_ip_click.py [OPTIONS] IP_ADDRESS

Options:
  -c, --count INTEGER
  --help               Show this message and exit.
```

По умолчанию click не отображает значение, которое указано в default. Если необходимо это изменить, надо добавить в настройку опции `show_default=True`:

```
$ python ping_ip_click.py --help
Usage: ping_ip_click.py [OPTIONS] IP_ADDRESS

Options:
  -c, --count INTEGER  [default: 3]
  --help               Show this message and exit.
```

### Более практичный пример

Предыдущий пример использовался для демонстрации базовых настроек click и на практике не очень полезен. Чтобы сделать скрипт более интересным, можно добавить возможность отправлять ICMP-запросы на несколько IP-адресов и выводить на стандартный поток вывода информацию о том какие адреса отвечают, а какие нет.

Пример скрипта без использования click:

```
import subprocess


def ping_ip(ip_address, count):
    """
    Ping IP address and return True/False
    """
    reply = subprocess.run(
        f"ping -c {count} -n {ip_address}",
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
    )
    if reply.returncode == 0:
        return True
    else:
        return False


if __name__ == "__main__":
    ip_list = ["8.8.8.8", "8.8.4.4", "10.1.1.1", "192.168.100.1"]
    for ip in ip_list:
        if ping_ip(ip, count=3):
            print(f"IP-адрес {ip:15} пингуется")
        else:
            print(f"IP-адрес {ip:15} не пингуется")
```

Пример выполнения скрипта

```
$ python ping_ip_list.py
IP-адрес 8.8.8.8         пингуется
IP-адрес 8.8.4.4         пингуется
IP-адрес 10.1.1.1        не пингуется
IP-адрес 192.168.100.1   пингуется
```

Этот скрипт отличается от предыдущего тем, что теперь аргументу передается не один IP-адрес, а несколько. Click поддерживает такую возможность с помощью указания nargs в настройках аргумента. Так как в данном случае количество IP-адресов точно не известно, надо сделать так чтобы аргумент мог принимать любое количество. Для этого надо

указать nargs=-1 и, так как надо передать хотя бы один адрес, дополнительно указать required=True:

```
@click.command()
@click.argument("ip_address", nargs=-1, required=True)
@click.option("--count", "-c", default=3)
def main(ip_address, count):
    for ip in ip_address:
        if ping_ip(ip, count=3):
            print(f"IP-адрес {ip:15} пингуется")
        else:
            print(f"IP-адрес {ip:15} не пингуется")


if __name__ == "__main__":
    main()
```

Опция –help выглядит так:

```
$ python ping_ip_list_click.py --help
Usage: ping_ip_list_click.py [OPTIONS] IP_ADDRESS...

Options:
  -c, --count INTEGER
  --help               Show this message and exit.
```

И вызывать скрипт теперь можно таким образом:

```
$ python ping_ip_list_click.py 8.8.8.8 10.1.1.1 8.8.4.4 192.168.100.1
IP-адрес 8.8.8.8         пингуется
IP-адрес 10.1.1.1        не пингуется
IP-адрес 8.8.4.4         пингуется
IP-адрес 192.168.100.1   пингуется

$ python ping_ip_list_click.py 8.8.8.8 10.1.1.1 8.8.4.4 192.168.100.1 -c 2
IP-адрес 8.8.8.8         пингуется
IP-адрес 10.1.1.1        не пингуется
IP-адрес 8.8.4.4         пингуется
IP-адрес 192.168.100.1   пингуется
```

Перечисленные IP-адреса попадают в функцию в виде кортежа со строками.

## Установка скрипта через setuptools

**См.также:**

Интеграция click с setuptools

Скрипты с интерфейсом командной строки можно запускать с указанием shebang и сделав скрипт исполняемым или использовать setuptools.

Setuptools - это набор утилит для работы с пакетами Python. В нем есть масса возможностей, тут рассматривается лишь базовые пример. Подробнее в документации

Преимущества использования setuptools:

- запуск будет работать на Linux/Windows одинаково
- setuptools умеет работать с Python package

### Пример базового использования setuptools

Пример использования setuptools для скрипта example_05_pomodoro_timer.py.

Скрипт example_05_pomodoro_timer.py (вывод сокращен):

```
from datetime import datetime, date, timedelta
import time
import sys

import click

# ...

@click.command()
@click.option("--pomodoros_to_run", "-r", default=5, show_default=True, type=int)
@click.option("--work_minutes", "-w", default=25, show_default=True, type=int)
@click.option("--short_break", "-s", default=5, show_default=True, type=int)
@click.option("--long_break", "-l", default=30, show_default=True, type=int)
@click.option("--set_size", "-p", default=4, show_default=True, type=int)
def cli(pomodoros_to_run, work_minutes, short_break, long_break, set_size):
    session_stats = {"total": pomodoros_to_run, "done": 0, "todo": pomodoros_to_run}
    global stats
    stats = update_session_stats(session_stats)

    click.clear()
    all_pomodoros = list(range(1, pomodoros_to_run + 1))
    pomodoro_sets = sets_of_pomodoros(all_pomodoros, set_size)
    for pomo_set in pomodoro_sets:
        run_pomodoro_set(pomo_set, work_minutes, short_break, long_break)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
if __name__ == "__main__":
    cli()
```

Файл setup.py

```
from setuptools import setup

setup(
    name='pomodoro',
    version='1.0',
    py_modules=['example_05_pomodoro_timer'],
    install_requires=[
        'Click',
    ],
    entry_points='''
        [console_scripts]
        pomodoro=example_05_pomodoro_timer:cli
    ''',
)
```

Это очень базовый пример файла setup.py, но для своих скриптов его может быть вполне достаточно. Параметры setup:

- `name='pomodoro'` - это имя приложения/скрипта
- `version='1.0'` - версия скрипта
- `py_modules=['example_05_pomodoro_timer']` - основной скрипт, который надо запускать
- `install_requires=['Click']` - зависимости скрипта

И последний параметр `entry_points`:

```
entry_points='''
    [console_scripts]
    pomodoro=example_05_pomodoro_timer:cli
'''
```

- первое слово pomodoro - это команда, которая вызывает скрипт,
- example_05_pomodoro_timer - какой скрипт запустить (то же самое, что в py_modules)
- cli - в нашем случае это имя функции к которой привязана настройка click

Установка скрипта после создания setup.py (тут есть много вариантов, в часности запуск установки в development режиме -e):

```
$ pip install .
Processing /home/vagrant/repos/advanced-pyneng-2/advpyneng-online-2-sep-nov-2020/examples/
↪03_click
Requirement already satisfied: Click in /home/vagrant/venv/pyneng-py3-8-0/lib/python3.8/
↪site-packages (from pomodoro==1.0) (7.1.2)
Building wheels for collected packages: pomodoro
  Building wheel for pomodoro (setup.py) ... done
  Created wheel for pomodoro: filename=pomodoro-1.0-py3-none-any.whl size=2416␣
↪sha256=15a4751c3e03ab7da4f5e2ee979f54fadcc4a1724ce8f6994ee5966c3c8af193
  Stored in directory: /tmp/pip-ephem-wheel-cache-n_s8xn0d/wheels/b3/ff/4f/
↪e229093f911f6ad2ba16bbb55fbc28dd4cf701f7fe60d9333a
Successfully built pomodoro
Installing collected packages: pomodoro
Successfully installed pomodoro-1.0
```

После этого скрипт можно вызывать по pomodoro в любом месте:

```
[~/repos/advanced-pyneng-2/advpyneng-online-2-sep-nov-2020]
$ pomodoro
It's time to work!
Pomodoro 1
Work: 0:00:01^C
Aborted!
```

Удалить скрипт:

```
$ pip uninstall pomodoro
Found existing installation: pomodoro 1.0
Uninstalling pomodoro-1.0:
  Would remove:
    /home/vagrant/venv/pyneng-py3-8-0/bin/pomodoro
    /home/vagrant/venv/pyneng-py3-8-0/lib/python3.8/site-packages/example_05_pomodoro_
↪timer.py
    /home/vagrant/venv/pyneng-py3-8-0/lib/python3.8/site-packages/pomodoro-1.0.dist-info/*
Proceed (y/n)? y
  Successfully uninstalled pomodoro-1.0
```

## Параметры

Click поддерживает два вида параметров: опции и аргументы. В click аргументы имеют больше ограничений.

Возможности доступные только в опциях:

- запрос ввода значения опции у пользователя
- опции могут использоваться как флаги

- значения опций можно считывать из переменных окружения с автоматическим префиксом
- опциям можно писать help

Возможности доступные только в аргументах:

- передача любого количества значений

### Типы параметров

По умолчанию тип параметра будет строкой str, но его можно задавать явно или получать косвенно с помощью значения по умолчанию.

Доступные типы (большинство типов показаны в примерах опций):

- базовые типы: str, int, float, bool
- click.File - специальный тип, который автоматически открывает и закрывает файл. Возвращает открытый файл
- click.Path - тип для проверки пути, файл это или каталог и подобного. Возвращает строку, не открытый файл
- click.Choice - набор допустимых значений
- click.IntRange - диапазон числовых значений
- click.DateTime - преобразует строку с датой в объект datetime

---

**Примечание:** Также можно создавать свои типы данных

---

### Базовые типы: str, int, float, bool

Если указать, что тип параметра int или float, click будет проверять, что скрипту как аргумент передается именно этот тип данных. Например:

```
@click.command()
@click.argument("ip_address")
@click.option("--count", "-c", type=int, help="Number of packets")
def main(ip_address, count):
    pass
```

При вызове скрипта, опция `-c` ожидает значение типа int:

```
$ python example_03_ping_ip_list_progress_bar.py 8.8.8.8 -c test
Usage: example_03_ping_ip_list_progress_bar.py [OPTIONS] IP_ADDRESS
Try 'example_03_ping_ip_list_progress_bar.py --help' for help.

Error: Invalid value for '--count' / '-c': test is not a valid integer
```

### click.File

Тип click.File

```
class click.File(mode='r', encoding=None, errors='strict', lazy=None, atomic=False)
```

Этот тип используется для работы с файлами. Особенность типа clicl.File в том, что файл автоматически открывается и закрывается click. Файл может быть открыт для чтения или записи, плюс специальное значение - указывает, что вместо файла надо открыть stdin/stdout.

Пример аргумента с типом click.File:

```
@click.command()
@click.argument("connection_params", type=click.File("r"))
def cli(connection_params):
    devices = yaml.safe_load(connection_params)
```

Так как click открывает файл, внутри функции cli connection_params это уже открытый файл.

### click.Path

Тип click.Path

```
class click.Path(exists=False, file_okay=True, dir_okay=True, writable=False,␣
↪readable=True, resolve_path=False, allow_dash=False, path_type=None)
```

Пример аргумента с типом click.Path:

```
@click.command()
@click.argument("source", type=click.Path(exists=True))
def cli(source):
    pass
```

```
$ python script.py sh_cdp_n_r22.txt
Usage: script.py SOURCE DESTINATION
Try 'script.py --help' for help.

Error: Invalid value for 'SOURCE': Path 'sh_cdp_n_r22.txt' does not exist.
```

### click.Choice

Тип click.Choice позволяет указать допустимые варианты значений для параметра:

```
@click.command()
@click.option("--key", "-k", type=click.Choice(["mac", "ip", "vlan"]))
def cli(key):
    pass
```

Если при вызове скрипта передать другое значение, возникнет ошибка:

```
$ python example_choice.py --key test
...
Error: Invalid value for '--key' / '-k': invalid choice: test. (choose from mac, ip, vlan)
```

### click.IntRange

Тип click.IntRange:

```
class click.IntRange(min=None, max=None, clamp=False)
```

Пример использования:

```
@click.command()
@click.option("--threads", "-t", type=click.IntRange(1, 10))
def cli(threads):
    pass
```

Если указанное значение не попадает в диапазон, по умолчанию возникнет ошибка. Если установить `clamp=True`, вместо ошибки, значение будет подгоняться под ближайшую границу. Например, если в примере выше передать 20, то threads будет равен 10.

## Аргументы

Аргументы создаются с помощью декоратора `@click.argument`:

```
@click.argument(name, type=None, required=True, default=None, nargs=None)
```

В самом простом случае, достаточно указать только имя аргумента:

```
@click.command()
@click.argument("name")
def cli(name):
    """Print NAME"""
    print(name)
```

Так будет выглядеть help скрипта:

```
$ python basics_01.py --help
Usage: basics_01.py [OPTIONS] NAME

  Print NAME

Options:
  --help  Show this message and exit.
```

И так вызов:

```
$ python basics_01.py R2D2
R2D2
```

### Переменное количество аргументов

Параметр nargs позволяет контролировать какое количество аргументов можно передать. По умолчанию, значение 1. Значение -1 - это специальное значение обозначающее, что аргументов может быть сколько угодно.

Пример использования nargs

```
import subprocess
import click


def ping_ip(ip_address):
    reply = subprocess.run(
        f"ping -c 2 -n {ip_address}",
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        encoding="utf-8",
    )
    if reply.returncode == 0:
        return True
    else:
        return False


@click.command()
@click.argument("ip_addresses", nargs=-1, required=True)
def cli(ip_addresses):
    """
    Ping IP_ADDRESSES
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    """
    for ip in ip_addresses:
        if ping_ip(ip):
            print(f"IP-адрес {ip:15} пингуется")
        else:
            print(f"IP-адрес {ip:15} не пингуется")


if __name__ == "__main__":
    cli()
```

## Опции

Класс click.Option:

```
class click.Option(
    param_decls=None,
    show_default=False,
    prompt=False,
    confirmation_prompt=False,
    hide_input=False,
    is_flag=None,
    flag_value=None,
    multiple=False,
    count=False,
    allow_from_autoenv=True,
    type=None,
    help=None,
    hidden=False,
    show_choices=True,
    show_envvar=False,
    **attrs
)
```

Имена опций:

- `@click.option("-f", "--foo-bar")`, имя параметра функции будет `"foo_bar"`
- `@click.option("-x")`, имя `"x"`
- `@click.option("-f", "--filename", "dest")`, имя `"dest"`
- `@click.option("--CamelCase")`, имя `"camelcase"`
- `@click.option("-f", "-fb")`, имя `"f"`
- `@click.option("--f", "--foo-bar")`, имя `"f"`

- `@click.option("---f")`, имя `"_f"`

### Запрос значения у пользователя

Запрос выполняется только если в опции не указано значение:

```
@click.command()
@click.option("--username", "-u", prompt=True)
@click.option("--password", "-p", prompt=True, hide_input=True)
@click.option("--secret", "-s", prompt=True, hide_input=True)
def cli(username, password, secret):
    pass
```

Параметр hide_input позволяет скрывать вводимое значение.

### Переменные окружения

```
@click.command()
@click.option("--username", "-u", envvar="NET_USER")
@click.option("--password", "-p", envvar="NET_PASSWORD")
@click.option("--secret", "-s", envvar="NET_SECRET")
def cli(username, password, secret):
    pass
```

Переменные окружения и prompt - сначала проверяется переменная окружения, потом, если ее нет, запрос у пользователя:

```
@click.command()
@click.option("--username", "-u", envvar="NET_USER", prompt=True)
@click.option("--password", "-p", envvar="NET_PASSWORD", prompt=True, hide_input=True)
@click.option("--secret", "-s", envvar="NET_SECRET", prompt=True, hide_input=True)
def cli(username, password, secret):
    pass
```

### Флаг

```
@click.option("--show-all", "-a", is_flag=True, help="show db content")
```

### Подтверждение ввода

confirmation_prompt может пригодится при запросе пароля или других критичных данных. В этом случае пароль запрашивается повторно автоматически и два введенных значения сравниваются:

```
@click.command()
@click.option("--username", "-u", prompt=True)
@click.option("--password", "-p", prompt=True, hide_input=True, confirmation_prompt=True)
def cli(username, password):
    print(username, password)
```

---

**Примечание:** Так как это распространенная задача, ввод пароля таким образом можно заменить декоратором click.password_option.

---

## Дополнительные возможности

- `click.password_option(*param_decls, **attrs)`
- `click.confirmation_option(*param_decls, **attrs)`
- `click.version_option(version=None, *param_decls, **attrs)`
- click.echo
- click.style
- click.secho
- click.clear
- click.pause
- click.prompt
- click.confirm
- click.echo_via_pager
- click.progressbar

### Декоратор click.password_option

Декоратор click.password_option равнозначен такой опции:

```
@click.option("--password", "-p", prompt=True, hide_input=True, confirmation_prompt=True)
```

Пример использования:

```
@click.command()
@click.option("--username", "-u", prompt=True)
@click.password_option()
def cli(username, password):
    print(username, password)
```

### click.echo, click.style, click.secho

Функция click.echo в целом работает как print, но при этом:

- добавляет обработку цветовых кодов ANSI в Windows
- автоматически скрывает коды ANSI, если вывод идет не в терминал

click.style добавляет возможность выводить текст в цвете. Использовать можно или так

```
click.echo(click.style('Hello World!', fg='green'))
```

Или использовать короткий вариант click.secho:

```
click.secho('Hello World!', fg='green')
```

### click.progressbar

```
def ping_ip_addresses(ip_addresses, count):
    reachable = []
    unreachable = []
    with click.progressbar(ip_addresses, label="Пингую адреса") as bar:
        for ip in bar:
            if ping_ip(ip, count):
                reachable.append(ip)
            else:
                unreachable.append(ip)
    return reachable, unreachable
```

Вывод:

```
$ python example_03_ping_ip_list_progress_bar.py 8.8.8.8 8.8.4.4 10.1.1.1 192.168.100.1
Пингую адреса  [####################################]  100%
IP-адрес 8.8.8.8         пингуется
IP-адрес 8.8.4.4         пингуется
IP-адрес 192.168.100.1   пингуется
IP-адрес 10.1.1.1        не пингуется
```

```
def send_command_to_devices(devices, command, limit):
    results = []
    with ThreadPoolExecutor(max_workers=limit) as executor:
        futures = [
            executor.submit(send_show_command, device, command) for device in devices
        ]
        with click.progressbar(
            length=len(futures), label="Connecting to devices"
        ) as bar:
            for future in as_completed(futures):
                results.append(future.result())
                bar.update(1)
    return results
```

### click.clear

Функция click.clear очищает экран. Удобно выполнять в начале работы скрипта.

```
def cli(pomodoros_to_run, work_minutes, short_break, long_break, set_size):
    click.clear()
    all_pomodoros = list(range(1, pomodoros_to_run + 1))
    pomodoro_sets = sets_of_pomodoros(all_pomodoros, set_size)
    for pomo_set in pomodoro_sets:
        run_pomodoro_set(pomo_set, work_minutes, short_break, long_break)
```

### click.pause

Функция click.pause останавливает выполнение скрипта, выводит сообщение «Press any key to continue ...» и ждет нажатия любой клавиши. Использовать можно в любом месте, таким образом

```
click.pause()
```

## Большие приложения

Для более сложных приложений, интерфейс командной строки, как правило, тоже усложняется. С помощью Click можно создавать сложные интерфейсы командной строки, но для этого надо разобраться с понятием контекста.

Контекст (Context) это внутренний объект Click, который создается при выполнении команды click и содержит информацию о том какая команда была вызвана, с какими параметрами. В простых случаях с контекстом не нужно работать напрямую, но можно посмотреть на него, если добавить декоратор `click.pass_context` к функции:

```
import click


@click.command()
@click.argument("ip_address")
@click.option("--count", "-c", default=2, type=int, help="Number of packets")
@click.pass_context
def ping_ip(ctx, ip_address, count):
    """
    Ping IP address and return True/False
    """
    print(ctx.command)
    print(ctx.params)


if __name__ == "__main__":
    ping_ip()
```

Вывод будет таким:

```
$ python example_01_ping_function.py 8.8.8.8
<Command ping-ip>
{'ip_address': '8.8.8.8', 'count': 2}
```

В выводе видно какая команда была вызвана - ping-ip и какие параметры были указаны: тут и значение адреса 8.8.8.8 и значение опции по умолчанию - 2.

Кроме того, что контекст содержит много информации для самого click, ему можно присваивать произвольные значение в атрибут obj. Этот атрибут используется для передачи информации между группой команд и командами. Когда значений несколько, как правило, используется словарь.

### click.group

click.group это декоратор, который работает так же как click.command, но при этом позволяет создавать подкоманды. Пример интерфейса с подкомандами - git. В зависимости от того какая команда пишется после git, появляются разные аргументы и опции.

Пример интерфейса с группой:

```
import click


@click.group()
def pomodoro_cli():
    pass


@pomodoro_cli.command()
@click.option("--day", "-d", is_flag=True)
@click.option("--week", "-w", is_flag=True)
@click.option("--month", "-m", is_flag=True)
def stats(day, week, month):
    print("STATS")


@pomodoro_cli.command()
@click.option("--pomodoros_to_run", "-r", default=5, show_default=True, type=int)
@click.option("--work_minutes", "-w", default=25, show_default=True, type=int)
@click.option("--short_break", "-s", default=5, show_default=True, type=int)
@click.option("--long_break", "-l", default=30, show_default=True, type=int)
@click.option("--set_size", "-p", default=4, show_default=True, type=int)
def work(pomodoros_to_run, work_minutes, short_break, long_break, set_size):
    pass


if __name__ == "__main__":
    pomodoro_cli()
```

Первой надо создать функцию, которая является группой, так как затем имя функции используется в декораторах других функций, вместо click:

```
@click.group()
def pomodoro_cli():
    pass
```

Следующие две функции создают подкоманды скрипта stats и work (как commit и add в git). Имена функций будут именами команд. У этих функций могут быть свои аргументы и опции. Единственное отличие от предыдущих примеров - это то, что вместо декоратора `click.`

command используется pomodoro_cli.command. Таким образом указывается, что команда относится к группе pomodoro_cli:

```
@pomodoro_cli.command()
@click.option("--day", "-d", is_flag=True)
@click.option("--week", "-w", is_flag=True)
@click.option("--month", "-m", is_flag=True)
def stats(day, week, month):
    print("STATS")


@pomodoro_cli.command()
@click.option("--pomodoros_to_run", "-r", default=5, show_default=True, type=int)
@click.option("--work_minutes", "-w", default=25, show_default=True, type=int)
@click.option("--short_break", "-s", default=5, show_default=True, type=int)
@click.option("--long_break", "-l", default=30, show_default=True, type=int)
@click.option("--set_size", "-p", default=4, show_default=True, type=int)
def work(pomodoros_to_run, work_minutes, short_break, long_break, set_size):
    pass
```

При такой настройке help скрипта выглядит таким образом:

```
$ python example_10_click_group_basics.py --help
Usage: example_10_click_group_basics.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  stats
  work
```

У каждой команды есть свой help:

```
$ python example_10_click_group_basics.py stats --help
Usage: example_10_click_group_basics.py stats [OPTIONS]

Options:
  -d, --day
  -w, --week
  -m, --month
  --help       Show this message and exit.


$ python example_10_click_group_basics.py work --help
Usage: example_10_click_group_basics.py work [OPTIONS]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Options:
  -r, --pomodoros_to_run INTEGER  [default: 5]
  -w, --work_minutes INTEGER      [default: 25]
  -s, --short_break INTEGER       [default: 5]
  -l, --long_break INTEGER        [default: 30]
  -p, --set_size INTEGER          [default: 4]
  --help                          Show this message and exit.
```

В этом случае у каждой команды свои параметры, плюс команд мало.

### click.pass_context

Часто бывают случаи, когда часть параметров команд пересекаются и, особенно если команд много, их становится неудобно описывать, так как параметры приходится повторять. В этом случае общие параметры можно перенести в функцию-группу, но появляется новая проблема - как теперь передать значение этих параметров в подкоманды? Тут на помощь приходит контекст.

Пример скрипта (example_11_click_context.py):

```
import click


@click.group()
@click.option("--db-filename", "-n", help="db filename")
@click.pass_context
def dhcp_db(context, db_filename):
    context.obj = {"db_filename": db_filename}


@dhcp_db.command()
@click.option("--db-schema", "-s", help="db schema filename")
@click.pass_context
def create(context, db_schema):
    """
    create DB
    """


@dhcp_db.command()
@click.argument("filename", nargs=-1, required=True)
@click.option("--switch-data", "-s", default=False, is_flag=True)
@click.pass_context
def add(context, filename, switch_data):
    """
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    add data to db from FILENAME
    """


@dhcp_db.command()
@click.option("--key", "-k", type=click.Choice(["mac", "ip", "vlan"]))
@click.option("--value", "-v", help="value of key")
@click.option("--show-all", "-a", is_flag=True, help="show db content")
@click.pass_context
def get(context, key, value, show_all):
    """
    get data from db
    """


if __name__ == "__dhcp_db__":
    dhcp_db()
```

## Дополнительные материалы

Документация:

- Click
- Примеры click

Полезные статьи:

- Comparing Python Command-Line Parsing Libraries – Argparse, Docopt, and Click

Видео:

- Building Command Line Applications with Click
- Sebastian Vetter - Click: A Pleasure To Write, A Pleasure To Use - PyCon 2016

Другие модули для создания cli:

- argparse, Argparse Tutorial
- typer
- docopt
- Python Fire

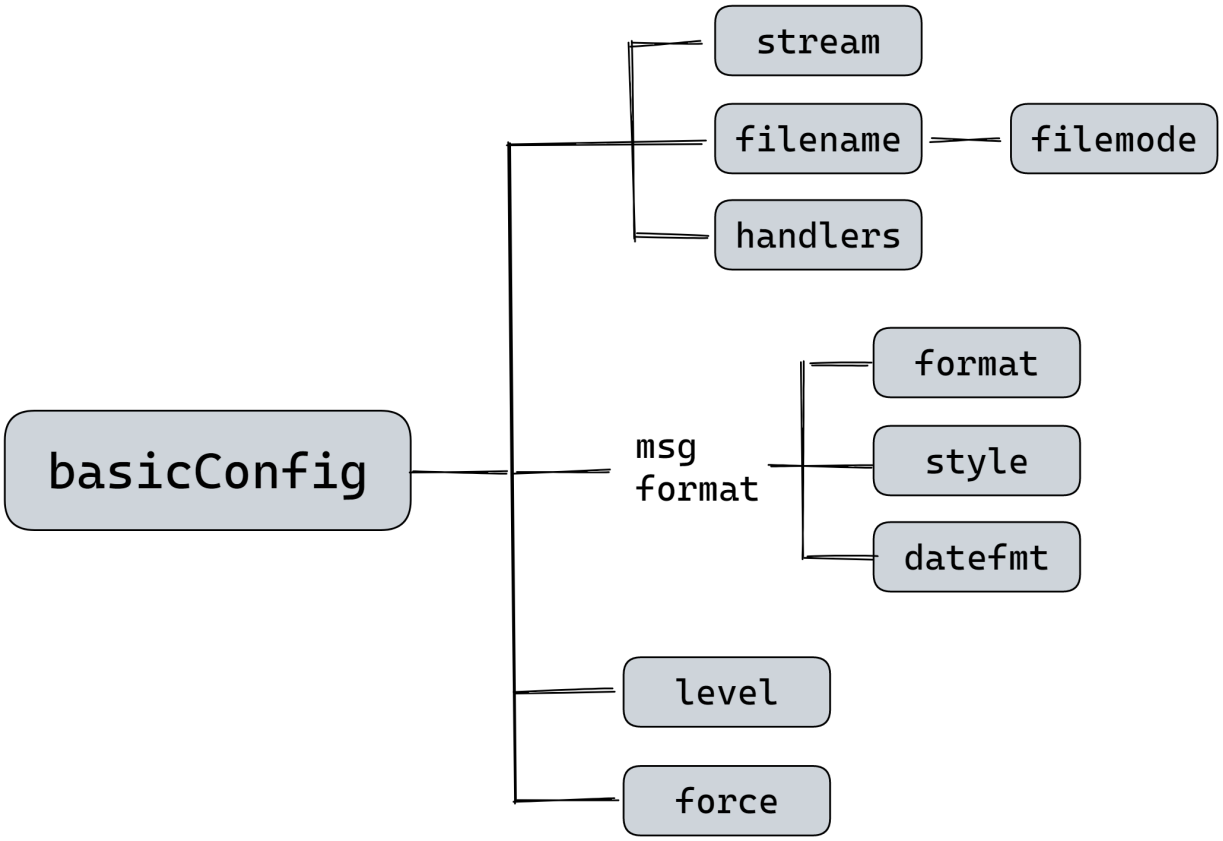
setuptools:

- Click Setuptools Integration
- How to package and deploy CLI applications with Python PyPA setuptools build

# 5. Модуль logging

| Уровень | Когда используется |
|---|---|
| DEBUG | Подробная информация для диагностики проблемы. |
| INFO | Подтверждение, что все работает как должно. |
| WARNING | Случилось что-то неожиданное, но программа все еще работает. Также может использоваться для индикации о будущих проблемах. |
| ERROR | Возникла ошибка и из-за нее не получилось выполнить часть задач. |
| CRITICAL | Серьезная ошибка из-за которой программа не может подолжить работу |

## Базовый пример




Вызов basicConfig должен происходить перед любыми вызовами logging.debug, logging.info и т.д.

basicConfig задуман как одноразовое простое средство настройки, только первый вызов что-то сделает: последующие вызовы фактически не выполняются.

logging_basic_1.py

```
import logging

logging.basicConfig(filename='mylog.log', level=logging.DEBUG)

logging.debug('Сообщение уровня debug')
logging.info('Сообщение уровня info')
logging.warning('Сообщение уровня warning')
```

Log-файл

```
DEBUG:root:Сообщение уровня debug
INFO:root:Сообщение уровня info
WARNING:root:Сообщение уровня warning
```

logging_basic_2.py

```
import logging

logging.basicConfig(filename='mylog2.log', level=logging.DEBUG)

logging.debug('Сообщение уровня debug:\n%s', str(globals()))
logging.info('Сообщение уровня info')
logging.warning('Сообщение уровня warning')
```

Log-файл

```
DEBUG:root:Сообщение уровня debug:
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_
↪importlib_external.SourceFileLoader object at 0xb72a57ac>, '__spec__': None, '__
↪annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__':
↪'logging_basic_2.py', '__cached__': None, 'logging': <module 'logging' from '/usr/local/
↪lib/python3.6/logging/__init__.py'>}
INFO:root:Сообщение уровня info
WARNING:root:Сообщение уровня warning
```

### Параметры logging.basicConfig

- filename Specifies that a FileHandler be created, using the specified filename, rather than a StreamHandler.
- filemode Specifies the mode to open the file, if filename is specified (if filemode is unspecified, it defaults to „a“).
- format
- datefmt - date/time format

- style - %, {, $
- level
- stream - stream to initialize the StreamHandler. Incompatible with filename - if both are present, a ValueError is raised.
- handlers - Incompatible with filename or stream

Пример вывода информации о потоках:

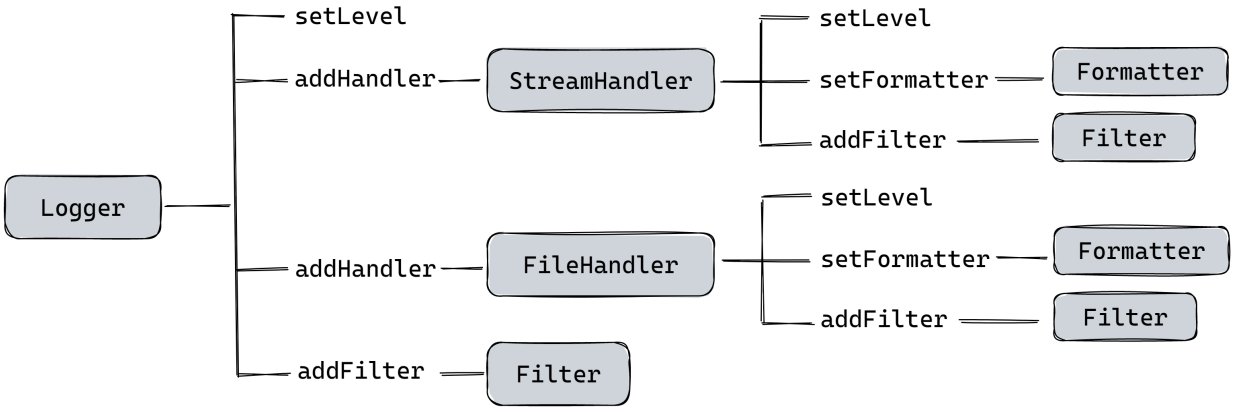
```
from concurrent.futures import ThreadPoolExecutor
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat
import logging
import yaml
from netmiko import ConnectHandler, NetMikoAuthenticationException


logging.getLogger('paramiko').setLevel(logging.WARNING)

logging.basicConfig(
    format='%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)


def send_show(device_dict, command):
    ip = device_dict['host']
    logging.info(f'===> {datetime.now().time()} Connection: {ip}')
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        logging.info(f'<=== {datetime.now().time()} Received:   {ip}')
    return result


def send_command_to_devices(devices, command):
    data = {}
    with ThreadPoolExecutor(max_workers=2) as executor:
        result = executor.map(send_show, devices, repeat(command))
        for device, output in zip(devices, result):
            data[device['host']] = output
    return data


if __name__ == '__main__':
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    pprint(send_command_to_devices(devices, 'sh ip int br'), width=120)
```

## Компоненты модуля logging




- Logger - это основной интерфейс для работы с модулем
- Handler - отправляет log-сообщения конкретному получателю
- Filter - позволяет фильтровать сообщения
- Formatter - указывает формат сообщения

Log события передаются между logger, handlers, filters и formatter в виде экземпляра LogRecord.

Вывод на стандартный поток ошибок logging_api_example_1.py

```
import logging

logger = logging.getLogger(__name__)

## messages
logger.debug('Сообщение уровня debug')
logger.info('Сообщение уровня info')
logger.warning('Сообщение уровня warning')
```

Результат выполнения

```
$ python logging_api_example_1.py
Сообщение уровня warning
```

logging_api_example_2.py

```
import logging

logger = logging.getLogger('My Script')
logger.setLevel(logging.DEBUG)

console = logging.StreamHandler()
console.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s',
                              datefmt='%H:%M:%S')
console.setFormatter(formatter)

logger.addHandler(console)

## messages
logger.debug('Сообщение уровня debug %s', 'SOS')
logger.info('Сообщение уровня info')
logger.warning('Сообщение уровня warning')
```

Результат выполнения

```
$ python logging_api_example_2.py
16:39:27 - My Script - DEBUG - Сообщение уровня debug: SOS
16:39:27 - My Script - INFO - Сообщение уровня info
16:39:27 - My Script - WARNING - Сообщение уровня warning
```

Вывод на стандартный поток вывода logging_api_example_2_stdout.py

```
import sys
import logging

logger = logging.getLogger('My Script')
logger.setLevel(logging.DEBUG)

console = logging.StreamHandler(sys.stdout)
console.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s',
                              datefmt='%H:%M:%S')
console.setFormatter(formatter)

logger.addHandler(console)

## messages
logger.debug('Сообщение уровня debug %s', 'SOS')
logger.info('Сообщение уровня info')
logger.warning('Сообщение уровня warning')
```

logging_api_example_2_new_format.py

```
import logging

logger = logging.getLogger('My Script')
logger.setLevel(logging.DEBUG)

console = logging.StreamHandler()
console.setLevel(logging.DEBUG)
formatter = logging.Formatter('{asctime} - {name} - {levelname} - {message}',
                              datefmt='%H:%M:%S', style='{')
console.setFormatter(formatter)

logger.addHandler(console)

## messages
logger.debug('Сообщение уровня debug: %s', 'SOS')
logger.info('Сообщение уровня info')
logger.warning('Сообщение уровня warning')
```

Результат выполнения

```
$ python logging_api_example_2.py
16:45:20 - My Script - DEBUG - Сообщение уровня debug: SOS
16:45:20 - My Script - INFO - Сообщение уровня info
16:45:20 - My Script - WARNING - Сообщение уровня warning
```

### Запись логов в файл

logging_api_example_3.py

```
import logging

logger = logging.getLogger('My Script')
logger.setLevel(logging.DEBUG)

logfile = logging.FileHandler('logfile.log')
logfile.setLevel(logging.WARNING)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s',
                              datefmt='%H:%M:%S')
logfile.setFormatter(formatter)

logger.addHandler(logfile)

## messages
logger.debug('Сообщение уровня debug')
logger.info('Сообщение уровня info')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
logger.warning('Сообщение уровня warning')
```

Результат выполнения. Файл logfile.log

```
17:58:34 - My Script - WARNING - Сообщение уровня warning
```

### Запись в файл и вывод на stderr

logging_api_example_4.py

```
import logging

logger = logging.getLogger('My Script')
logger.setLevel(logging.DEBUG)

### stderr
console = logging.StreamHandler()
console.setLevel(logging.DEBUG)
formatter = logging.Formatter('{asctime} - {name} - {levelname} - {message}',
                              datefmt='%H:%M:%S', style='{')
console.setFormatter(formatter)

logger.addHandler(console)

### File
logfile = logging.FileHandler('logfile3.log')
logfile.setLevel(logging.WARNING)
formatter = logging.Formatter('{asctime} - {name} - {levelname} - {message}',
                              datefmt='%H:%M:%S', style='{')
logfile.setFormatter(formatter)

logger.addHandler(logfile)

## messages
logger.debug('Сообщение уровня debug')
logger.info('Сообщение уровня info')
logger.warning('Сообщение уровня warning')
```

### Handlers

### RotatingFileHandler

logging_api_example_5_file_rotation.py

```
import logging
import logging.handlers

logger = logging.getLogger('My Script')
logger.setLevel(logging.DEBUG)

logfile = logging.handlers.RotatingFileHandler(
    'logfile_with_rotation.log', maxBytes=10, backupCount=3)
logfile.setLevel(logging.DEBUG)
formatter = logging.Formatter('{asctime} - {name} - {levelname} - {message}',
                              datefmt='%H:%M:%S', style='{')
logfile.setFormatter(formatter)

logger.addHandler(logfile)

## messages
logger.debug('Сообщение уровня debug')
logger.info('Сообщение уровня info')
logger.warning('Сообщение уровня warning')
```

Результат выполнения

```
$ ls -1 logfile_with_rotation*
logfile_with_rotation.log
logfile_with_rotation.log.1
logfile_with_rotation.log.2
logfile_with_rotation.log.3
logfile_with_rotation.log
```

logfile_with_rotation.log - это самый свежий файл, затем идет logfile_with_rotation.log.1, logfile_with_rotation.log.2 и тд.

### Logging tree

netmiko_func.py

```
import logging
from netmiko import ConnectHandler


logger = logging.getLogger('superscript.netfunc')
#logger = logging.getLogger('netfunc')

device_params = {
     'device_type': 'cisco_ios',
     'ip': '192.168.100.1',
     'username': 'cisco',
     'password': 'cisco',
     'secret': 'cisco'}


def send_show_command(device, command):
    with ConnectHandler(**device) as ssh:
        ssh.enable()
        output = ssh.send_command(command)
        logger.debug('Вывод команды:\n{}'.format(output))
    return output

if __name__ == '__main__':
    send_show_command(device_params, 'sh ip int br')
```

logging_api_example_6_mult_files.py

```
import logging
from netmiko_func import send_show_command, device_params

logger = logging.getLogger('superscript')
logger.setLevel(logging.DEBUG)

console = logging.StreamHandler()
console.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s',
                              datefmt='%H:%M:%S')
console.setFormatter(formatter)

logger.addHandler(console)

if __name__ == "__main__":
    logger.debug('Before function')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    send_show_command(device_params, 'sh ip int br')
    logger.debug('After function')
```

Результат выполнения

```
$ python logging_api_example_6_mult_files.py
19:16:44 - superscript - DEBUG - Before function
19:16:50 - superscript.netfunc - DEBUG - Вывод команды:
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
Ethernet0/2                190.16.200.1    YES NVRAM  up                    up
Ethernet0/3                192.168.230.1   YES NVRAM  administratively down down
Ethernet0/3.100            10.100.0.1      YES NVRAM  administratively down down
Ethernet0/3.200            10.200.0.1      YES NVRAM  administratively down down
Ethernet0/3.300            10.30.0.1       YES NVRAM  administratively down down
Loopback0                  10.1.1.2        YES manual up                    up
19:16:50 - superscript - DEBUG - After function
```

### logger.exception

logging_api_example_7_exception.py

```
import logging
from netmiko_func import send_show_command, device_params

logger = logging.getLogger('superscript')
logger.setLevel(logging.DEBUG)

console = logging.StreamHandler()
console.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s',
                              datefmt='%H:%M:%S')
console.setFormatter(formatter)

logger.addHandler(console)

logger.debug('Before exception')
try:
    2 + 'test'
except TypeError:
    logger.exception('Error')
logger.debug('After exception')
```

Результат выполнения

```
$ python logging_api_example_7_exception.py
19:23:24 - superscript - DEBUG - Before exception
19:23:24 - superscript - ERROR - Error
Traceback (most recent call last):
  File "logging_api_example_7_exception.py", line 17, in <module>
    2 + 'test'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
19:23:24 - superscript - DEBUG - After exception
```

### Конфигурация logging из словаря

logging_api_example_8.py

```
import logging

logger = logging.getLogger('superscript')
logger.setLevel(logging.DEBUG)

console = logging.StreamHandler()
console.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s',
                              datefmt='%H:%M:%S')
console.setFormatter(formatter)

logger.addHandler(console)

## messages
logger.debug('Сообщение уровня debug %s', 'SOS')
logger.info('Сообщение уровня info')
logger.warning('Сообщение уровня warning')
```

logging_api_example_8_yaml_cfg.py

```
import logging
import logging.config
import yaml

# create logger
logger = logging.getLogger('superscript')

#read config
with open('log_config.yml') as f:
    log_config = yaml.load(f)

logging.config.dictConfig(log_config)
```

(continues on next page)

(продолжение с предыдущей страницы)

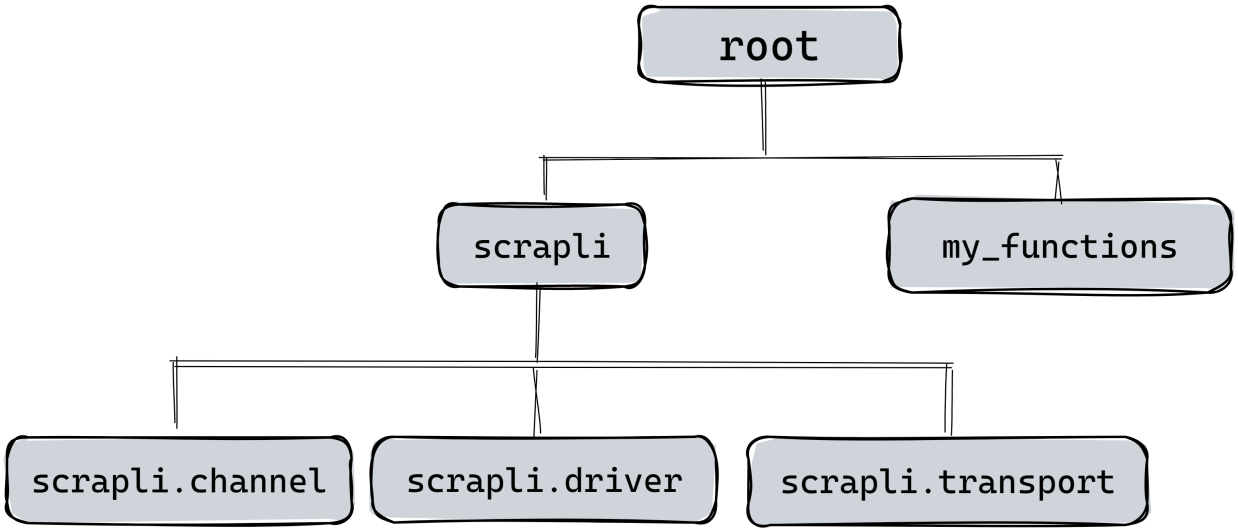
```
## messages
logger.debug('Сообщение уровня debug %s', 'SOS')
logger.info('Сообщение уровня info')
logger.warning('Сообщение уровня warning')
```

log_config.yml

```
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
loggers:
  superscript:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
  level: DEBUG
  handlers: [console]
```

```
$python logging_api_example_8_yaml_cfg.py
2018-02-17 19:50:56,266 - superscript - DEBUG - Сообщение уровня debug SOS
2018-02-17 19:50:56,266 - superscript - INFO - Сообщение уровня info
2018-02-17 19:50:56,266 - superscript - WARNING - Сообщение уровня warning
```

## Иерархия логеров




В модуле logging есть иерархия логеров. Самый главный в иерархии root. Остальные под ним, часто, на одном уровне. В одном модуле может быть много логеров с разной иерархией, например, в paramiko есть логер paramiko.transport, он по иерархии ниже paramiko.

Такой конфиг настраивает logger root:

```
logging.basicConfig(
    level=logging.INFO
)
```

Так как он самый главный, его настройка приводит к тому, что все остальные логеры тоже начинают отображать информацию. При такой настройке настраивается конкретный logger и никакие другие логи уже не сыпятся.

```
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

console = logging.StreamHandler()
console.setLevel(logging.DEBUG)
formatter = logging.Formatter(
    "%(threadName)s %(name)s %(levelname)s %(asctime)s: %(message)s", datefmt="%H:%M:%S"
)
console.setFormatter(formatter)
logger.addHandler(console)
```

Хотя тут тоже можно использовать root, для этого надо первую строку написать так

```
logger = logging.getLogger()
```

С базовым конфигом надо отключить или хотя бы приглушить на какой-то уровень существующие логеры других модулей:

```
logging.getLogger("paramiko").setLevel(logging.WARNING)

logging.basicConfig(
    level=logging.INFO
)
```

Со своим логером (не root) - надо наоборот добавить строк чтобы включить logger netmiko/paramiko. Для этого надо добавлять такие строки (учитывая что предыдущие все есть с handler, formatter и тп

```
netmiko_log = logging.getLogger("netmiko")
netmiko_log.setLevel(logging.DEBUG)
netmiko_log.addHandler(console)
```

## Rich Handler

Настройка с logging.basicConfig:

```
import logging
from rich.logging import RichHandler

FORMAT = "%(message)s"
logging.basicConfig(
    level="NOTSET", format=FORMAT, datefmt="[%X]", handlers=[RichHandler()]
)

log = logging.getLogger("rich")
log.info("Hello, World!")
```

Настройка с Handler

```
from concurrent.futures import ThreadPoolExecutor
from pprint import pprint
from itertools import repeat
import logging

import yaml
from scrapli import Scrapli
from scrapli.exceptions import ScrapliException
from rich.logging import RichHandler
```

(continues on next page)

(продолжение с предыдущей страницы)

```
logging.getLogger("scrapli").setLevel(logging.WARNING)
log = logging.getLogger(__name__)
log.setLevel(logging.DEBUG)

### stderr
console = RichHandler(level=logging.DEBUG)
formatter = logging.Formatter(
    "{name} - {message}", datefmt="%X", style="{"
)
console.setFormatter(formatter)
log.addHandler(console)

### File
logfile = logging.FileHandler("logfile3.log")
logfile.setLevel(logging.DEBUG)
formatter = logging.Formatter("{asctime} - {name} - {levelname} - {message}", style="{")
logfile.setFormatter(formatter)

log.addHandler(logfile)
```

## Фильтры

Фильтры можно применять к logger или к handler

---

**Примечание:** LogRecord attributes

---

```
class LevelFilter(logging.Filter):
    def __init__(self, level):
        self.level = level

    def filter(self, record):
        return record.levelno == self.level


log = logging.getLogger(__name__)
log.setLevel(logging.DEBUG)
log.addFilter(LevelFilter(logging.DEBUG))

logfile = logging.FileHandler("logfile3.log")
logfile.setLevel(logging.DEBUG)
formatter = logging.Formatter("{asctime} - {name} - {levelname} - {message}", style="{")
logfile.setFormatter(formatter)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
log.addHandler(logfile)
```

### Handler Filter

```
class LevelFilter(logging.Filter):
    def __init__(self, level):
        self.level = level

    def filter(self, record):
        return record.levelno == self.level


log = logging.getLogger(__name__)
log.setLevel(logging.DEBUG)

logfile = logging.FileHandler("logfile3.log")
logfile.setLevel(logging.DEBUG)
logfile.addFilter(LevelFilter(logging.DEBUG))
logfile.addFilter(MessageFilter("test"))

formatter = logging.Formatter("{asctime} - {name} - {levelname} - {message}", style="{")
logfile.setFormatter(formatter)

log.addHandler(logfile)
```

### Использование фильтра для добавления информации в запись

**Примечание:** Using Filters to impart contextual information

```
class AddIPFilter(logging.Filter):
    def filter(self, record):
        match = re.search(r"\d+\.\d+\.\d+\.\d+", record.msg)
        if match:
            record.ip = match.group()
        else:
            record.ip = None
        return True
```

## NullHandler

```
import logging


log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())


def send_show(device_dict, command):
    ip = device_dict["host"]
    log.info(f"===>  Connection: {ip}")

    try:
        with ConnectHandler(**device_dict) as ssh:
            ssh.enable()
            result = ssh.send_command(command)
            log.debug(f"<===  Received:   {ip}")
            log.debug(f"Получен вывод команды {command}\n\n{result}")
        return result
    except SSHException as error:
        log.error(f"Ошибка {error} на {ip}")
```

## Дополнительные материалы

### Документация:

- Logging tutorial
- Logging Cookbook
- Модуль python-json-logger

Альтернативы модулю logging

- loguru

### Статьи:

Основы:

- How To Use Logging in Python 3
- Python Logging Basics

Более подробные:

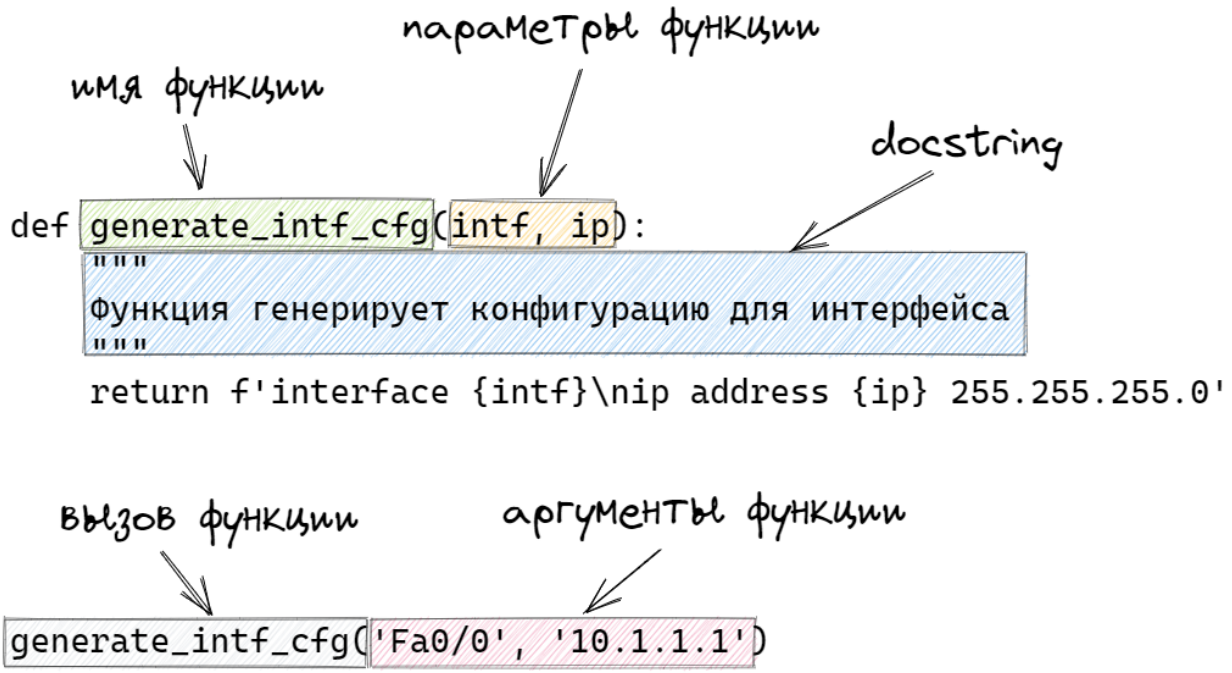
- A guide to logging in Python
- Good logging practice in Python
- Python Logging Tutorial
- Understanding Python’s logging module
- Пример YAML файла с конфигурацией logging

2

# II. Декораторы

# 6. Функции

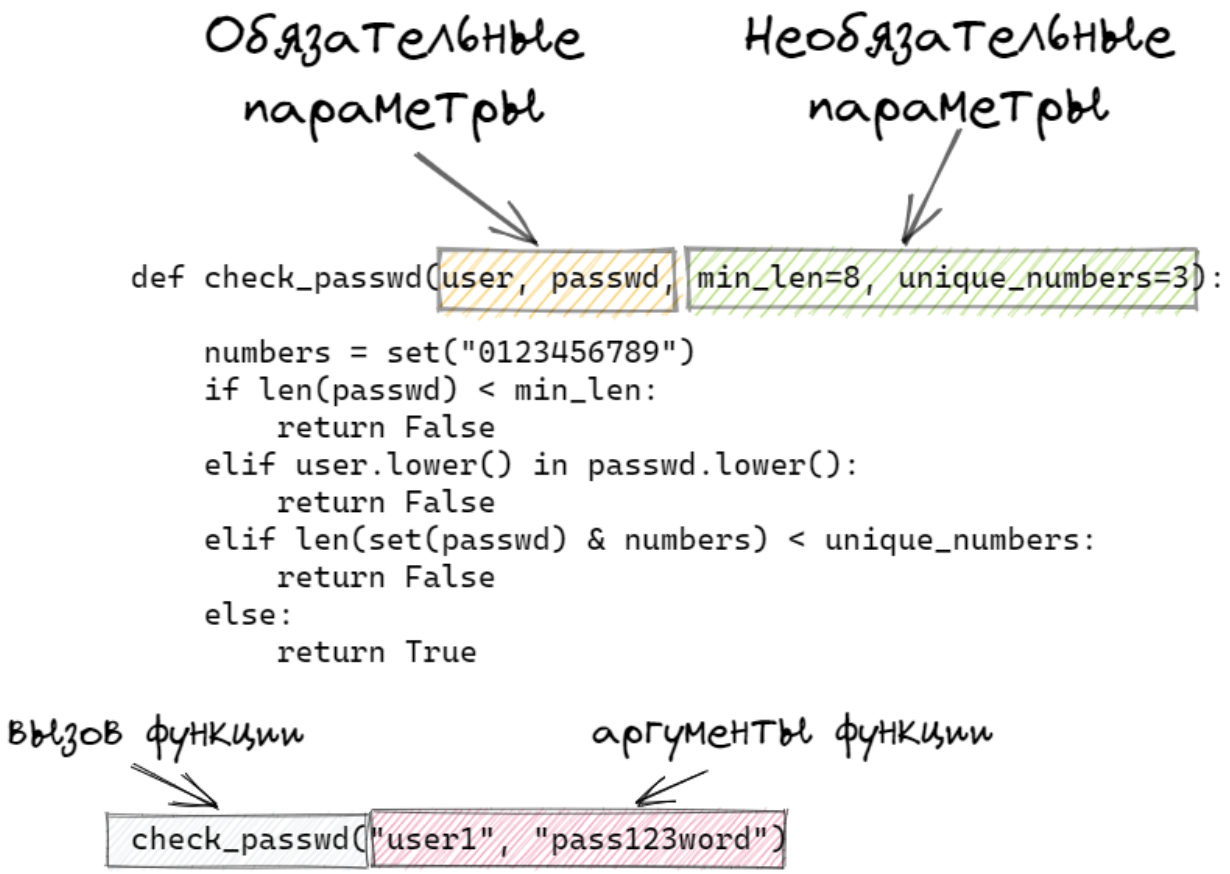
## Терминология




### Типы параметров функции

При создании функции можно указать, какие аргументы нужно передавать обязательно, а какие нет. Соответственно, функция может быть создана с:

- **обязательными параметрами**
- **необязательными параметрами** (опциональными, параметрами со значением по умолчанию)

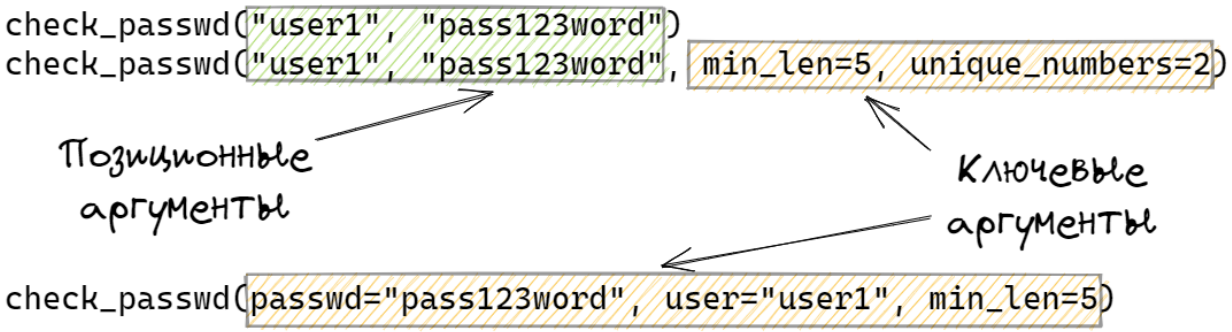



### Типы аргументов функции

При вызове функции аргументы можно передавать двумя способами:

- как **позиционные** - передаются в том же порядке, в котором они определены при создании функции. То есть, порядок передачи аргументов определяет, какое значение получит каждый аргумент
- как **ключевые** - передаются с указанием имени аргумента и его значения. В таком случае, аргументы могут быть указаны в любом порядке, так как их имя указывается явно.

```
def check_passwd(user, passwd, min_len=8, unique_numbers=3):
    numbers = set("0123456789")
    if len(passwd) < min_len:
        return False
    elif user.lower() in passwd.lower():
        return False
    elif len(set(passwd) & numbers) < unique_numbers:
        return False
    else:
        return True
```




### Аргументы переменной длины

Иногда необходимо сделать так, чтобы функция принимала не фиксированное количество аргументов, а любое. Для такого случая в Python можно создавать функцию со специальным параметром, который принимает аргументы переменной длины. Такой параметр может быть как ключевым, так и позиционным.

### Позиционные аргументы переменной длины

Параметр, который принимает позиционные аргументы переменной длины, создается добавлением перед именем параметра звездочки. Имя параметра может быть любым, но по договоренности чаще всего используют имя *args

Пример функции:

```
def sum_arg(a, *args):
    print(a, args)
    return a + sum(args)
```

Функция sum_arg создана с двумя параметрами:

- параметр a
  - если передается как позиционный аргумент, должен идти первым
  - если передается как ключевой аргумент, то порядок не важен

- параметр *args - ожидает аргументы переменной длины
  - сюда попадут все остальные аргументы в виде кортежа
  - эти аргументы могут отсутствовать

Вызов функции с разным количеством аргументов:

```
In [2]: sum_arg(1, 10, 20, 30)
1 (10, 20, 30)
Out[2]: 61

In [3]: sum_arg(1, 10)
1 (10,)
Out[3]: 11

In [4]: sum_arg(1)
1 ()
Out[4]: 1
```

Можно создать и такую функцию:

```
In [5]: def sum_arg(*args):
  ....:     print(args)
  ....:     return sum(args)
  ....:

In [6]: sum_arg(1, 10, 20, 30)
(1, 10, 20, 30)
Out[6]: 61

In [7]: sum_arg()
()
Out[7]: 0
```

### Ключевые аргументы переменной длины

Параметр, который принимает ключевые аргументы переменной длины, создается добавлением перед именем параметра двух звездочек. Имя параметра может быть любым, но, по договоренности, чаще всего, используют имя **kwargs (от keyword arguments).

Пример функции:

```
In [8]: def sum_arg(a, **kwargs):
  ....:     print(a, kwargs)
  ....:     return a + sum(kwargs.values())
  ....:
```

Функция sum_arg создана с двумя параметрами:

- параметр a
  - если передается как позиционный аргумент, должен идти первым
  - если передается как ключевой аргумент, то порядок не важен
- параметр **kwargs - ожидает ключевые аргументы переменной длины
  - сюда попадут все остальные ключевые аргументы в виде словаря
  - эти аргументы могут отсутствовать

Вызов функции с разным количеством ключевых аргументов:

```
In [9]: sum_arg(a=10, b=10, c=20, d=30)
10 {'c': 20, 'b': 10, 'd': 30}
Out[9]: 70

In [10]: sum_arg(b=10, c=20, d=30, a=10)
10 {'c': 20, 'b': 10, 'd': 30}
Out[10]: 70
```

## Пространства имен. Области видимости

Область видимости определяет где переменная доступна. Область видимость переменной зависит от того, где переменная создана.

Чаще всего, речь будет о двух областях видимости:

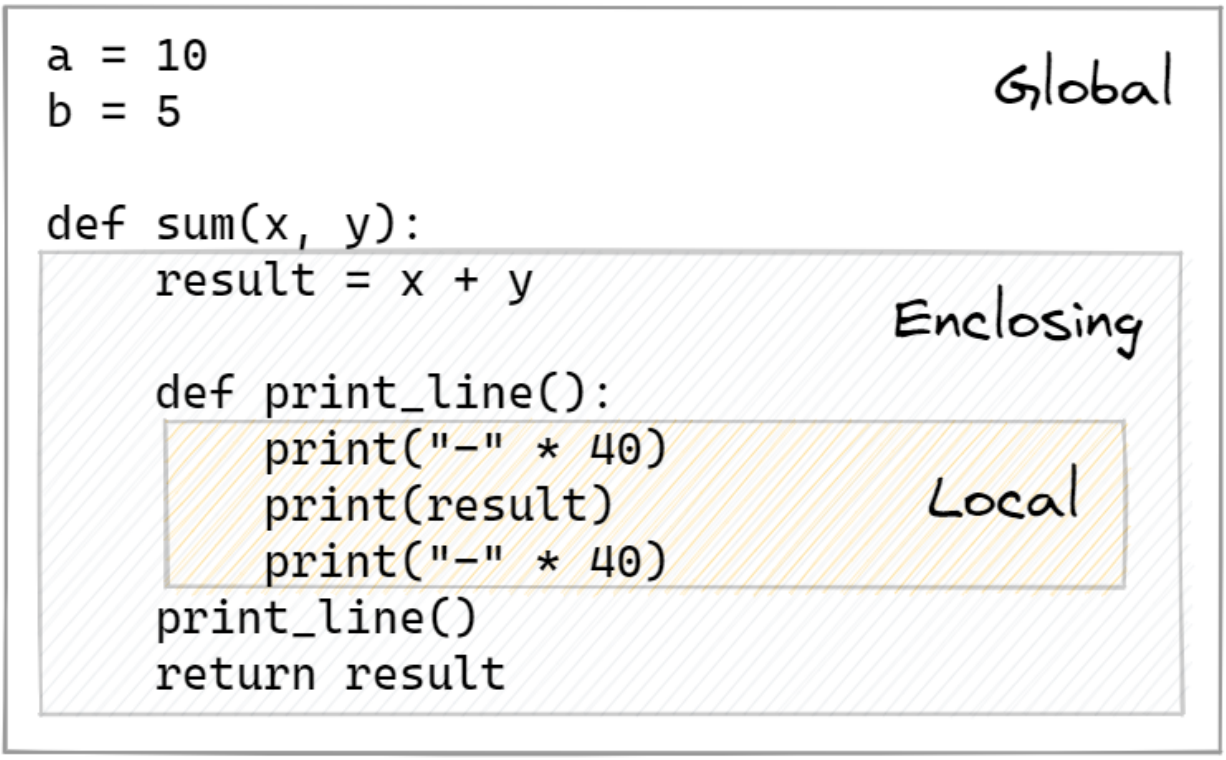
- глобальной - переменные, которые определены вне функции
- локальной - переменные, которые определены внутри функции

При использовании имен переменных в программе, Python каждый раз ищет, создает или изменяет эти имена в соответствующем пространстве имен. Пространство имен, которое доступно в каждый момент, зависит от области, в которой находится код.

### Поиск переменных

При поиске переменных, Python использует правило LEGB. Например, если внутри функции выполняется обращение к имени переменной, Python ищет переменную в таком порядке по областям видимости (до первого совпадения):

L (local) - в локальной (внутри функции) E (enclosing) - в локальной области объемлющих функций (это те функции, внутри которых находится наша функция) G (global) - в глобальной (в скрипте) B (built-in) - во встроенной (зарезервированные значения Python)



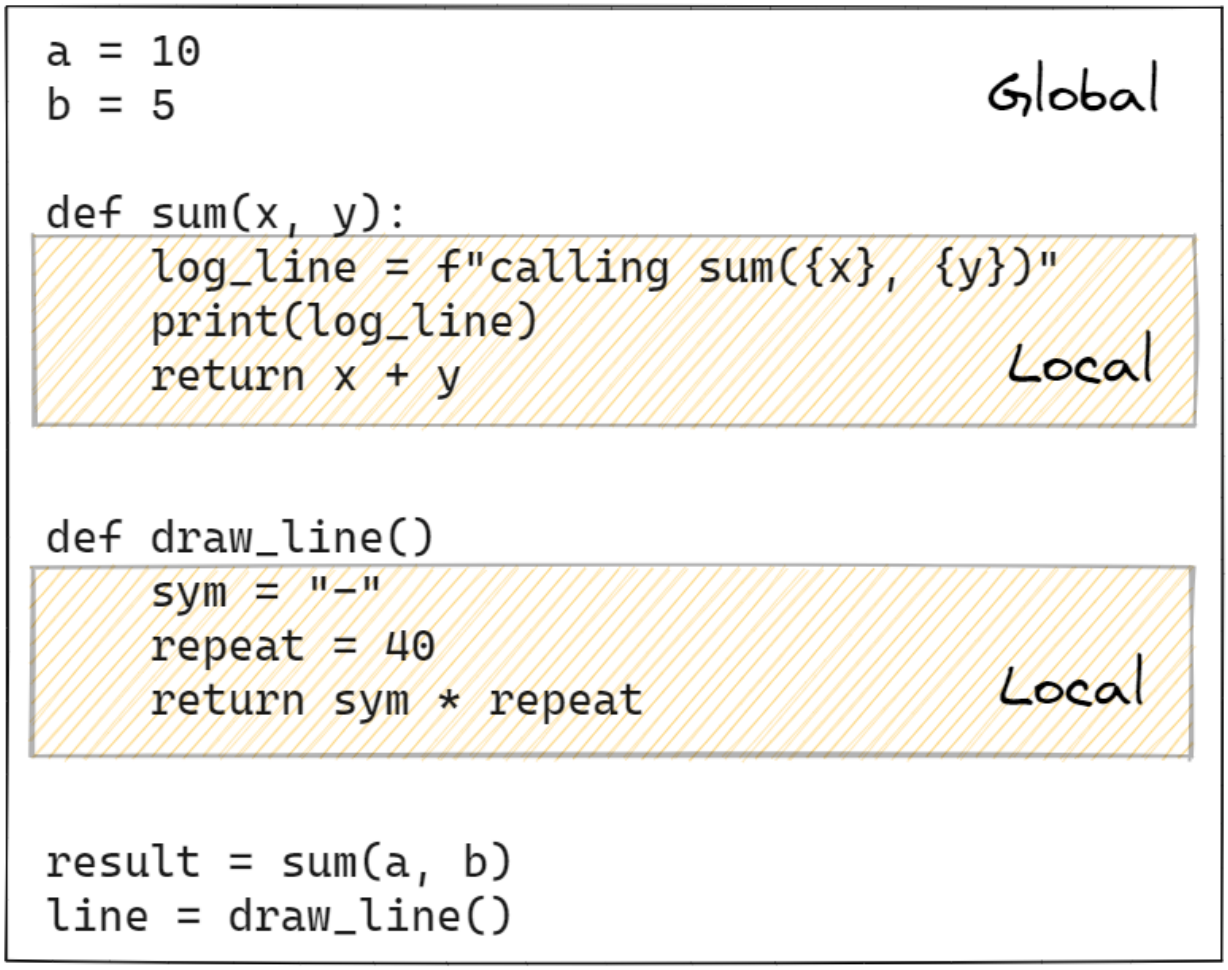

### Локальные и глобальные переменные

Локальные переменные:

- переменные, которые определены внутри функции
- эти переменные становятся недоступными после выхода из функции

Глобальные переменные:

- переменные, которые определены вне функции
- эти переменные „глобальны“ только в пределах модуля, чтобы они были доступны в другом модуле, их надо импортировать




Пример локальной intf_config:

```
In [1]: def configure_intf(intf_name, ip, mask):
   ...:     intf_config = f'interface {intf_name}\nip address {ip} {mask}'
   ...:     return intf_config
   ...:

In [2]: intf_config
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-5983e972fb1c> in <module>
----> 1 intf_config

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

Обратите внимание, что переменная intf_config недоступна за пределами функции. Для того чтобы получить результат функции, надо вызвать функцию и присвоить результат в переменную:

```
In [3]: result = configure_intf('F0/0', '10.1.1.1', '255.255.255.0')

In [4]: result
Out[4]: 'interface F0/0\nip address 10.1.1.1 255.255.255.0'
```

## Функции - объекты первого класса

В Python все функции являются объектами первого класса. Это означает, что Python поддерживает:

- передачу функций в качестве аргументов другим функциям
- возвращение функции как результата других функций
- присваивание функций переменным
- сохранение функций в структурах данных

Например, первый пункт «передача функций в качестве аргументов другим функциям» встречается при использовании встроенной функции map. Тут map применяет функцию str к каждому элементу списка:

```
In [1]: list(map(str, [1, 2, 3]))
Out[1]: ['1', '2', '3']
```

Функция delay ожидает как аргумент задержку в секундах, другую функцию и ее аргументы:

```
import time

def delay(seconds, func, *args, **kwargs):
    print(f'Delay {seconds} seconds...')
    time.sleep(seconds)
    return func(*args, **kwargs)
```

Теперь функции delay можно передавать любую другую функцию как аргумент и она выполнится после указанной паузы:

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


In [5]: delay(5, summ, 1, 4)
Delay 5 seconds...
Out[5]: 5
```

Сохранение функций в структурах данных:

```
In [8]: functions = [delay, summ]

In [9]: functions
Out[9]:
[<function __main__.delay(seconds, func, *args, **kwargs)>,
 <function __main__.summ(a, b)>]
```

Присваивание функций переменным:

```
In [10]: delay_execution = delay

In [11]: delay_execution
Out[11]: <function __main__.delay(seconds, func, *args, **kwargs)>

In [12]: delay_execution(5, summ, 1, 4)
Delay 5 seconds...
Out[12]: 5
```

## Полезные встроенные функции

### Работа с атрибутами объекта

Функции hasattr, getattr, delattr, setattr.

hasattr:

```
class Test:
    def __init__(self, name):
        self.name = name

    def method1(self):
        print("method1")


In [7]: t1 = Test("object1")

In [8]: hasattr(t1, "name")
Out[8]: True

In [9]: hasattr(t1, "test")
Out[9]: False

In [10]: hasattr(t1, "method1")
Out[10]: True
```

getattr

```
In [11]: getattr(t1, "method1")
Out[11]: <bound method Test.method1 of <__main__.Test object at 0xb5213ef8>>

In [12]: getattr(t1, "name")
Out[12]: 'object1'

In [13]: getattr(t1, "test")
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-13-200257dcfffb> in <module>
----> 1 getattr(t1, "test")

AttributeError: 'Test' object has no attribute 'test'

In [14]: getattr(t1, "test", None)

In [15]: getattr(t1, "test", False)
Out[15]: False
```

setattr

```
In [16]: setattr(t1, "test", False)

In [17]: t1.test
Out[17]: False
```

delattr

```
In [19]: delattr(t1, "test")

In [20]: t1.test
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-20-8ad1d771f5eb> in <module>
----> 1 t1.test

AttributeError: 'Test' object has no attribute 'test'
```

### vars

```
class Test:
    def __init__(self, name):
        self.name = name

    def method1(self):
        print("method1")
```

```
In [22]: vars(Test)
Out[22]:
mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.Test.__init__(self, name)>,
              'method1': <function __main__.Test.method1(self)>,
              '__dict__': <attribute '__dict__' of 'Test' objects>,
              '__weakref__': <attribute '__weakref__' of 'Test' objects>,
              '__doc__': None})
```

```
In [23]: t1 = Test("object1")

In [24]: vars(t1)
Out[24]: {'name': 'object1'}
```

### isinstance, issubclass

```
In [39]: from collections.abc import Iterator, Iterable

In [40]: vlans = [1, 2, 3]

In [41]: isinstance(vlans, list)
Out[41]: True

In [42]: isinstance(vlans, Iterable)
Out[42]: True

In [43]: isinstance(vlans, Iterator)
Out[43]: False
```

### callable

```
def summ(a, b):
    print(locals())
    return a + b


In [36]: callable(summ)
Out[36]: True

In [37]: callable(Test)
Out[37]: True

In [38]: callable(t1)
Out[38]: False
```

### dir

```
class Test:
    def __init__(self, name):
        self.name = name

    def method1(self):
        print("method1")


In [34]: dir(Test)
Out[34]:
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'method1']
```

### eval, exec

eval

```
In [29]: eval("10 + 5")
Out[29]: 15
```

exec

```
upper_func = """
def upper(string):
    return string.upper()
"""

In [31]: exec(upper_func)

In [32]: upper("test")
Out[32]: 'TEST'
```

### locals, globals

globals

```
In [25]: globals()
Out[25]:
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environment',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 '__builtins__': <module 'builtins' (built-in)>,
 ...
```

locals

```
def summ(a, b):
    print(locals())
    return a + b


In [28]: summ(3, 4)
{'a': 3, 'b': 4}
Out[28]: 7
```

## Дополнительные материалы

- Introspection of Function Parameters
- inspect
- rich inspect
- Introspection in Python

# 7. Closure

## Замыкание (Closure)

Замыкание (closure) — функция, которая находится внутри другой функции и ссылается на переменные объявленные в теле внешней функции (свободные переменные).

Внутренняя функция создается каждый раз во время выполнения внешней. Каждый раз при вызове внешней функции происходит создание нового экземпляра внутренней функции, с новыми ссылками на переменные внешней функции.

Ссылки на переменные внешней функции действительны внутри вложенной функции до тех пор, пока работает вложенная функция, даже если внешняя функция закончила работу, и переменные вышли из области видимости.

Пример замыкания:

```
def multiply(num1):
    var = 10
    def inner(num2):
        return num1 * num2
    return inner
```

Тут замыканием является функция inner. Функция inner использует внутри себя переменную num1 - параметр функции multiply, поэтому переменная num1 будет запомнена, а вот переменная var не используется и запоминатся не будет.

Использование созданной функции выглядит так:

Сначала делается вызов функции multiply с передачей одного аргумента, значение которого запишется в переменную num1:

```
In [2]: mult_by_9 = multiply(9)
```

Переменная mult_by_9 ссылается на внутреннюю функцию inner и при этом внутренняя функция помнит значение num1 = 9 и поэтому все числа будут умножаться на 9:

```
In [3]: mult_by_9
Out[3]: <function __main__.multiply.<locals>.inner(num2)>

In [4]: mult_by_9.__closure__
Out[4]: (<cell at 0xb0bd5f2c: int object at 0x836bf60>,)

In [5]: mult_by_9.__closure__[0].cell_contents
Out[5]: 9

In [8]: mult_by_9(10)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Out[8]: 90

In [9]: mult_by_9(2)
Out[9]: 18
```

Еще один пример замыкания с несколькими свободными переменными:

```
def func1():
    a = 1
    b = 'line'
    c = [1, 2, 3]

    def func2():
        return a, b, c

    return func2

In [11]: call_func = func1()

In [12]: call_func
Out[12]: <function __main__.func1.<locals>.func2()>

In [13]: call_func.__closure__
Out[13]:
(<cell at 0xb12170bc: int object at 0x836bee0>,
 <cell at 0xb12172e4: str object at 0xb732d720>,
 <cell at 0xb12177f4: list object at 0xb4e6d66c>)

In [14]: for item in call_func.__closure__:
    ...:     print(item, item.cell_contents)
    ...:
<cell at 0xb12170bc: int object at 0x836bee0> 1
<cell at 0xb12172e4: str object at 0xb732d720> line
<cell at 0xb12177f4: list object at 0xb4e6d66c> [1, 2, 3]
```

### Изменение свободных переменных

Для получения значения свободной переменной достаточно обратиться к ней, однако, при изменении значений есть нюансы. Если переменная ссылается на изменяемый объект, например, список, изменение содержимого делается стандартным образом без каких-либо проблем. Однако если необходимо, к примеру, добавить 1 к числу, мы получим ошибку:

```
def func1():
    a = 1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    b = 'line'
    c = [1, 2, 3]

    def func2():
        c.append(4)
        a = a + 1
        return a, b, c

    return func2


In [32]: call_func = func1()

In [33]: call_func()
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-33-9288e4e0f32f> in <module>
----> 1 call_func()

<ipython-input-31-56414e2c364b> in func2()
      6     def func2():
      7         c.append(4)
----> 8         a += 1
      9         return a, b, c
     10

UnboundLocalError: local variable 'a' referenced before assignment

In [34]: for item in call_func.__closure__:
    ...:     print(item, item.cell_contents)
    ...:
<cell at 0xb12174c4: str object at 0xb732d720> line
<cell at 0xb1217af4: list object at 0xb11e5dac> [1, 2, 3, 4]
```

Если необходимо присвоить свободной переменной другое значение, необходимо явно объявить ее как nonlocal:

```
def func1():
    a = 1
    b = 'line'
    c = [1, 2, 3]

    def func2():
        nonlocal a
        c.append(4)
        a += 1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        return a, b, c

    return func2


In [41]: call_func = func1()

In [42]: call_func()
Out[42]: (2, 'line', [1, 2, 3, 4])

In [43]: for item in call_func.__closure__:
    ...:     print(item, item.cell_contents)
    ...:
<cell at 0xb11fc6bc: int object at 0x836bef0> 2
<cell at 0xb11fcdac: str object at 0xb732d720> line
<cell at 0xb11fc56c: list object at 0xb117fe2c> [1, 2, 3, 4]
```

Использование nonlocal нужно только если необходимо менять свободную переменную сохраняя измененное значение между вызовами внутренней функции. Для обычного переприсваивания значения ничего делать не нужно.

Пример использования nonlocal с повторным вызовом внутренней функции:

```
def countdown(n):
    def step():
        nonlocal n
        r = n
        n -= 1
        return r
    return step

In [49]: do_step = countdown(10)

In [50]: do_step()
Out[50]: 10

In [51]: do_step()
Out[51]: 9

In [52]: do_step()
Out[52]: 8

In [53]: do_step()
Out[53]: 7
```

### Примеры использования замыкания

Пример с подключением SSH:

```
from netmiko import ConnectHandler

device_params = {
    'device_type': 'cisco_ios',
    'ip': '192.168.100.1',
    'username': 'cisco',
    'password': 'cisco',
    'secret': 'cisco'
}

def netmiko_ssh(**params_dict):
        ssh = ConnectHandler(**params_dict)
        ssh.enable()
        def send_show_command(command):
            return ssh.send_command(command)
        netmiko_ssh.send_show_command = send_show_command
        return send_show_command


In [25]: r1 = netmiko_ssh(**device_params)

In [26]: r1('sh clock')
Out[26]: '*15:14:13.240 UTC Wed Oct 2 2019'
```

## Дополнительные материалы

- `<>`__
- `<>`__

# 8. Декораторы

Декоратор в Python это функция, которая используется для изменения функции, метода или класса. Декораторы используются для добавления какого-то функционала к функциям/классам.

---

**Примечание:** Более полное определение декоратора: декоратор это вызываемый объект, который используется для изменения других вызываемых объектов.

---

## Декораторы без аргументов

Декоратор в Python это функция, которая используется для изменения функции, метода или класса. Декораторы используются для добавления какого-то функционала к функциям/классам.

Синтаксис декоратора - это синтаксический сахар, эти два определения функций эквивалентны:

```
def f(...):
    ...
f = verbose(f)

@verbose
def f(...):
    ...
```

Например, допустим, есть ряд функций к которым надо добавить print с информацией о том какая функция вызывается:

```
def upper(string):
    return string.upper()

def lower(string):
    return string.lower()

def capitalize(string):
    return string.capitalize()
```

Самый базовый вариант будет вручную добавить строку в каждой функции:

```
def upper(string):
    print('Вызываю функцию upper')
    return string.upper()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def lower(string):
    print('Вызываю функцию lower')
    return string.lower()

def capitalize(string):
    print('Вызываю функцию capitalize')
    return string.capitalize()
```

Однако в этом случае будет очень много повторений, а главное, при необходимости, например, заменить print на logging или просто изменить сообщение придется редактировать большое количество функций. Вместо этого можно создать одну функцию, которая перед вызовом исходной функции будет выводить сообщение:

```
def verbose(func):
    def wrapper(*args, **kwargs):
        print(f'Вызываю функцию {func.__name__}')
        return func(*args, **kwargs)
    return wrapper
```

Функция verbose принимает как аргумент функцию, а затем возвращает внутреннюю функцю wrapper внутри которой выводится сообщение, а затем вызывается исходная функция. Для того чтобы функция verbose работала надо заменить функцию upper внутренней функцией wrapper таким образом:

```
In [10]: upper = verbose(upper)
```

Теперь при вызове функции upper, вызывается внутренняя функция wrapper и перед вызовом самой upper выводится сообщение:

```
In [12]: upper(s)
Вызываю функцию upper
Out[12]: 'LINE'
```

К сожалению, в этом случае надо после определения каждой функции добавлять строку для модификации ее поведения:

```
def verbose(func):
    def wrapper(*args, **kwargs):
        print(f'Вызываю функцию {func.__name__}')
        return func(*args, **kwargs)
    return wrapper

def upper(string):
    return string.upper()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
upper = verbose(upper)

def lower(string):
    return string.lower()
lower = verbose(lower)

def capitalize(string):
    return string.capitalize()
capitalize = verbose(capitalize)
```

Так как показанный выше синтаксис не очень удобен, в Python есть другой синтаксис, который позволяет сделать то же самое более компактно:

```
def verbose(func):
    def wrapper(*args, **kwargs):
        print(f'Вызываю функцию {func.__name__}')
        return func(*args, **kwargs)
    return wrapper

@verbose
def upper(string):
    return string.upper()

@verbose
def lower(string):
    return string.lower()

@verbose
def capitalize(string):
    return string.capitalize()
```

При использовании декораторов, информация исходной функции заменяется внутренней функцией декоратора:

```
In [2]: lower
Out[2]: <function __main__.verbose.<locals>.wrapper(*args, **kwargs)>

In [4]: lower?
Signature: lower(*args, **kwargs)
Docstring: <no docstring>
File:      ~/repos/experiments/netdev_try/<ipython-input-1-32089045b87b>
Type:      function
```

Чтобы исправить это необходимо воспользоваться декоратором wraps из модуля functools:

```
from functools import wraps

def verbose(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f'Вызываю функцию {func.__name__}')
        return func(*args, **kwargs)
    return wrapper

@verbose
def upper(string):
    return string.upper()

@verbose
def lower(string):
    return string.lower()

@verbose
def capitalize(string):
    return string.capitalize()

In [7]: lower
Out[7]: <function __main__.lower(string)>

In [8]: lower?
Signature: lower(string)
Docstring: <no docstring>
File:      ~/repos/experiments/netdev_try/<ipython-input-6-13e6266ce16f>
Type:      function
```

Декоратор wraps переносит информацию исходной функции на внутреннюю и хотя это можно сделать и вручную, лучше пользоваться wraps.

## Примеры декораторов

Декоратор отображает: имя функции и значение аргументов:

```
def debugger_with_args(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f'Вызываю функцию {func.__name__} с args {args} и kwargs {kwargs}')
        return func(*args, **kwargs)
    return wrapper
```

(continues on next page)

(продолжение с предыдущей страницы)

```
@debugger_with_args
def func(a, b, verbose=True):
    return a, b, verbose


In [3]: func(1, 'a', verbose=False)
Вызываю функцию func с args (1, 'a') и kwargs {'verbose': False}
Out[3]: (1, 'a', False)
```

Декоратор проверяет что все аргументы функции - строки:

```
def all_args_str(func):
    @wraps(func)
    def wrapper(*args):
        if not all(isinstance(arg, str) for arg in args):
            raise ValueError('Все аргументы должны быть строками')
        return func(*args)
    return wrapper


@all_args_str
def to_upper(*args):
    result = [s.upper() for s in args]
    return result


In [6]: to_upper('a', 'b')
Out[6]: ['A', 'B']

In [7]: to_upper(1, 'b')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-7-bf0a0ae9f18c> in <module>
----> 1 to_upper(1, 'b')

<ipython-input-4-9ddfa715e195> in wrapper(*args)
      3     def wrapper(*args):
      4         if not all(isinstance(arg, str) for arg in args):
----> 5             raise ValueError('Все аргументы должны быть строками')
      6         return func(*args)
      7     return wrapper

ValueError: Все аргументы должны быть строками
```

## Примеры модулей которые используют декораторы

### pytest

```
@pytest.fixture(scope='module')
def first_router_from_devices_yaml():
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
        r1 = devices[0]
    return r1
```

### click

```
@click.command()
@click.option("--username", "-u", prompt=True)
@click.option("--password", "-p", prompt=True, hide_input=True, confirmation_prompt=True)
def cli(username, password):
    pass
```

### flask

```
@main.route('/')
def index():
    pass


@main.route('/labs', methods=['GET', 'POST'])
def labs():
    pass


@main.route('/stats')
def stats():
    pass
```

### backoff

```
@backoff.on_exception(
    backoff.expo, requests.exceptions.RequestException
)
def get_url(url):
    return requests.get(url)
```

### dataclasses.dataclass

```
@dataclass
class IPAddress:
    ip: str
    mask: int


In [12]: ip1 = IPAddress('10.1.1.1', 28)

In [13]: ip1
Out[13]: IPAddress(ip='10.1.1.1', mask=28)
```

## Стек декораторов

К функции может применяться несколько декораторов. Порядок применения декораторов будет зависеть от того в каком порядке они записаны:

```
def stars(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('*'*30)
        return func(*args, **kwargs)
    return wrapper


def lines(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('-'*30)
        return func(*args, **kwargs)
    return wrapper


def equals(func):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('='*30)
        return func(*args, **kwargs)
    return wrapper


@stars
@lines
@equals
def func(a, b):
    return a + b


In [23]: func(4,5)
******************************
------------------------------
==============================
Out[23]: 9

In [24]: def func(a, b):
    ...:     return a + b
    ...: func = stars(lines(equals(func)))

In [30]: func(4,5)
******************************
------------------------------
==============================
```

## Декораторы с аргументами

Иногда необходимо чтобы у декоратора была возможность принимать аргументы. В таком случае надо добавить еще один уровень вложенности для декоратора.

Самый базовый вариант декоратора с аргументами, когда функция не подменяется и аргументы функции не перехватываются. Тут к функции только добавляются атрибуты, которые указаны при вызове декоратора:

```
def add_mark(**kwargs):
    def decorator(func):
        for key, value in kwargs.items():
            setattr(func, key, value)
        return func
    return decorator
```

(continues on next page)

(продолжение с предыдущей страницы)

```
@add_mark(test=True, ordered=True)
def test_function(a, b):
    return a + b


In [73]: test_function.ordered
Out[73]: True

In [74]: test_function.test
Out[74]: True
```

Пошагово происходит следующее: сначала вызывается функция add_mark с соответствующими аргументами

```
decorate = add_mark(test=True, ordered=True)
```

Полученный результат будет декоратором, который ждет функцию как аргумент. То есть, то же самое можно сделать в два шага:

```
def add_mark(**kwargs):
    def decorator(func):
        for key, value in kwargs.items():
            setattr(func, key, value)
        return func
    return decorator

decorate = add_mark(test=True, ordered=True)

@decorate
def test_function(a, b):
    return a + b


In [73]: test_function.ordered
Out[73]: True

In [74]: test_function.test
Out[74]: True
```

Как только понадобится что-то делать с аргументами функции или добавить что-то до или после вызова функции, добавляется еще один уровень. Например, переделаем декоратор all_args_str таким образом, чтобы тип данных можно было передавать как аргумент. Декоратор all_args_str:

```
def all_args_str(func):
    @wraps(func)
    def wrapper(*args):
        if not all(isinstance(arg, str) for arg in args):
            raise ValueError('Все аргументы должны быть строками')
        return func(*args)
    return wrapper
```

Добавляем еще один уровень для добавления аргумента:

```
def restrict_args_type(required_type):
    def decorator(func):
        @wraps(func)
        def wrapper(*args):
            if not all(isinstance(arg, required_type) for arg in args):
                raise ValueError(f'Все аргументы должны быть {required_type.__name__}')
            return func(*args)
        return wrapper
    return decorator
```

Теперь, при применении декоратора, надо указывать какого типа должны быть аргументы:

```
In [89]: @restrict_args_type(str)
    ...: def to_upper(*args):
    ...:     result = [s.upper() for s in args]
    ...:     return result
    ...:

In [90]: to_upper('a', 2)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-90-b46c3ca71e5d> in <module>
----> 1 to_upper('a', 2)

<ipython-input-88-ea0c777e0f6e> in wrapper(*args)
      4             def wrapper(*args):
      5                 if not all(isinstance(arg, required_type) for arg in args):
----> 6                     raise ValueError(f'Все аргументы должны быть {required_type.__
↪name__}')
      7                 return func(*args)
      8             return wrapper

ValueError: Все аргументы должны быть str

In [91]: to_upper('a', 'a')
Out[91]: ['A', 'A']
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [93]: @restrict_args_type(int)
    ...: def to_bin(*args):
    ...:     result = [bin(a) for a in args]
    ...:     return result
    ...:

In [94]: to_bin(1,2,3)
Out[94]: ['0b1', '0b10', '0b11']

In [95]: to_bin('a', 'b')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-95-e4007cc06928> in <module>
----> 1 to_bin('a', 'b')

<ipython-input-88-ea0c777e0f6e> in wrapper(*args)
      4             def wrapper(*args):
      5                 if not all(isinstance(arg, required_type) for arg in args):
----> 6                     raise ValueError(f'Все аргументы должны быть {required_type.__
↪name__}')
      7                 return func(*args)
      8             return wrapper

ValueError: Все аргументы должны быть int
```

Также при необходимости можно сделать готовые декораторы для определенных типов данных:

```
In [96]: restrict_args_to_str = restrict_args_type(str)

In [97]: restrict_args_to_int = restrict_args_type(int)

In [98]: @restrict_args_to_str
    ...: def to_upper(*args):
    ...:     result = [s.upper() for s in args]
    ...:     return result
    ...:

In [99]: @restrict_args_to_int
    ...: def to_bin(*args):
    ...:     result = [bin(a) for a in args]
    ...:     return result
    ...:
```

## Примеры декораторов с аргументами

В Flask декораторы используются для сопоставления функция с ссылками на сайте:

```
url_function_map = {}

def register(route):
    def decorator(func):
        url_function_map[route] = func
        return func
    return decorator

@register('/')
def func(a,b):
    return a+b

@register('/scripts')
def func2(a,b):
    return a+b


In [3]: url_function_map
Out[3]: {'/': <function __main__.func>, '/scripts': <function __main__.func2>}
```

А также для ограничения доступа к определенным ссылкам:

```
from functools import wraps


class User:
    def __init__(self, username, permissions=None):
        self.username = username
        self.permissions = permissions

    def has_permission(self, permission):
        return permission in self.permissions


natasha = User('nata', ['admin', 'user'])
oleg = User('oleg', ['user'])

current_user = natasha


class AccessDenied(Exception):
    pass
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def permission_required(permission):
    def decorator(func):
        @wraps(func)
        def decorated_function(*args, **kwargs):
            if not current_user.has_permission(permission):
                raise AccessDenied('You shall not pass!')
            return func(*args, **kwargs)
        return decorated_function
    return decorator


@permission_required('admin')
def secret_func():
    return 42


In [77]: secret_func()
Out[77]: 42

In [78]: current_user = oleg

In [79]: secret_func()
---------------------------------------------------------------------------
AccessDenied                                 Traceback (most recent call last)
<ipython-input-79-23f2f66c4b3b> in <module>()
----> 1 secret_func()

<ipython-input-75-240afbb2dcfe> in decorated_function(*args, **kwargs)
      4         def decorated_function(*args, **kwargs):
      5             if not current_user.has_permission(permission):
----> 6                 raise AccessDenied('You shall not pass!')
      7             return func(*args, **kwargs)
      8         return decorated_function

AccessDenied: You shall not pass!
```

Иногда в зависимости от типа аргумента надо вызываться разные функции:

```
from netmiko import ConnectHandler
import yaml
from pprint import pprint


def send_show_command(device, show_command):
    with ConnectHandler(**device) as ssh:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        ssh.enable()
        result = ssh.send_command(show_command)
    return result

def send_config_commands(device, config_commands):
    with ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_config_set(config_commands)
    return result

def send_commands(device_list, config=None, show=None):
    if show:
        return send_show_command(device_list, show)
    elif config:
        return send_config_commands(device_list, config)



if __name__ == "__main__":
    commands = [ 'logging 10.255.255.1',
                 'logging buffered 20010',
                 'no logging console' ]
    show_command = "sh ip int br"
    with open('devices.yaml') as f:
        dev_list = yaml.safe_load(f)

    send_commands(dev_list, config=commands)
    send_commands(dev_list, show=show_command)
```

В стандартной библиотеке есть интересный декоратор singledispatch:

```
from netmiko import ConnectHandler
import yaml
from pprint import pprint
from functools import singledispatch
from collections.abc import Iterable, Sequence


@singledispatch
def send_commands(command, device):
    print('original func')
    raise NotImplementedError('Поддерживается только список или строка')

@send_commands.register(str)
def _(show_command, device):
    print('Выполняем show')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    with ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_command(show_command)
    return result

@send_commands.register(Iterable)
def _(config_commands, device):
    print('Выполняем config')
    with ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_config_set(config_commands)
    return result

if __name__ == "__main__":
    commands = ['logging 10.255.255.1',
                'logging buffered 20010',
                'no logging console' ]
    show_command = "sh ip int br"

    with open('devices.yaml') as f:
        r1 = yaml.safe_load(f)[0]

    print(send_commands(tuple(commands), r1))
    print(send_commands(show_command, r1))
```

## Декораторы в стандартной библиотеке

- classmethod
- staticmethod
- property
- contextlib: contextmanager
- dataclassess: dataclass

functools:

- cache
- lru_cache
- total_ordering
- singledispatch
- wraps

### property

```
class IPAddress:
    def __init__(self, address, mask):
        self._address = address
        self._mask = int(mask)

    @property
    def mask(self):
        return self._mask
```

### classmethod

```
class CiscoSSH(BaseSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)
        self._mgmt_ip = None

    @classmethod
    def default_params(cls, ip):
        params = {
            'ip': ip,
            'username': 'cisco',
            'password': 'cisco',
            'enable_password': 'cisco'}
        return cls(**params)
```

### staticmethod

```
class CiscoSSH(BaseSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)
        self._mgmt_ip = None

    @staticmethod
    def _parse_show(command, command_output,
                    index_file='index', templates='templates'):
        attributes = {'Command': command,
                      'Vendor': 'cisco_ios'}
        cli_table = clitable.CliTable(index_file, templates)
        cli_table.ParseCmd(command_output, attributes)
        return [dict(zip(cli_table.header, row)) for row in cli_table]

    def send_show_command(self, command, parse=True):
        command_output = super().send_show_command(command)
        if not parse:
            return command_output
        return self._parse_show(command, command_output)
```

### contextlib.contextmanager

```
from contextlib import contextmanager
from time import time, sleep


@contextmanager
def timecode():
    start = time()
    yield
    execution_time = time() - start
    print(f"Время выполнения: {execution_time:.2f}")


In [9]: with timecode():
   ...:     sleep(3)
   ...:
Время выполнения: 3.00
```

### dataclasses.dataclass

```
@dataclass
class IPAddress:
    ip: str
    mask: int


In [12]: ip1 = IPAddress('10.1.1.1', 28)

In [13]: ip1
Out[13]: IPAddress(ip='10.1.1.1', mask=28)
```

### functools.cache

```
from functools import cache, lru_cache


@cache
def factorial(n):
    print(f"{n=}")
    return n * factorial(n-1) if n else 1


print(f"{factorial(4)=}")
print(f"{factorial(5)=}")
print(f"{factorial(6)=}")
```

без cache

```
n=4
n=3
n=2
n=1
n=0
factorial(4)=24
n=5
n=4
n=3
n=2
n=1
n=0
factorial(5)=120
n=6
n=5
```

(continues on next page)

(продолжение с предыдущей страницы)

```
n=4
n=3
n=2
n=1
n=0
factorial(6)=720
```

с cache:

```
n=4
n=3
n=2
n=1
n=0
factorial(4)=24
n=5
factorial(5)=120
n=6
factorial(6)=720
```

### functools lru_cache

```
from functools import lru_cache


@lru_cache(maxsize=100)
def fib(n):
    print(f"{n=}")
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)


print([fib(n) for n in range(10)])
print([fib(n) for n in range(16)])
```

```
n=0
n=1
n=2
n=3
n=4
n=5
n=6
n=7
```

(continues on next page)

(продолжение с предыдущей страницы)

```
n=8
n=9
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
n=10
n=11
n=12
n=13
n=14
n=15
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

```
@lru_cache(maxsize=1)
def send_show_command(host, username, password, secret, device_type, show_command):
    with ConnectHandler(
        host=host,
        username=username,
        password=password,
        secret=secret,
        device_type=device_type,
    ) as ssh:
        ssh.enable()
        print(f"Вызываю команду {show_command}")
        result = ssh.send_command(show_command)
    return result
```

### functools.singledispatch

```
@singledispatch
def send_commands(command, device):
    print("singledispatch")
    raise NotImplementedError("Поддерживается только строка или iterable")


@send_commands.register(str)
def _(command, device):
    print("str")
    with ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
    return result


@send_commands.register(Iterable)
def _(config_commands, device):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    print("Аргумент iterable")
    with ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_config_set(config_commands)
    return result
```

## Декоратор класса

### Регистрация классов

```
CLASS_MAPPER_BASE = {}

def register_class(cls):
    CLASS_MAPPER_BASE[cls.device_type] = cls.__name__
    return cls


@register_class
class CiscoSSH:
    device_type = 'cisco_ios'
    def __init__(self, ip, username, password):
        pass


@register_class
class JuniperSSH:
    device_type = 'juniper'
    def __init__(self, ip, username, password):
        pass
```

### Декоратор добавляет метод pprint

```
from rich import print as rprint


def add_pprint(cls):
    def pprint(self, methods=False):
        methods_class_attrs = vars(type(self))
        methods = {
            name: method
            for name, method in methods_class_attrs.items()
            if not name.startswith("__") and callable(method)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        }
        self_attrs = vars(self)
        rprint(self_attrs)
        if methods:
            rprint(methods)

    cls.pprint = pprint
    return cls
```

```
@add_pprint
class IPv4Address:
    def __init__(self, ip):
        self.ip = ip
        self._int_ip = int(ip_address(ip))

    def as_int(self):
        return self._int_ip


In [15]: ip1 = IPv4Address("10.1.1.1")

In [16]: ip1.pprint()
{'ip': '10.1.1.1', '_int_ip': 167837953}
{
    'as_int': <function IPv4Address.as_int at 0xb4235df0>,
    'pprint': <function add_pprint.<locals>.pprint at 0xb4235388>
}

In [17]: ip1.pprint(methods=True)
{'ip': '10.1.1.1', '_int_ip': 167837953}
{
    'as_int': <function IPv4Address.as_int at 0xb4235df0>,
    'pprint': <function add_pprint.<locals>.pprint at 0xb4235388>
}
```

### Применение декоратора ко всем методам класса

```
def verbose(func):
    @wraps(func)
    def inner(*args, **kwargs):
        print(f"Вызываю {func.__name__}")
        print("Аргументы", args[1:], kwargs)
        return func(*args, **kwargs)
    return inner
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def verbose_methods(cls):
    methods = {
        name: method
        for name, method in vars(cls).items()
        if not name.startswith("__") and callable(method)
    }
    for name, method in methods.items():
        setattr(cls, name, verbose(method))
    return cls
```

## Класс как декоратор

### Декоратор без аргументов

```
class verbose:
    def __init__(self, func):
        print(f"Декорация функции {func.__name__}")
        self.func = func

    def __call__(self, *args, **kwargs):
        print(f"У функции {self.func.__name__} такие аргументы")
        print(f"{args=}")
        print(f"{kwargs=}")
        result = self.func(*args, **kwargs)
        return result


@verbose
def upper(string):
    return string.upper()
```

### Декоратор с аргументами

```
from functools import update_wrapper

class verbose:
    def __init__(self, message):
        print("вызов verbose")
        self.msg = message
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def __call__(self, func):
        print(f"Декорация функции {func.__name__}")
        def inner(*args, **kwargs):
            print(f"У функции {func.__name__} такие аргументы")
            print(f"{args=}")
            print(f"{kwargs=}")
            result = func(*args, **kwargs)

        update_wrapper(wrapper=inner, wrapped=func)
        return inner


@verbose("hello")
def upper(string):
    return string.upper()
```

## Дополнительные материалы

Примеры декораторов

- scrapli ChannelTimeout
- click
- backoff
- functools.lru_cache
- dataclasses.dataclass

Ссылки

- Примеры и идеи декораторов
- Primer on Python Decorators

# 3

# III. Объектно-ориентированное программирование

# 9. Основы ООП

## Основы ООП

- Класс (class) - элемент программы, который описывает какой-то тип данных. Класс описывает шаблон для создания объектов, как правило, указывает переменные этого объекта и действия, которые можно выполнять применимо к объекту.
- Экземпляр класса (instance) - объект, который является представителем класса.
- Метод (method) - функция, которая определена внутри класса и описывает какое-то действие, которое поддерживает класс
- Переменная экземпляра (instance variable, а иногда и instance attribute) - данные, которые относятся к объекту
- Переменная класса (class variable) - данные, которые относятся к классу и разделяются всеми экземплярами класса
- Атрибут экземпляра (instance attribute) - переменные и методы, которые относятся к объектам (экземплярам) созданным на основании класса. У каждого объекта есть своя копия атрибутов.

Пример из реальной жизни в стиле ООП:

- Проект дома - это класс
- Конкретный дом, который был построен по проекту - экземпляр класса
- Такие особенности как цвет дома, количество окон - переменные экземпляра, то есть конкретного дома
- Дом можно продать, перекрасить, отремонтировать - это методы

Например, при работе с netmiko первое, что нужно было сделать создать подключение:

```
from netmiko import ConnectHandler

device = {
    "device_type": "cisco_ios",
    "host": "192.168.100.1",
    "username": "cisco",
    "password": "cisco",
    "secret": "cisco",
}

ssh = ConnectHandler(**device)
```

Переменная ssh - это объект, который представляет реальное соединение с оборудованием. Благодаря функции type, можно выяснить экземпляром какого класса является объект ssh:

```
In [3]: type(ssh)
Out[3]: netmiko.cisco.cisco_ios.CiscoIosSSH
```

У ssh есть свои методы и переменные, которые зависят от состояния текущего объекта. Например, переменная экземпляра `ssh.host` доступна у каждого экземпляра класса `netmiko.cisco.cisco_ios.CiscoIosSSH` и возвращает IP-адрес или имя хоста, в зависимости от того что указывалось в словаре device:

```
In [4]: ssh.host
Out[4]: '192.168.100.1'
```

Метод send_command выполняет команду на оборудовании:

```
In [5]: ssh.send_command("sh clock")
Out[5]: '*10:08:50.654 UTC Tue Feb 2 2021'
```

Метод enable переходит в режим enable и при этом объект ssh сохраняет состояние: до и после перехода видно разное приглашение:

```
In [6]: ssh.find_prompt()
Out[6]: 'R1>'

In [7]: ssh.enable()
Out[7]: 'enable\r\nPassword: \r\nR1#'

In [8]: ssh.find_prompt()
Out[8]: 'R1#'
```

В этом примере показаны важные аспекты ООП: объединение данных и действия над данными, а также сохранение состояния.

До сих пор, при написании кода, данные и действия были разделены. Чаще всего, действия описаны в виде функций, а данные передаются как аргументы этим функциям. При создании класса, данные и действия объединяются. Конечно же, эти данные и действия связаны. То есть, методами класса становятся те действия, которые характерны именно для объекта такого типа, а не какие-то произвольные действия.

Например, в экзепляре класса str, все методы относятся к работе с этой строкой:

```
In [10]: s = 'string'

In [11]: s.upper()
Out[11]: 'STRING'

In [12]: s.center(20, '=')
Out[12]: '=======string======='
```

Выше, при обращении к атрибутам экземпляра (переменным и методам) используется такой синтаксис: `objectname.attribute`. Эта запись `s.lower()` означает: вызвать метод lower у объекта s. Обращение к методам и переменным выполняется одинаково, но для вызова метода, надо добавить скобки и передать все необходимые аргументы.

Всё описанное неоднократно использовалось в книге, но теперь мы разберемся с формальной терминологией.

## Создание класса

---

**Примечание:** Обратите внимание, что тут основы поясняются с учетом того, что у читающего нет опыта работы с ООП. Некоторые примеры не очень правильны с точки зрения идеологии Python, но помогают лучше понять происходящее. В конце даются пояснения как это правильней делать.

---

Для создания классов в питоне используется ключевое слово `class`. Самый простой класс, который можно создать в Python:

```
In [1]: class Switch:
   ...:     pass
   ...:
```

---

**Примечание:** Имена классов: в Python принято писать имена классов в формате CamelCase.

---

Для создания экземпляра класса, надо вызвать класс:

```
In [2]: sw1 = Switch()

In [3]: print(sw1)
<__main__.Switch object at 0xb44963ac>
```

Используя точечную нотацию, можно получать значения переменных экземпляра, создавать новые переменные и присваивать новое значение существующим:

```
In [5]: sw1.hostname = 'sw1'

In [6]: sw1.model = 'Cisco 3850'
```

В другом экземпляре класса Switch, переменные могут быть другие:

```
In [7]: sw2 = Switch()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [8]: sw2.hostname = 'sw2'

In [9]: sw2.model = 'Cisco 3750'
```

Посмотреть значение переменных экземпляра можно используя ту же точечную нотацию:

```
In [10]: sw1.model
Out[10]: 'Cisco 3850'

In [11]: sw2.model
Out[11]: 'Cisco 3750'
```

## Создание метода

Прежде чем мы начнем разбираться с методами класса, посмотрим пример функции, которая ожидает как аргумент экземпляр класса Switch и выводит информацию о нем, используя переменные экземпляра hostname и model:

```
In [1]: class Switch:
   ...:     pass
   ...:

In [2]: def info(sw_obj):
   ...:     print('Hostname: {}\nModel: {}'.format(sw_obj.hostname, sw_obj.model))
   ...:

In [3]: sw1 = Switch()

In [4]: sw1.hostname = 'sw1'

In [5]: sw1.model = 'Cisco 3850'

In [6]: info(sw1)
Hostname: sw1
Model: Cisco 3850
```

В функции info параметр `sw_obj` ожидает экземпляр класса `Switch`. Скорее всего, в этом примере нет ничего нового, ведь аналогичным образом ранее мы писали функции, которые ожидают строку, как аргумент, а затем вызывают какие-то методы у этой строки.

Этот пример поможет разобраться с методом info, который мы добавим в класс Switch.

Для добавления метода, необходимо создать функцию внутри класса:

```
In [15]: class Switch:
    ...:     def info(self):
    ...:         print('Hostname: {}\nModel: {}'.format(self.hostname, self.model))
    ...:
```

Если присмотреться, метод info выглядит точно так же, как функция info, только вместо имени sw_obj, используется self. Почему тут используется странное имя self, мы разберемся позже, а пока посмотрим как вызвать метод info:

```
In [16]: sw1 = Switch()

In [17]: sw1.hostname = 'sw1'

In [18]: sw1.model = 'Cisco 3850'

In [19]: sw1.info()
Hostname: sw1
Model: Cisco 3850
```

В примере выше сначала создается экземпляр класса Switch, затем в экземпляр добавляются переменные hostname и model, и только после этого вызывается метод info. Метод info выводит информацию про коммутатор, используя значения, которые хранятся в переменных экземпляра.

Вызов метода отличается, от вызова функции: мы не передаем ссылку на экземпляр класса Switch. Нам это не нужно, потому что мы вызываем метод у самого экземпляра. Еще один непонятный момент - зачем же мы тогда писали self.

Все дело в том, что Python преобразует такой вызов:

```
In [39]: sw1.info()
Hostname: sw1
Model: Cisco 3850
```

Вот в такой:

```
In [38]: Switch.info(sw1)
Hostname: sw1
Model: Cisco 3850
```

Во втором случае, в параметре self уже больше смысла, он действительно принимает ссылку на экземпляр и на основании этого выводит информацию.

С точки зрения использования объектов, удобней вызывать методы используя первый вариант синтаксиса, поэтому, практически всегда именно он и используется.

---

**Примечание:** При вызове метода экземпляра класса, ссылка на экземпляр передается пер-

вым аргументом. При этом, экземпляр передается неявно, но параметр надо указывать явно.

Такое преобразование не является особенностью пользовательских классов и работает и для встроенных типов данных аналогично. Например, стандартный способ вызова метода append в списке, выглядит так:

```
In [4]: a = [1,2,3]

In [5]: a.append(5)

In [6]: a
Out[6]: [1, 2, 3, 5]
```

При этом, то же самое можно сделать и используя второй вариант, вызова через класс:

```
In [7]: a = [1,2,3]

In [8]: list.append(a, 5)

In [9]: a
Out[9]: [1, 2, 3, 5]
```

## Параметр self

Параметр self указывался выше в определении методов, а также при использовании переменных экземпляра в методе. Параметр self это ссылка на конкретный экземпляр класса. При этом, само имя self не является особенным, а лишь договоренностью. Вместо self можно использовать другое имя, но так делать не стоит.

Пример с использованием другого имени, вместо self:

```
In [15]: class Switch:
    ...:     def info(sw_object):
    ...:         print(f'Hostname: {sw_object.hostname}\nModel: {sw_object.model}')
    ...:
```

Работать все будет аналогично:

```
In [16]: sw1 = Switch()

In [17]: sw1.hostname = 'sw1'

In [18]: sw1.model = 'Cisco 3850'

In [19]: sw1.info()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Hostname: sw1
Model: Cisco 3850
```

**Предупреждение:** Хотя технически использовать другое имя можно, всегда используйте self.

Во всех «обычных» методах класса первым параметром всегда будет self. Кроме того, создание переменной экземпляра внутри класса также выполняется через self.

Пример класса Switch с новым методом generate_interfaces: метод generate_interfaces должен сгенерировать список с интерфейсами на основании указанного типа и количества и создать переменную в экземпляре класса. Для начала, вариант создания обычно переменной внутри метода:

```
In [5]: class Switch:
   ...:     def generate_interfaces(self, intf_type, number_of_intf):
   ...:         interfaces = [f"{intf_type}{number}" for number in range(1, number_of_
↪intf + 1)]
   ...:
```

В этом случае, в экземплярах класса не будет переменной interfaces:

```
In [6]: sw1 = Switch()

In [7]: sw1.generate_interfaces('Fa', 10)

In [8]: sw1.interfaces
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-8-e6b457e4e23e> in <module>()
----> 1 sw1.interfaces

AttributeError: 'Switch' object has no attribute 'interfaces'
```

Этой переменной нет, потому что она существует только внутри метода, а область видимости у метода такая же, как и у функции. Даже другие методы одного и того же класса, не видят переменные в других методах.

Чтобы список с интерфейсами был доступен как переменная в экземплярах, надо присвоить значение в self.interfaces:

```
In [9]: class Switch:
   ...:     def info(self):
   ...:         print(f"Hostname: {self.hostname}\nModel: {self.model}")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...:
   ...:     def generate_interfaces(self, intf_type, number_of_intf):
   ...:         interfaces = [f"{intf_type}{number}" for number in range(1, number_of_
↪intf+1)]
   ...:         self.interfaces = interfaces
   ...:
```

Теперь, после вызова метода generate_interfaces, в экземпляре создается переменная interfaces:

```
In [10]: sw1 = Switch()

In [11]: sw1.generate_interfaces('Fa', 10)

In [12]: sw1.interfaces
Out[12]: ['Fa1', 'Fa2', 'Fa3', 'Fa4', 'Fa5', 'Fa6', 'Fa7', 'Fa8', 'Fa9', 'Fa10']
```

## Метод __init__

Для корректной работы метода info, необходимо чтобы у экземпляра были переменные hostname и model. Если этих переменных нет, возникнет ошибка:

```
In [15]: class Switch:
    ...:     def info(self):
    ...:         print('Hostname: {}\nModel: {}'.format(self.hostname, self.model))
    ...:

In [59]: sw2 = Switch()

In [60]: sw2.info()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-60-5a006dd8aae1> in <module>()
----> 1 sw2.info()

<ipython-input-57-30b05739380d> in info(self)
      1 class Switch:
      2     def info(self):
----> 3         print('Hostname: {}\nModel: {}'.format(self.hostname, self.model))

AttributeError: 'Switch' object has no attribute 'hostname'
```

Практически всегда, при создании объекта, у него есть какие-то начальные данные. Например, чтобы создать подключение к оборудование с помощью netmiko, надо передать параметры подключения.

В Python эти начальные данные про объект указываются в методе __init__. Метод __init__ выполняется после того как Python создал новый экземпляр и, при этом, методу __init__ передаются аргументы с которыми был создан экземпляр:

```
In [32]: class Switch:
    ...:     def __init__(self, hostname, model):
    ...:         self.hostname = hostname
    ...:         self.model = model
    ...:
    ...:     def info(self):
    ...:         print(f'Hostname: {self.hostname}\nModel: {self.model}')
    ...:
```

Обратите внимание на то, что у каждого экземпляра, который создан из этого класса, будут созданы переменные: `self.model` и `self.hostname`.

Теперь, при создании экземпляра класса Switch, обязательно надо указать hostname и model:

```
In [33]: sw1 = Switch('sw1', 'Cisco 3850')
```

И, соответственно, метод info отрабатывает без ошибок:

```
In [36]: sw1.info()
Hostname: sw1
Model: Cisco 3850
```

---

**Примечание:** Метод __init__ иногда называют конструктором класса, хотя технически в Python сначала выполняется метод __new__, а затем __init__. В большинстве случаев, метод __new__ использовать не нужно.

---

Важной особенностью метода __init__ является то, что он не должен ничего возвращать. Python сгенерирует исключение, если попытаться это сделать.

## Пример класса

Пример класса, который описывает сеть:

```
class Network:
    def __init__(self, network):
        self.network = network
        self.hosts = tuple(str(ip) for ip in ipaddress.ip_network(network).hosts())
        self.allocated = []

    def allocate(self, ip):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        if ip in self.hosts:
            if ip not in self.allocated:
                self.allocated.append(ip)
            else:
                raise ValueError(f"IP-адрес {ip} уже находится в allocated")
        else:
            raise ValueError(f"IP-адрес {ip} не входит в сеть {self.network}")
```

Использование класса:

```
In [2]: net1 = Network("10.1.1.0/29")

In [3]: net1.allocate("10.1.1.1")

In [4]: net1.allocate("10.1.1.2")

In [5]: net1.allocated
Out[5]: ['10.1.1.1', '10.1.1.2']

In [6]: net1.allocate("10.1.1.100")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-9a4157e02c78> in <module>
----> 1 net1.allocate("10.1.1.100")

<ipython-input-1-c5255d37a7fd> in allocate(self, ip)
     12                 raise ValueError(f"IP-адрес {ip} уже находится в allocated")
     13         else:
---> 14             raise ValueError(f"IP-адрес {ip} не входит в сеть {self.network}")
     15

ValueError: IP-адрес 10.1.1.100 не входит в сеть 10.1.1.0/29
```

## Область видимости

У каждого метода в классе своя локальная область видимости. Это значит, что один метод класса не видит переменные другого метода класса. Для того чтобы переменные были доступны, надо присваивать их экземпляру через `self.name`. По сути метод - это функция привязанная к объекту. Поэтому все нюансы, которые касаются функций, относятся и к методам.

Переменные экземпляра доступны в другом методе, потому что каждому методу первым аргументом передается сам экзепляр. В примере ниже, в методе `__init__` переменные hostname и model присваиваются экземпляру, а затем в info используются, за счет того, что экземпляр передается первым аргументом:

```
class Switch:
    def __init__(self, hostname, model):
        self.hostname = hostname
        self.model = model

    def info(self):
        print('Hostname: {}\nModel: {}'.format(self.hostname, self.model))
```

## Переменные класса

Помимо переменных экземпляра, существуют также переменные класса. Они создаются, при указании переменных внутри самого класса, не метода:

```
class Network:
    all_allocated_ip = []

    def __init__(self, network):
        self.network = network
        self.hosts = tuple(
            str(ip) for ip in ipaddress.ip_network(network).hosts()
        )
        self.allocated = []

    def allocate(self, ip):
        if ip in self.hosts:
            if ip not in self.allocated:
                self.allocated.append(ip)
                type(self).all_allocated_ip.append(ip)
            else:
                raise ValueError(f"IP-адрес {ip} уже находится в allocated")
        else:
            raise ValueError(f"IP-адрес {ip} не входит в сеть {self.network}")
```

К переменным класса можно обращаться по-разному:

- `self.all_allocated_ip`
- `Network.all_allocated_ip`
- `type(self).all_allocated_ip`

Вариант `self.all_allocated_ip` позволяет обратиться к значению переменной класса или добавить элемент, если переменная класса изменяемый тип данных. Минус этого варианта в том, что если в методе написать `self.all_allocated_ip = ...`, вместо изменения переменной класса, будет создана переменная экземпляра.

Вариант `Network.all_allocated_ip` будет работать корректно, но небольшой минус этого

варианта в том, что имя класса прописано вручную. Вместо него можно использовать третий вариант `type(self).all_allocated_ip`, так как `type(self)` возвращает класс.

Теперь у класса есть переменная all_allocated_ip в которую записываются все IP-адреса, которые выделены в сетях:

```
In [3]: net1 = Network("10.1.1.0/29")

In [4]: net1.allocate("10.1.1.1")
   ...: net1.allocate("10.1.1.2")
   ...: net1.allocate("10.1.1.3")
   ...:

In [5]: net1.allocated
Out[5]: ['10.1.1.1', '10.1.1.2', '10.1.1.3']

In [6]: net2 = Network("10.2.2.0/29")

In [7]: net2.allocate("10.2.2.1")
   ...: net2.allocate("10.2.2.2")
   ...:

In [9]: net2.allocated
Out[9]: ['10.2.2.1', '10.2.2.2']

In [10]: Network.all_allocated_ip
Out[10]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']
```

Переменная доступна не только через класс, но и через экземпляры:

```
In [40]: Network.all_allocated_ip
Out[40]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']

In [41]: net1.all_allocated_ip
Out[41]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']

In [42]: net2.all_allocated_ip
Out[42]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']
```

# 10. Специальные методы

Специальные методы в Python - это методы, которые отвечают за «стандартные» возможности объектов и вызываются автоматически при использовани этих возможностей. Например, выражение a + b, где a и b это числа, преобразуется в такой вызов a.__add__(b), то есть, специальный метод __add__ отвечает за операцию сложения. Все специальные методы начинаются и заканчиваются двойным подчеркиванием, поэтому на английском их часто называют dunder методы, сокращенно от «double underscore».

---

**Примечание:** Специальные методы часто называют волшебными (magic) методами.

---

Специальные методы отвечают за такие возможности как работа в менеджерах контекста, создание итераторов и итерируемых объектов, операции сложения, умножения и другие. Добавляя специальные методы в объекты, которые созданы пользователем, мы делаем их похожими на встроенные объекты.

## Подчеркивание в именах

В Python подчеркивание в начале или в конце имени указывает на специальные имена. Чаще всего это всего лишь договоренность, но иногда это действительно влияет на поведение объекта.

### Одно подчеркивание перед именем

Одно подчеркивание перед именем метода указывает, что метод является внутренней особенностью реализации и его не стоит использовать напрямую.

Например, класс CiscoSSH использует paramiko для подключения к оборудованию:

```
import time
import paramiko


class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        self.client = paramiko.SSHClient()
        self.client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        self.client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            allow_agent=False)

        self.ssh = self.client.invoke_shell()
        self.ssh.send('enable\n')
        self.ssh.send(enable + '\n')
        if disable_paging:
            self.ssh.send('terminal length 0\n')
        time.sleep(1)
        self.ssh.recv(1000)

    def send_show_command(self, command):
        self.ssh.send(command + '\n')
        time.sleep(2)
        result = self.ssh.recv(5000).decode('ascii')
        return result
```

После создания экземпляра класса, доступен не только метод send_show_command, но и атрибуты client и ssh (3 строка это подсказки по tab в ipython):

```
In [2]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [3]: r1.
            client
            send_show_command()
            ssh
```

Если же необходимо указать, что client и ssh являются внутренними атрибутами, которые нужны для работы класса, но не предназначены для пользователя, надо поставить нижнее подчеркивание перед именем:

```
class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        self._client = paramiko.SSHClient()
        self._client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        self._client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self._ssh = self._client.invoke_shell()
        self._ssh.send('enable\n')
        self._ssh.send(enable + '\n')
        if disable_paging:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(1000)

    def send_show_command(self, command):
        self._ssh.send(command + '\n')
        time.sleep(2)
        result = self._ssh.recv(5000).decode('ascii')
        return result
```

**Примечание:** Часто такие методы и атрибуты называются приватными, но это не значит, что методы и переменные недоступны пользователю.

### Два подчеркивания перед именем

Два подчеркивания перед именем метода используются не просто как договоренность. Такие имена трансформируются в формат «имя класса + имя метода». Это позволяет создавать уникальные методы и атрибуты классов.

Такое преобразование выполняется только в том случае, если в конце менее двух подчеркиваний или нет подчеркиваний.

```
In [14]: class Switch(object):
    ...:     __quantity = 0
    ...:
    ...:     def __configure(self):
    ...:         pass
    ...:

In [15]: dir(Switch)
Out[15]:
['_Switch__configure', '_Switch__quantity', ...]
```

Хотя методы создавались без приставки _Switch, она была добавлена.

Если создать подкласс, то метод __configure не перепишет метод родительского класса Switch:

```
In [16]: class CiscoSwitch(Switch):
    ...:     __quantity = 0
    ...:     def __configure(self):
    ...:         pass
    ...:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [17]: dir(CiscoSwitch)
Out[17]:
['_CiscoSwitch__configure', '_CiscoSwitch__quantity', '_Switch__configure', '_Switch__
↪quantity', ...]
```

### Два подчеркивания перед и после имени

Таким образом обозначаются специальные переменные и методы.

Например, в модуле Python есть такие специальные переменные:

- `__name__` - эта переменная равна строке `__main__`, когда скрипт запускается напрямую, и равна имени модуля, когда импортируется
- `__file__` - эта переменная равна имени скрипта, который был запущен напрямую, и равна полному пути к модулю, когда он импортируется

Переменная `__name__` чаще всего используется, чтобы указать, что определенная часть кода должна выполняться, только когда модуль выполняется напрямую:

```
def multiply(a, b):

    return a * b

if __name__ == '__main__':
    print(multiply(3, 5))
```

А переменная `__file__` может быть полезна в определении текущего пути к файлу скрипта:

```
import os

print('__file__', __file__)
print(os.path.abspath(__file__))
```

Вывод будет таким:

```
__file__ example2.py
/home/vagrant/repos/tests/example2.py
```

Кроме того, таким образом в Python обозначаются специальные методы. Эти методы вызываются при использовании функций и операторов Python и позволяют реализовать определенный функционал.

Как правило, такие методы не нужно вызывать напрямую. Но, например, при создании своего класса может понадобиться описать такой метод, чтобы объект поддерживал какие-то операции в Python.

Например, для того, чтобы можно было получить длину объекта, он должен поддерживать метод `__len__`.

## Методы __str__, __repr__

Специальные методы __str__ и __repr__ отвечают за строковое представления объекта. При этом используются они в разных местах.

Рассмотрим пример класса IPAddress, который отвечает за представление IPv4 адреса:

```
In [1]: class IPAddress:
   ...:     def __init__(self, ip):
   ...:         self.ip = ip
   ...:
```

После создания экземпляров класса, у них есть строковое представление по умолчанию, которое выглядит так (этот же вывод отображается при использовании print):

```
In [2]: ip1 = IPAddress('10.1.1.1')

In [3]: ip2 = IPAddress('10.2.2.2')

In [4]: str(ip1)
Out[4]: '<__main__.IPAddress object at 0xb4e4e76c>'

In [5]: str(ip2)
Out[5]: '<__main__.IPAddress object at 0xb1bd376c>'
```

К сожалению, это представление не очень информативно. И было бы лучше, если бы отображалась информация о том, какой именно адрес представляет этот экземпляр. За отображение информации при применении функции str, отвечает специальный метод __str__ - как аргумент метод ожидает только экземпляр и должен возвращать строку

```
In [6]: class IPAddress:
   ...:     def __init__(self, ip):
   ...:         self.ip = ip
   ...:
   ...:     def __str__(self):
   ...:         return f"IPAddress: {self.ip}"
   ...:

In [7]: ip1 = IPAddress('10.1.1.1')

In [8]: ip2 = IPAddress('10.2.2.2')

In [9]: str(ip1)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Out[9]: 'IPAddress: 10.1.1.1'

In [10]: str(ip2)
Out[10]: 'IPAddress: 10.2.2.2'
```

Второе строковое представление, которое используется в объектах Python, отображается при использовании функции repr, а также при добавлении объектов в контейнеры типа списков:

```
In [11]: ip_addresses = [ip1, ip2]

In [12]: ip_addresses
Out[12]: [<__main__.IPAddress at 0xb4e40c8c>, <__main__.IPAddress at 0xb1bc46ac>]

In [13]: repr(ip1)
Out[13]: '<__main__.IPAddress object at 0xb4e40c8c>'
```

За это отображение отвечает метод __repr__, он тоже должен возвращать строку, но при этом принято, чтобы метод возвращал строку, скопировав которую, можно получить экземпляр класса:

```
In [14]: class IPAddress:
    ...:     def __init__(self, ip):
    ...:         self.ip = ip
    ...:
    ...:     def __str__(self):
    ...:         return f"IPAddress: {self.ip}"
    ...:
    ...:     def __repr__(self):
    ...:         return f"IPAddress('{self.ip}')"
    ...:

In [15]: ip1 = IPAddress('10.1.1.1')

In [16]: ip2 = IPAddress('10.2.2.2')

In [17]: ip_addresses = [ip1, ip2]

In [18]: ip_addresses
Out[18]: [IPAddress('10.1.1.1'), IPAddress('10.2.2.2')]

In [19]: repr(ip1)
Out[19]: "IPAddress('10.1.1.1')"
```

## Поддержка арифметических операторов

За поддержку арифметических операций также отвечают специальные методы, например, за операцию сложения отвечает метод __add__:

```
__add__(self, other)
```

Добавим к классу IPAddress поддержку суммирования с числами, но чтобы не усложнять реализацию метода, воспользуемся возможностями модуля ipaddress

```
In [1]: import ipaddress

In [2]: ipaddress1 = ipaddress.ip_address('10.1.1.1')

In [3]: int(ipaddress1)
Out[3]: 167837953

In [4]: ipaddress.ip_address(167837953)
Out[4]: IPv4Address('10.1.1.1')
```

Класс IPAddress с методом __add__:

```
In [5]: class IPAddress:
   ...:     def __init__(self, ip):
   ...:         self.ip = ip
   ...:
   ...:     def __str__(self):
   ...:         return f"IPAddress: {self.ip}"
   ...:
   ...:     def __repr__(self):
   ...:         return f"IPAddress('{self.ip}')"
   ...:
   ...:     def __add__(self, other):
   ...:         ip_int = int(ipaddress.ip_address(self.ip))
   ...:         sum_ip_str = str(ipaddress.ip_address(ip_int + other))
   ...:         return IPAddress(sum_ip_str)
   ...:
```

Переменная ip_int ссылается на значение исходного адреса в десятичном формате. а sum_ip_str это строка с IP-адресом полученным в результате сложения двух чисел. Как правило, желательно чтобы операция суммирования возвращала экземпляр того же класса, поэтому в последней строке метода создается экземпляр класса IPAddress и ему как аргумент передается строка с итоговым адресом.

Теперь экземпляры класса IPAddress должны поддерживать операцию сложения с числом. В результате мы получаем новый экземпляр класса IPAddress.

```
In [6]: ip1 = IPAddress('10.1.1.1')

In [7]: ip1 + 5
Out[7]: IPAddress('10.1.1.6')
```

Так как внутри метода используется модуль ipaddress, а он поддерживает создание IP-адреса только из десятичного числа, надо ограничить метод на работу только с данными типа int. Если же второй элемент был объектом другого типа, надо сгенерировать исключение. Исключение и сообщение об ошибке возьмем из аналогичной ошибки функции ipaddress.ip_address:

```
In [8]: a1 = ipaddress.ip_address('10.1.1.1')

In [9]: a1 + 4
Out[9]: IPv4Address('10.1.1.5')

In [10]: a1 + 4.0
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-a0a045adedc5> in <module>
----> 1 a1 + 4.0

TypeError: unsupported operand type(s) for +: 'IPv4Address' and 'float'
```

Теперь класс IPAddress выглядит так:

```
In [11]: class IPAddress:
    ...:     def __init__(self, ip):
    ...:         self.ip = ip
    ...:
    ...:     def __str__(self):
    ...:         return f"IPAddress: {self.ip}"
    ...:
    ...:     def __repr__(self):
    ...:         return f"IPAddress('{self.ip}')"
    ...:
    ...:     def __add__(self, other):
    ...:         if not isinstance(other, int):
    ...:             raise TypeError(f"unsupported operand type(s) for +:"
    ...:                             f" 'IPAddress' and '{type(other).__name__}'")
    ...:
    ...:         ip_int = int(ipaddress.ip_address(self.ip))
    ...:         sum_ip_str = str(ipaddress.ip_address(ip_int + other))
    ...:         return IPAddress(sum_ip_str)
    ...:
```

Если второй операнд не является экзепляром класса int, генерируется исключение TypeError. В исключении выводится информация, что суммирование не поддерживается между экзем-

плярами класса IPAddress и экземпляром класса операнда. Имя класса получено из самого класса, после обращения к type: `type(other).__name__`.

Проверка суммирования с десятичным числом и генерации ошибки:

```
In [12]: ip1 = IPAddress('10.1.1.1')

In [13]: ip1 + 5
Out[13]: IPAddress('10.1.1.6')

In [14]: ip1 + 5.0
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-5e619f8dc37a> in <module>
----> 1 ip1 + 5.0

<ipython-input-11-77b43bc64757> in __add__(self, other)
     11     def __add__(self, other):
     12         if not isinstance(other, int):
---> 13             raise TypeError(f"unsupported operand type(s) for +:"
     14                             f" 'IPAddress' and '{type(other).__name__}'")
     15

TypeError: unsupported operand type(s) for +: 'IPAddress' and 'float'

In [15]: ip1 + '1'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-c5ce818f55d8> in <module>
----> 1 ip1 + '1'

<ipython-input-11-77b43bc64757> in __add__(self, other)
     11     def __add__(self, other):
     12         if not isinstance(other, int):
---> 13             raise TypeError(f"unsupported operand type(s) for +:"
     14                             f" 'IPAddress' and '{type(other).__name__}'")
     15

TypeError: unsupported operand type(s) for +: 'IPAddress' and 'str'
```

**См.также:**

Руководство по специальным методам (англ) Numeric magic methods

## Протоколы

Специальные методы отвечают не только за поддержку операций типа сложение, сравнение, но и за поддержку протоколов. Протокол - это набор методов, которые должны быть реализованы в объекте, чтобы он поддерживал определенное поведение. Например, в Python есть такие протоколы: итерации, менеджер контекста, контейнеры и другие. После создания в объекте определенных методов, объект будет вести себя как встроенный и использовать интерфейс понятный всем, кто пишет на Python.

---

**Примечание:** Таблица с абстрактных классов в которой описаны какие методы должны присутствовать у объекта, чтобы он поддерживал определенный протокол

---

### Протокол итерации

**Итерируемый объект (iterable)** - это объект, который способен возвращать элементы по одному. Для Python это любой объект у которого есть метод `__iter__` или метод `__getitem__`. Если у объекта есть метод __iter__, итерируемый объект превращается в итератор вызовом `iter(name)`, где name - имя итерируемого объекта. Если метода __iter__ нет, Python перебирает элементы используя __getitem__.

```
class Items:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        print('Вызываю __getitem__')
        return self.items[index]


In [2]: iterable_1 = Items([1, 2, 3, 4])

In [3]: iterable_1[0]
Вызываю __getitem__
Out[3]: 1

In [4]: for i in iterable_1:
   ...:     print('>>>>', i)
   ...:
Вызываю __getitem__
>>>> 1
Вызываю __getitem__
>>>> 2
Вызываю __getitem__
```

(continues on next page)

(продолжение с предыдущей страницы)

```
>>>> 3
Вызываю __getitem__
>>>> 4
Вызываю __getitem__

In [5]: list(map(str, iterable_1))
Вызываю __getitem__
Вызываю __getitem__
Вызываю __getitem__
Вызываю __getitem__
Вызываю __getitem__
Out[5]: ['1', '2', '3', '4']
```

Если у объекта есть метод __iter__ (который обязан возвращать итератор), при переборе значений используется он:

```
class Items:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        print('Вызываю __getitem__')
        return self.items[index]

    def __iter__(self):
        print('Вызываю __iter__')
        return iter(self.items)


In [12]: iterable_1 = Items([1, 2, 3, 4])

In [13]: for i in iterable_1:
     ...:     print('>>>>', i)
     ...:
Вызываю __iter__
>>>> 1
>>>> 2
>>>> 3
>>>> 4

In [14]: list(map(str, iterable_1))
Вызываю __iter__
Out[14]: ['1', '2', '3', '4']
```

В Python за получение итератора отвечает функция iter():

```
In [1]: lista = [1, 2, 3]

In [2]: iter(lista)
Out[2]: <list_iterator at 0xb4ede28c>
```

Функция iter отработает на любом объекте, у которого есть метод __iter__ или метод __getitem__. Метод __iter__ возвращает итератор. Если этого метода нет, функция iter() проверяет, нет ли метода __getitem__ - метода, который позволяет получать элементы по индексу. Если метод __getitem__ есть, элементы будут перебираться по индексу (начиная с 0).

**Итератор (iterator)** - это объект, который возвращает свои элементы по одному за раз. С точки зрения Python - это любой объект, у которого есть метод __next__. Этот метод возвращает следующий элемент, если он есть, или возвращает исключение StopIteration, когда элементы закончились. Кроме того, итератор запоминает, на каком объекте он остановился в последнюю итерацию. Также у каждого итератора присутствует метод __iter__ - то есть, любой итератор является итерируемым объектом. Этот метод возвращает сам итератор.

Пример создания итератора из списка:

```
In [3]: lista = [1, 2, 3]

In [4]: i = iter(lista)
```

Теперь можно использовать функцию next(), которая вызывает метод __next__, чтобы взять следующий элемент:

```
In [5]: next(i)
Out[5]: 1

In [6]: next(i)
Out[6]: 2

In [7]: next(i)
Out[7]: 3

In [8]: next(i)
---------------------------------------------------------------
StopIteration             Traceback (most recent call last)
<ipython-input-8-bed2471d02c1> in <module>()
----> 1 next(i)

StopIteration:
```

После того, как элементы закончились, возвращается исключение StopIteration. Для того, чтобы итератор снова начал возвращать элементы, его надо заново создать. Аналогичные действия выполяются, когда цикл for проходится по списку:

```
In [9]: for item in lista:
   ...:     print(item)
   ...:
1
2
3
```

Когда мы перебираем элементы списка, к списку сначала применяется функция iter(), чтобы создать итератор, а затем вызывается его метод __next__ до тех пор, пока не возникнет исключение StopIteration.

Пример функции my_for, которая работает с любым итерируемым объектом и имитирует работу встроенной функции for:

```
def my_for(iterable):
    if getattr(iterable, "__iter__", None):
        print('Есть __iter__')
        iterator = iter(iterable)
        while True:
            try:
                print(next(iterator))
            except StopIteration:
                break
    elif getattr(iterable, "__getitem__", None):
        print('Нет __iter__, но есть __getitem__')
        index = 0
        while True:
            try:
                print(iterable[index])
            except IndexError:
                break
```

Проверка работы функции на объекте у которого есть метод __iter__:

```
In [18]: my_for([1,2,3,4])
Есть __iter__
1
2
3
4
```

Проверка работы функции на объекте у которого нет метода __iter__, но есть __getitem__:

```
class Items:
    def __init__(self, items):
        self.items = items
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def __getitem__(self, index):
        print('Вызываю __getitem__')
        return self.items[index]


In [20]: iterable_1 = Items([1,2,3,4,5])

In [21]: my_for(iterable_1)
Нет __iter__, но есть __getitem__
Вызываю __getitem__
1
Вызываю __getitem__
2
Вызываю __getitem__
3
Вызываю __getitem__
4
Вызываю __getitem__
5
Вызываю __getitem__
```

### Создание итератора

Пример класса Network:

```
In [10]: import ipaddress
    ...:
    ...: class Network:
    ...:     def __init__(self, network):
    ...:         self.network = network
    ...:         subnet = ipaddress.ip_network(self.network)
    ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
```

Пример создания экземпляра класса Network:

```
In [14]: net1 = Network('10.1.1.192/30')

In [15]: net1
Out[15]: <__main__.Network at 0xb3124a6c>

In [16]: net1.addresses
Out[16]: ['10.1.1.193', '10.1.1.194']

In [17]: net1.network
Out[17]: '10.1.1.192/30'
```

Создаем итератор из класса Network:

```
In [12]: class Network:
    ...:     def __init__(self, network):
    ...:         self.network = network
    ...:         subnet = ipaddress.ip_network(self.network)
    ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
    ...:         self._index = 0
    ...:
    ...:     def __iter__(self):
    ...:         print('Вызываю __iter__')
    ...:         return self
    ...:
    ...:     def __next__(self):
    ...:         print('Вызываю __next__')
    ...:         if self._index < len(self.addresses):
    ...:             current_address = self.addresses[self._index]
    ...:             self._index += 1
    ...:             return current_address
    ...:         else:
    ...:             raise StopIteration
    ...:
```

Метод __iter__ в итераторе должен возвращать сам объект, поэтому в методе указано return self, а метод __next__ возвращает элементы по одному и генерирует исключение StopIteration, когда элементы закончились.

```
In [14]: net1 = Network('10.1.1.192/30')

In [15]: for ip in net1:
    ...:     print(ip)
    ...:
Вызываю __iter__
Вызываю __next__
10.1.1.193
Вызываю __next__
10.1.1.194
Вызываю __next__
```

Чаще всего, итератор это одноразовый объект и перебрав элементы, мы уже не можем это сделать второй раз:

```
In [16]: for ip in net1:
    ...:     print(ip)
    ...:
Вызываю __iter__
Вызываю __next__
```

### Создание итерируемого объекта

Очень часто классу достаточно быть итерируемым объектом и не обязательно быть итератором. Если объект будет итерируемым, его можно использовать в цикле for, функциях map, filter, sorted, enumerate и других. Также, как правило, объект проще сделать итерируемым, чем итератором.

Для того чтобы класс Network создавал итерируемые объекты, надо чтобы в классе был метод __iter__ (__next__ не нужен) и чтобы метод возвращал итератор. Так как в данном случае, Network перебирает адреса, которые находятся в списке self.addresses, самый просто вариант возвращать итератор, это вернуть `iter(self.addresses)`:

```
In [17]: class Network:
    ...:     def __init__(self, network):
    ...:         self.network = network
    ...:         subnet = ipaddress.ip_network(self.network)
    ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
    ...:
    ...:     def __iter__(self):
    ...:         return iter(self.addresses)
    ...:
```

Теперь все экземпляры класса Network будут итерируемыми объектами:

```
In [18]: net1 = Network('10.1.1.192/30')

In [19]: for ip in net1:
    ...:     print(ip)
    ...:
10.1.1.193
10.1.1.194
```

### Протокол последовательности

В самом базовом варианте, протокол последовательности (sequence) включает два метода: __len__ и __getitem__. В более полном варианте также методы: __contains__, __iter__, __reversed__, index и count. Если последовательность изменяема, добавляются еще несколько методов.

Добавим методы __len__ и __getitem__ к классу Network:

```
In [1]: class Network:
   ...:     def __init__(self, network):
   ...:         self.network = network
   ...:         subnet = ipaddress.ip_network(self.network)
   ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
...:
...:     def __iter__(self):
...:         return iter(self.addresses)
...:
...:     def __len__(self):
...:         return len(self.addresses)
...:
...:     def __getitem__(self, index):
...:         return self.addresses[index]
...:
```

Метод __len__ вызывается функцией len:

```
In [2]: net1 = Network('10.1.1.192/30')

In [3]: len(net1)
Out[3]: 2
```

А метод __getitem__ при обращении по индексу таким образом:

```
In [4]: net1[0]
Out[4]: '10.1.1.193'

In [5]: net1[1]
Out[5]: '10.1.1.194'

In [6]: net1[-1]
Out[6]: '10.1.1.194'
```

Метод __getitem__ отвечает не только обращение по индексу, но и за срезы:

```
In [7]: net1 = Network('10.1.1.192/28')

In [8]: net1[0]
Out[8]: '10.1.1.193'

In [9]: net1[3:7]
Out[9]: ['10.1.1.196', '10.1.1.197', '10.1.1.198', '10.1.1.199']

In [10]: net1[3:]
Out[10]:
['10.1.1.196',
 '10.1.1.197',
 '10.1.1.198',
 '10.1.1.199',
 '10.1.1.200',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 '10.1.1.201',
 '10.1.1.202',
 '10.1.1.203',
 '10.1.1.204',
 '10.1.1.205',
 '10.1.1.206']
```

Так как в данном случае, внутри метода __getitem__ используется список, ошибки отрабатывают корректно автоматически:

```
In [11]: net1[100]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-11-09ca84e34cb6> in <module>
----> 1 net1[100]

<ipython-input-2-bc213b4a03ca> in __getitem__(self, index)
     12
     13     def __getitem__(self, index):
---> 14         return self.addresses[index]
     15

IndexError: list index out of range

In [12]: net1['a']
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-facd90673864> in <module>
----> 1 net1['a']

<ipython-input-2-bc213b4a03ca> in __getitem__(self, index)
     12
     13     def __getitem__(self, index):
---> 14         return self.addresses[index]
     15

TypeError: list indices must be integers or slices, not str
```

Реализация остальных методов протокола последовательности вынесена в задания раздела:

- __contains__ - этот метод отвечает за проверку наличия элемента в последовательности `'10.1.1.198' in net1`. Если в объекте не определен этот метод, наличие элемента проверяется перебором элементов с помощью __iter__, а если и его нет перевором индексов с __getitem__.
- __reversed__ - используется встроенной функцией reversed. Этот метод как правило, лучше не создавать и полагаться на то, что функция reversed при отсутствии метода

__reversed__ будет использовать методы __len__ и __getitem__.

- index - возвращает индекс первого элемента, значение которого равно указаному. Работает полностью аналогично методу index в списках и кортежах.
- count - возвращает количество значений. Работает полностью аналогично методу count в списках и кортежах.

### Менеджер контекста

Менеджер контекста позволяет выполнять указанные действия в начале и в конце блока with. За работу менеджера контекста отвечают два метода:

- `__enter__(self)` - указывает, что надо сделать в начале блока with. Значение, которое возвращает метод, присваивается переменной после as.
- `__exit__(self, exc_type, exc_value, traceback)` - указывает, что надо сделать в конце блока with или при его прерывании. Если внутри блока возникло исключение, exc_type, exc_value, traceback будут содержать информацию об исключении, если исключения не было, они будут равны None.

Примеры использования менеджера контекста:

- открытие/закрытие файла
- открытие/закрытие сессии SSH/Telnet
- работа с транзакциями в БД

Класс CiscoSSH использует paramiko для подключения к оборудованию:

```
class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self.ssh = client.invoke_shell()
        self.ssh.send('enable\n')
        self.ssh.send(enable + '\n')
        if disable_paging:
            self.ssh.send('terminal length 0\n')
        time.sleep(1)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        self.ssh.recv(1000)

    def send_show_command(self, command):
        self.ssh.send(command + '\n')
        time.sleep(2)
        result = self.ssh.recv(5000).decode('ascii')
        return result
```

Пример использования класса:

```
In [9]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [10]: r1.send_show_command('sh clock')
Out[10]: 'sh clock\r\n*12:58:47.523 UTC Sun Jul 28 2019\r\nR1#'

In [11]: r1.send_show_command('sh ip int br')
Out[11]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method Status
↪            Protocol\r\nEthernet0/0              192.168.100.1   YES NVRAM  up
↪            up      \r\nEthernet0/1              192.168.200.1   YES NVRAM  up
↪            up      \r\nEthernet0/2              19.1.1.1        YES NVRAM  up
↪            up      \r\nEthernet0/3              192.168.230.1   YES NVRAM  up
↪            up      \r\nLoopback0                4.4.4.4         YES NVRAM  up
↪            up      \r\nLoopback90               90.1.1.1        YES manual up
↪            up      \r\nR1#'
```

Для того чтобы класс поддерживал работу в менеджере контекста, надо добавить методы __enter__ и __exit__:

```
class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        print('Метод __init__')
        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self.ssh = client.invoke_shell()
        self.ssh.send('enable\n')
        self.ssh.send(enable + '\n')
        if disable_paging:
            self.ssh.send('terminal length 0\n')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        time.sleep(1)
        self.ssh.recv(1000)

    def __enter__(self):
        print('Метод __enter__')
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print('Метод __exit__')
        self.ssh.close()

    def send_show_command(self, command):
        self.ssh.send(command + '\n')
        time.sleep(2)
        result = self.ssh.recv(5000).decode('ascii')
        return result
```

Пример использования класса в менеджере контекста:

```
In [14]: with CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco') as r1:
    ...:     print(r1.send_show_command('sh clock'))
    ...:
Метод __init__
Метод __enter__
sh clock
*13:05:50.677 UTC Sun Jul 28 2019
R1#
Метод __exit__
```

Даже если внутри блока возникнет исключение, метод __exit__ выполняется:

```
In [18]: with CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco') as r1:
    ...:     result = r1.send_show_command('sh clock')
    ...:     result / 2
    ...:
Метод __init__
Метод __enter__
Метод __exit__
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-b9ff1fa74be2> in <module>
      1 with CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco') as r1:
      2     result = r1.send_show_command('sh clock')
----> 3     result / 2
      4
```

(continues on next page)

(продолжение с предыдущей страницы)

```
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

# 11. Classmethod, staticmethod, property

В Python есть ряд полезных встроенных декораторов, которые позволяют менять поведение методов класса.

## Декоратор property

Python позволяет создавать и изменять переменные экземпляров:

```
In [1]: class Robot:
   ...:     def __init__(self, name):
   ...:         self.name = name
   ...:

In [2]: bb8 = Robot('BB-8')

In [3]: bb8.name
Out[3]: 'BB-8'

In [4]: bb8.name = 'R2D2'

In [5]: bb8.name
Out[5]: 'R2D2'
```

Однако иногда нужно сделать так чтобы при изменении/установке значения переменной, проверялся ее тип или диапазон значений, также иногда необходимо сделать переменную неизменяемой и сделать ее доступной только для чтения. В некоторых языках программирования для этого используются методы get и set, например:

```
In [9]: class IPAddress:
   ...:     def __init__(self, address, mask):
   ...:         self._address = address
   ...:         self._mask = int(mask)
   ...:
   ...:     def set_mask(self, mask):
   ...:         if not isinstance(mask, int):
   ...:             raise TypeError("Маска должна быть числом")
   ...:         if not mask in range(8, 32):
   ...:             raise ValueError("Маска должна быть в диапазоне от 8 до 32")
   ...:         self._mask = mask
   ...:
   ...:     def get_mask(self):
   ...:         return self._mask
   ...:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [10]: ip1 = IPAddress('10.1.1.1', 24)

In [12]: ip1.set_mask(23)

In [13]: ip1.get_mask()
Out[13]: 23
```

По сравнению со стандартным синтаксисом обращения к атрибутам, этот вариант выглядит очень громоздко. В Python есть более компактный вариант сделать то же самое - property.

Property как правило, используется как декоратор метода и превращает метод в переменную экземпляра с точки зрения пользователя класса.

Пример создания property:

```
In [14]: class IPAddress:
    ...:     def __init__(self, address, mask):
    ...:         self._address = address
    ...:         self._mask = int(mask)
    ...:
    ...:     @property
    ...:     def mask(self):
    ...:         return self._mask
    ...:
```

Теперь можно обращаться к mask как к обычной переменной:

```
In [15]: ip1 = IPAddress('10.1.1.1', 24)

In [16]: ip1.mask
Out[16]: 24
```

Один из плюсов property - переменная становится доступной только для чтения:

```
In [17]: ip1.mask = 30
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-17-e153170a5893> in <module>
----> 1 ip1.mask = 30

AttributeError: can't set attribute
```

Также property позволяет добавлять метод setter, который будет отвечать за изменение значения переменной и, так как это тоже метод, позволяет включить логику с проверкой или динамическим вычислением значения.

```
In [19]: class IPAddress:
    ...:     def __init__(self, address, mask):
    ...:         self._address = address
    ...:         self._mask = int(mask)
    ...:
    ...:     @property
    ...:     def mask(self):
    ...:         return self._mask
    ...:
    ...:     @mask.setter
    ...:     def mask(self, mask):
    ...:         if not isinstance(mask, int):
    ...:             raise TypeError("Маска должна быть числом")
    ...:         if not mask in range(8, 32):
    ...:             raise ValueError("Маска должна быть в диапазоне от 8 до 32")
    ...:         self._mask = mask
    ...:

In [20]: ip1 = IPAddress('10.1.1.1', 24)

In [21]: ip1.mask
Out[21]: 24

In [23]: ip1.mask = 30

In [24]: ip1.mask = 320
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-8573933afac9> in <module>
----> 1 ip1.mask = 320

<ipython-input-19-d0e571cd5e2b> in mask(self, mask)
     13             raise TypeError("Маска должна быть числом")
     14         if not mask in range(8, 32):
---> 15             raise ValueError("Маска должна быть в диапазоне от 8 до 32")
     16         self._mask = mask
     17

ValueError: Маска должна быть в диапазоне от 8 до 32
```

Пример использования property для динамического получения значения:

```
from base_ssh import BaseSSH
import time


```

(continues on next page)

(продолжение с предыдущей страницы)

```
class CiscoSSH(BaseSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)
        self._cfg = None

    @property
    def cfg(self):
        if not self._cfg:
            self._cfg = self.send_show_command('sh run')
        return self._cfg
```

При обращении к переменной cfg первый раз, на оборудовании выполняется команда sh run и записывается в переменную self._cfg, второй раз значение просто берется из переменной:

```
In [6]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [7]: r1.cfg # тут возникает пауза
Out[7]: 'sh run\r\nBuilding configuration...\r\n\r\nCurrent configuration : 2286 bytes\r\
↪n!\r\nversion 15.2\r\n...'

In [8]: r1.cfg
Out[8]: 'sh run\r\nBuilding configuration...\r\n\r\nCurrent configuration : 2286 bytes\r\
↪n!\r\nversion 15.2\r\n...'
```

В этом примере property используется для создания переменной, которая отвечает за чтение/изменение основного IP-адреса:

```
import re
import time
from base_ssh import BaseSSH


class CiscoSSH(BaseSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)
        self._mgmt_ip = None

    def config_mode(self):
        self._ssh.send('conf t\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def exit_config_mode(self):
        self._ssh.send('end\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def send_config_commands(self, commands):
        result = self.config_mode()
        result += super().send_config_commands(commands)
        result += self.exit_config_mode()
        return result

    @property
    def mgmt_ip(self):
        if not self._mgmt_ip:
            loopback0 = self.send_show_command('sh run interface lo0')
            self._mgmt_ip = re.search('ip address (\S+) ', loopback0).group(1)
        return self._mgmt_ip

    @mgmt_ip.setter
    def mgmt_ip(self, new_ip):
        if self.mgmt_ip != new_ip:
            self.send_config_commands([f'interface lo0',
                                       f'ip address {new_ip} 255.255.255.255'])
            self._mgmt_ip = new_ip
```

Теперь при чтении переменной mgmt_ip считывается конфиг или читается переменная _mgmt_ip, а при записи адрес перенастраивается на оборудовании:

```
In [19]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [22]: r1.mgmt_ip
Out[22]: '4.4.4.4'

In [23]: r1.mgmt_ip = '10.4.4.4'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [24]: r1.mgmt_ip
Out[24]: '10.4.4.4'

In [27]: print(r1.send_show_command('sh run interface lo0'))
sh run interface lo0
Building configuration...

Current configuration : 64 bytes
!
interface Loopback0
 ip address 10.4.4.4 255.255.255.255
end

R1#
```

### Варианты создания property

Стандартный вариант применения property без setter

```
class Book:
    def __init__(self, title, price, quantity):
        self.title = title
        self.price = price
        self.quantity = quantity

    # метод, который декорирован property становится getter'ом
    @property
    def total(self):
        print('getter')
        return self.price * self.quantity
```

Стандартный вариант применения property с setter

```
class Book:
    def __init__(self, title, price, quantity):
        self.title = title
        self.price = price
        self.quantity = quantity

    # total остается атрибутом только для чтения
    @property
    def total(self):
        return round(self.price * self.quantity, 2)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    # a price доступен для чтения и записи
    @property # этот метод превращается в getter
    def price(self):
        print('price getter')
        return self._price

    # при записи делается проверка значения
    @price.setter
    def price(self, value):
        print('price setter')
        if not isinstance(value, (int, float)):
            raise TypeError('Значение должно быть числом')
        if not value >= 0:
            raise ValueError('Значение должно быть положительным')
        self._price = float(value)
```

Декораторы с явным setter

```
class Book:
    def __init__(self, title, price, quantity):
        self.title = title
        self.price = price
        self.quantity = quantity

    # создаем пустую property для total
    total = property()

    @total.getter
    def total(self):
        return round(self.price * self.quantity, 2)

    # создаем пустую property для price
    price = property()

    # позже указываем getter
    @price.getter
    def price(self):
        print('price getter')
        return self._price

    @price.setter
    def price(self, value):
        print('price setter')
        if not isinstance(value, (int, float)):
            raise TypeError('Значение должно быть числом')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        if not value >= 0:
            raise ValueError('Значение должно быть положительным')
        self._price = float(value)
```

property без декораторов

```
class Book:
    def __init__(self, title, price, quantity):
        self.title = title
        self.price = price
        self.quantity = quantity

    def _get_total(self):
        return round(self.price * self.quantity, 2)

    def _get_price(self):
        print('price getter')
        return self._price

    def _set_price(self, value):
        print('price setter')
        if not isinstance(value, (int, float)):
            raise TypeError('Значение должно быть числом')
        if not value >= 0:
            raise ValueError('Значение должно быть положительным')
        self._price = float(value)

    total = property(_get_total)
    price = property(_get_price, _set_price)
```

Второй вариант property без декораторов

```
class Book:
    def __init__(self, title, price, quantity):
        self.title = title
        self.price = price
        self.quantity = quantity

    def _get_total(self):
        return round(self.price * self.quantity, 2)

    def _get_price(self):
        print('price getter')
        return self._price

    def _set_price(self, value):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        print('price setter')
        if not isinstance(value, (int, float)):
            raise TypeError('Значение должно быть числом')
        if not value >= 0:
            raise ValueError('Значение должно быть положительным')
        self._price = float(value)

    total = property()
    total = total.getter(_get_total)

    price = property()
    price = price.getter(_get_price)
    price = price.setter(_set_price)
```

## Декоратор classmethod

Иногда нужно реализовать несколько способов создания экземпляра, при этом в Python можно создавать только один метод __init__. Конечно, можно реализовать все варианты в одном __init__, но при этом часто параметры __init__ становятся или слишком общими, или их слишком много.

Существует другой вариант решения проблемы - создать альтернативный конструктор с помощью декоратора classmethod.

Пример альтернативного конструктора в стандартной библиотеке:

```
In [25]: r1 = {
    ...: 'hostname': 'R1',
    ...: 'OS': 'IOS',
    ...: 'Vendor': 'Cisco'
    ...: }

In [28]: dict.fromkeys(['hostname', 'os', 'vendor'])
Out[28]: {'hostname': None, 'os': None, 'vendor': None}

In [29]: dict.fromkeys(['hostname', 'os', 'vendor'], '')
Out[29]: {'hostname': '', 'os': '', 'vendor': ''}
```

```
import time
from textfsm import clitable
from base_ssh import BaseSSH

class CiscoSSH(BaseSSH):
    def __init__(self, ip, username, password, enable_password,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)
        self._mgmt_ip = None

    @classmethod
    def default_params(cls, ip):
        params = {
            'ip': ip,
            'username': 'cisco',
            'password': 'cisco',
            'enable_password': 'cisco'}
        return cls(**params)
```

```
In [8]: r1 = CiscoSSH.default_params('192.168.100.1')

In [9]: r1.send_show_command('sh clock')
Out[9]: '*16:38:01.883 UTC Sun Jan 28 2018'
```

## Декоратор staticmethod

Статический метод - это метод, который не привязан к состоянию экземпляра или класса. Для создания статического метода используется декоратор staticmethod.

Преимущества использования staticmethod:

- Это подсказка для тех, кто читает код, которая указывает на то, что метод не зависит от состояния экземпляра класса.

Большинству методов для работы нужна ссылка на экземпляр, поэтому как первый аргумент используется self. Однако иногда бывают методы, которые никак не связаны с экземпляром и зависят только от аргументов. Как правило, в таком случае можно даже вынести метод из класса и сделать его функцией. Если же метод логически связан с работой класса, но работает одинаково независимо от состояния экземпляров, метод декорируют декоратором staticmethod, чтобы указать это явно.

```
import time
from textfsm import clitable
from base_ssh import BaseSSH
```

(continues on next page)

(продолжение с предыдущей страницы)

```
class CiscoSSH(BaseSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)
        self._mgmt_ip = None

    @staticmethod
    def _parse_show(command, command_output,
                    index_file='index', templates='templates'):
        attributes = {'Command': command,
                      'Vendor': 'cisco_ios'}
        cli_table = clitable.CliTable(index_file, templates)
        cli_table.ParseCmd(command_output, attributes)
        return [dict(zip(cli_table.header, row)) for row in cli_table]

    def send_show_command(self, command, parse=True):
        command_output = super().send_show_command(command)
        if not parse:
            return command_output
        return self._parse_show(command, command_output)
```

```
In [6]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [7]: r1.send_show_command('sh ip int br')
Out[7]:
[{'intf': 'Ethernet0/0',
  'address': '192.168.100.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/1',
  'address': '192.168.200.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/2',
  'address': '19.1.1.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/3',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 'address': '192.168.230.1',
 'status': 'up',
 'protocol': 'up'},
{'intf': 'Loopback0',
 'address': '10.4.4.4',
 'status': 'up',
 'protocol': 'up'},
{'intf': 'Loopback90',
 'address': '90.1.1.1',
 'status': 'up',
 'protocol': 'up'}]
```

# 12. Наследование

## Терминология

### Интерфейс/Протокол (Interface/Protocol)

Интерфейс - набор атрибутов и методов, которые реализуют определенное поведение. Примеры: итератор, менеджер контекста, последовательность.

### Наследование (Inheritance)

Наследование - концепция ООП, которая возволяет дочернему классу использовать компоненты (методы и переменные) родительского класса.

Как правило, для наследования есть две основные причины:

- создание подтипа (interface inheritance)
- наследование для использования кода

В Python синтаксис наследования используется с абстрактными классами для наследования интерфейса/протокола. Кроме того, синтаксис наследования используется с Mixin.

Принцип подстановки Барбары Лисков

### Агрегирование (Aggregation)

Агрегация (агрегирование по ссылке) — отношение «часть-целое» между двумя равноправными объектами, когда один объект (контейнер) имеет ссылку на другой объект. Оба объекта могут существовать независимо: если контейнер будет уничтожен, то его содержимое — нет.

### Композиция (Composition)

Композиция (агрегирование по значению) — более строгий вариант агрегирования, когда включаемый объект может существовать только как часть контейнера. Если контейнер будет уничтожен, то и включённый объект тоже будет уничтожен.

```
from jinja2 import Environment, FileSystemLoader

env = Environment(loader=FileSystemLoader('templates'))
template = env.get_template('router_template.txt')
```

### Полиморфизм (Polymorphism)

Как правило, различают два варианта полиморфизма:

1. способность функции/метода обрабатывать данные разных типов
2. один интерфейс - много реализаций. Пример: одно и то же имя метода в разных классах

### Метакласс (Metaclass)

Метакласс - это класс экземпляры которого тоже являются классами.

### Абстрактный класс (abstract class)

Абстрактный класс - базовый класс, который не предполагает создания экземпляров. Как правило, содержит абстрактные методы - методы, которые обязательно должны быть созданы в дочерних классах.

В Python абстрактные классы часто используются для создания интерфейса/протокола.

### Примесь (Mixin)

Примесь это класс, который реализует какое-то одно ограниченное поведение (метод).

В Python примеси делаются с помощью классов. Так как в Python нет отдельного типа для примесей, классам-примесям принято давать имена заканчивающиеся на Mixin.

## Основы наследования

Наследование позволяет создавать новые классы на основе существующих. Различают дочерний и родительские классы: дочерний класс наследует родительский. При наследовании, дочерний класс наследует все методы и атрибуты родительского класса.

Пример класса BaseSSH, который выполняет подключение по SSH с помощью paramiko:

```
import paramiko
import time


class BaseSSH:
    def __init__(self, ip, username, password):
        self.ip = ip
        self.username = username
        self.password = password
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        self._MAX_READ = 10000

        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self._ssh = client.invoke_shell()
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self._ssh.close()

    def close(self):
        self._ssh.close()

    def send_show_command(self, command):
        self._ssh.send(command + '\n')
        time.sleep(2)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def send_config_commands(self, commands):
        if isinstance(commands, str):
            commands = [commands]
        for command in commands:
            self._ssh.send(command + '\n')
            time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result
```

Этот класс будет использоваться как основа для классов, которые отвечают за подключение к устройствам разных вендоров. Например, класс CiscoSSH будет отвечать за подключение к устройствам Cisco будет наследовать класс BaseSSH.

Синтаксис наследования:

```
class CiscoSSH(BaseSSH):
    pass
```

После этого в классе CiscoSSH доступны все методы и атрибуты класса BaseSSH:

```
In [3]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco')

In [4]: r1.ip
Out[4]: '192.168.100.1'

In [5]: r1._MAX_READ
Out[5]: 10000

In [6]: r1.send_show_command('sh ip int br')
Out[6]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method Status
↪           Protocol\r\nEthernet0/0                192.168.100.1   YES NVRAM  up
↪           up      \r\nEthernet0/1                192.168.200.1   YES NVRAM  up
↪           up      \r\nEthernet0/2                19.1.1.1        YES NVRAM  up
↪           up      \r\nEthernet0/3                192.168.230.1   YES NVRAM  up
↪           up      \r\nLoopback0                  4.4.4.4         YES NVRAM  up
↪           up      \r\nLoopback33                 3.3.3.3         YES manual up
↪           up      \r\nLoopback90                 90.1.1.1        YES manual up
↪           up      \r\nR1#'



In [7]: r1.send_show_command('enable')
Out[7]: 'enable\r\nPassword: '

In [8]: r1.send_show_command('cisco')
Out[8]: '\r\nR1#'

In [9]: r1.send_config_commands(['conf t', 'int loopback 33',
   ...:                          'ip address 3.3.3.3 255.255.255.255', 'end'])
Out[9]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.\r\
↪nR1(config)#int loopback 33\r\nR1(config-if)#ip address 3.3.3.3 255.255.255.255\r\
↪nR1(config-if)#end\r\nR1#'
```

После наследования всех методов родительского класса, дочерний класс может:

- оставить их без изменения
- полностью переписать их
- дополнить метод
- добавить свои методы

В классе CiscoSSH надо создать метод __init__ и добавить к нему параметры:

- enable_password - пароль enable
- disable_paging - отвечает за включение/отключение постраничного вывода команд

Метод __init__ можно создать полностью с нуля, однако базовая логика подключения по SSH будет одинаковая в BaseSSH и CiscoSSH, поэтому лучше добавить необходимые параметры, а для подключения, вызвать метод __init__ у класса BaseSSH. Есть несколько вариантов вызова родительского метода, например, все эти варианты вызовут метод send_show_command родительского класса из дочернего класса CiscoSSH:

```
command_result = BaseSSH.send_show_command(self, command)
command_result = super(CiscoSSH, self).send_show_command(command)
command_result = super().send_show_command(command)
```

Первый вариант BaseSSH.send_show_command явно указывает имя родительского класса - это самый понятный вариант для восприятия, однако его минус в том, что при смене имени родительского класса, имя надо будет менять во всех местах, где вызывались методы родительского класса. Также у этого варианта есть минусы, при использовании множественного наследования. Второй и третий вариант по сути равнозначны, но третий короче, поэтому мы будем использовать его.

Класс CiscoSSH с методом __init__:

```
class CiscoSSH(BaseSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)
```

Метод __init__ в классе CiscoSSH добавил параметры enable_password и disable_paging, и использует их соответственно для перехода в режим enable и отключения постраничного вывода. Пример подключения:

```
In [10]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [11]: r1.send_show_command('sh clock')
Out[11]: 'sh clock\r\n*11:30:50.280 UTC Mon Aug 5 2019\r\nR1#'
```

Теперь при подключении также выполняется переход в режим enable и по умолчанию отключен paging, так что можно попробовать выполнить длинную команду, например sh run.

Еще один метод, который стоит доработать - метод send_config_commands: так как класс CiscoSSH предназначен для работы с Cisco, можно в добавить в него переход в конфигурационный режим перед командами и выход после.

```
class CiscoSSH(BaseSSH):
    def __init__(self, ip, username, password, enable_password=None,
                 disable_paging=True):
        super().__init__(ip, username, password)
        if enable_password
            self._ssh.send('enable\n')
            self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def config_mode(self):
        self._ssh.send('conf t\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def exit_config_mode(self):
        self._ssh.send('end\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def send_config_commands(self, commands):
        result = self.config_mode()
        result += super().send_config_commands(commands)
        result += self.exit_config_mode()
        return result
```

Пример использования метода send_config_commands:

```
In [12]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [13]: r1.send_config_commands(['interface loopback 33',
    ...:                          'ip address 3.3.3.3 255.255.255.255'])
Out[13]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.\r\
↪nR1(config)#interface loopback 33\r\nR1(config-if)#ip address 3.3.3.3 255.255.255.255\r\
↪nR1(config-if)#end\r\nR1#'
```

# Исключения

## Встроенные исключения

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
```

(continues on next page)

(продолжение с предыдущей страницы)

```
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

### Пользовательские исключения

Пользователь может создать свое исключение, которое описывает работу своего приложения/скрипта. Для этого, необходимо наследовать класс Exception.

```
class Error(Exception):
    """Свое исключение для модуля."""
    pass
```

## Множественное наследование

В Python дочерний класс может наследовать несколько родительских.

**class A:**

**def __init__(self):**
print(„A.__init__“)

**class B:**

**def __init__(self):**
print(„B.__init__“)

**class C(A, B):**

**def __init__(self):**
print(„C.__init__“)

## Abstract Base Classes (ABC)

Иногда, при создании иерархии классов, необходимо чтобы ряд классов поддерживал одинаковый интерфейс, например, одинаковый набор методов. Частично эту задачу можно решить с помощью наследования, однако далеко не всегда дочерним классам подойдет реализация метода из родительского класса.

Абстрактный класс - это класс в котором созданы абстрактные методы - методы, которые обязательно должны присутствовать в дочерних классах. Создавть экзепмляр абстрактного класса нельзя, его надо наследовать и уже у дочернего класса можно создать экземпляр. При этом экземпляр дочернего класса можно создать только в том случае, если у дочернего класса есть реализация всех абстрактных методов.

Базовый пример абстрактного класса:

```
In [1]: import abc

In [2]: class Parent(abc.ABC):
   ...:     @abc.abstractmethod
   ...:     def get_info(self, parameter):
   ...:         """Get parameter info"""
   ...:
   ...:     @abc.abstractmethod
   ...:     def set_info(self, parameter, value):
   ...:         """Set parameter to value"""
   ...:
```

Нельзя создать экземпляр класса Parent:

```
In [3]: p1 = Parent()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-0b1eb161869e> in <module>
----> 1 p1 = Parent()

TypeError: Can't instantiate abstract class Parent with abstract methods get_info, set_
↪info
```

Дочерний класс обязательно должен добавить свою реализацию абстрактных методов, иначе при создании экземпляра возникнет исключение:

```
In [4]: class Child(Parent):
   ...:     pass
   ...:

In [5]: c1 = Child()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-07f905b6a091> in <module>
----> 1 c1 = Child()

TypeError: Can't instantiate abstract class Child with abstract methods get_info, set_info
```

После создания методов get_info и set_info, можно создать экземпляр класса Child:

```
In [6]: class Child(Parent):
   ...:     def __init__(self):
   ...:         self._parameters = {}
   ...:
   ...:     def get_info(self, parameter):
   ...:         return self._parameters.get(parameter)
   ...:
   ...:     def set_info(self, parameter, value):
   ...:         self._parameters[parameter] = value
   ...:         return self._parameters
   ...:

In [7]: c1 = Child()

In [8]: c1.set_info('name', 'BB-8')
Out[8]: {'name': 'BB-8'}
```

Пример абстрактного класса BaseSSH:

```
import paramiko
import time
import abc


class BaseSSH(abc.ABC):
    def __init__(self, ip, username, password):
        self.ip = ip
        self.username = username
        self.password = password
        self._MAX_READ = 10000

        client = paramiko.SSHClient()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self._ssh = client.invoke_shell()
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self._ssh.close()

    def close(self):
        self._ssh.close()

    @abc.abstractmethod
    def send_command(self, command):
        """Send command and get command output"""

    @abc.abstractmethod
    def send_config_commands(self, commands):
        """Send configuration command(s)"""
```

Соответственно в дочерних классах обязательно должны быть методы send_command и send_config_commands:

```
class CiscoSSH(BaseSSH):
    device_type = 'cisco_ios'
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def send_command(self, command):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        self._ssh.send(command + '\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def config_mode(self):
        self._ssh.send('conf t\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def exit_config_mode(self):
        self._ssh.send('end\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def send_config_commands(self, commands):
        result = self.config_mode()
        result += super().send_config_commands(commands)
        result += self.exit_config_mode()
        return result
```

### Абстрактные классы в стандартной библиотеке Python

В стандартной библиотеке Python есть несколько готовых абстрактных классов, которые можно использовать для наследования или проверки типа объекта. Большая часть классов находится в `collections.abc` и часть из них показана в таблице ниже.

Полный перечень классов `collections.abc` доступен в документации

| ABC | Наследует | Абстрактные методы | Mixin методы |
|---|---|---|---|
| Container | | __contains__ | |
| Hashable | | __hash__ | |
| Iterable | | __iter__ | |
| Iterator | Iterable | __next__ | __iter__ |
| Reversible | Iterable | __reversed__ | |
| Generator | Iterator | send, throw | close, __iter__, __next__ |
| Sized | | __len__ | |
| Callable | | __call__ | |
| Collection | Sized, Iterable, Container | __contains__, __iter__, __len__ | |
| Sequence | Reversible, Collection | __getitem__, __len__ | __contains__, __iter__, __reversed__, index, count |

Один из вариантов использования абстрактных классов collections.abc - это проверка того поддерживает ли объект протокол. Например, проверить является ли объект итерируемым можно таким образом:

```
In [1]: from collections.abc import Iterable

In [2]: l1 = [1, 2, 3]

In [3]: s1 = 'line'

In [4]: n1 = 5

In [5]: isinstance(l1, Iterable)
Out[5]: True

In [6]: isinstance(s1, Iterable)
Out[6]: True

In [7]: isinstance(n1, Iterable)
Out[7]: False
```

Второй вариант использования классов collections.abc - наследование классов для поддержки определенного интерфейса. Например, повторим пример с классом Network из подраздела «Протокол последовательности», но теперь с наследованием класса Sequence:

```
In [1]: from collections.abc import Sequence
In [2]: import ipaddress

In [3]: class Network(Sequence):
   ...:     def __init__(self, network):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...:         self.network = network
   ...:         subnet = ipaddress.ip_network(self.network)
   ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
   ...:
```

Пробуем создать экземпляр класса Network:

```
In [4]: net1 = Network('10.1.1.192/29')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-9c2ed79a8719> in <module>
----> 1 net1 = Network('10.1.1.192/29')

TypeError: Cant instantiate abstract class Network with abstract methods __getitem__, __
↪len__
```

Исключение указывает, что экземпляр не может быть создан, так как в классе Network нет методов __getitem__ и __len__. Это методы, которые созданы как абстрактные и в таблице выше указаны в соответствующем столбце. Добавляем эти методы в класс Network:

```
In [5]: class Network(Sequence):
   ...:     def __init__(self, network):
   ...:         self.network = network
   ...:         subnet = ipaddress.ip_network(self.network)
   ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
   ...:
   ...:     def __getitem__(self, index):
   ...:         return self.addresses[index]
   ...:
   ...:     def __len__(self):
   ...:         return len(self.addresses)
   ...:
```

Теперь можно создать экземпляр класса Network и экземпляр поддерживает обращение по индексу, а также работает функция len:

```
In [6]: net1 = Network('10.1.1.192/29')

In [7]: net1.addresses
Out[7]:
['10.1.1.193',
 '10.1.1.194',
 '10.1.1.195',
 '10.1.1.196',
 '10.1.1.197',
 '10.1.1.198']
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [8]: len(net1)
Out[8]: 6

In [9]: net1[4]
Out[9]: '10.1.1.197'
```

Кроме того, за счет наследования Sequence, в классе появились методы __contains__, __iter__, __reversed__, index и count:

```
In [10]: '10.1.1.193' in net1
Out[10]: True

In [11]: i = iter(net1)

In [12]: next(i)
Out[12]: '10.1.1.193'

In [13]: next(i)
Out[13]: '10.1.1.194'


In [14]: list(reversed(net1))
Out[14]:
['10.1.1.198',
 '10.1.1.197',
 '10.1.1.196',
 '10.1.1.195',
 '10.1.1.194',
 '10.1.1.193']

In [15]: net1.index('10.1.1.195')
Out[15]: 2

In [16]: net1.count('10.1.1.197')
Out[16]: 1
```

## Mixin классы

Mixin классы - это классы у которых нет данных, но есть методы. Mixin используются для добавления одних и тех же методов в разные классы.

В Python примеси делаются с помощью классов. Так как в Python нет отдельного типа для примесей, классам-примесям принято давать имена заканчивающиеся на Mixin.

С одной стороны, то же самое можно сделать с помощью наследования обычных классов, но не всегда те методы, которые нужны в разных дочерних классах, имеют смысл в родительском.

```
import time
import inspect
from base_ssh import BaseSSH


class SourceCodeMixin:
    @property
    def sourcecode(self):
        return inspect.getsource(self.__class__)


class AttributesMixin:
    @property
    def attributes(self):
        # data attributes
        for name, value in self.__dict__.items():
            print(f"{name:25}{str(value):<20}")
        # methods
        for name, value in self.__class__.__dict__.items():
            if not name.startswith('__'):
                print(f"{name:25}{str(value):<20}")
```

```
In [1]: from mixin_example import CiscoSSH

In [2]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [3]: r1.attributes
ip                        192.168.100.1
username                  cisco
password                  cisco
_MAX_READ                 10000
_ssh                      <paramiko.Channel 0 (open) window=8161 -> <paramiko.Transport at␣
↪0xb36a412c (cipher aes128-cbc, 128 bits) (active; 1 open channel(s))>>
config_mode               <function CiscoSSH.config_mode at 0xb36a15cc>
exit_config_mode          <function CiscoSSH.exit_config_mode at 0xb36a1614>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
send_config_commands     <function CiscoSSH.send_config_commands at 0xb36a165c>


In [4]: print(r1.sourcecode)
class CiscoSSH(SourceCodeMixin, AttributesMixin, BaseSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def config_mode(self):
        self._ssh.send('conf t\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def exit_config_mode(self):
        self._ssh.send('end\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def send_config_commands(self, commands):
        result = self.config_mode()
        result += super().send_config_commands(commands)
        result += self.exit_config_mode()
        return result
```

## Дополнительные материалы

### Наследование

- Real Python Inheritance and Composition: A Python OOP Guide
- python-course.eu Inheritance
- python-course.eu Multiple Inheritance
- mro
- Guido van Rossum Method Resolution Order

- The wonders of cooperative inheritance, or using super in Python 3

### super

- Python docs super
- Статья Python’s super() considered super!
- Raymond Hettinger - Super considered super! - PyCon 2015
- Real Python Supercharge Your Classes With Python super()

### Исключения

- Python Built-in Exceptions
- User-defined Exceptions
- raise

### Mixin

- What is a mixin, and why are they useful?

Примеры mixin:

- class contextlib.ContextDecorator

### ABC

- python docs abc
- python docs collections.abc
- python-course.eu ABC

### SOLID

- SOLID wikipedia
- Open-closed principle
- Liskov substitution principle

# 13. Data classes

## Data classes

Часто в Python необходимо создавать классы в которых указаны только несколько переменных. При этом, для реализации таких операций как сравнение экземпляров класса требуется создать несколько специальных методов, добавить сюда строковое представление объекта и для создания довольно простого класса, требуется много кода.

---

**Примечание:** Data classes это новый функционал, он входит в стандартную бибилиотеку начиная с Python 3.7. Для предыдущих версий надо ставить отдельный модуль dataclasses или использовать сторонний типа модуля attr.

---

Модуль dataclasses предоставляет декоратор dataclass с помощью которого можно существенно упростить создание классов:

```
In [9]: dataclass?
Signature:
dataclass(
    _cls=None,
    *,
    init=True,
    repr=True,
    eq=True,
    order=False,
    unsafe_hash=False,
    frozen=False,
)
Docstring:
Returns the same class as was passed in, with dunder methods
added based on the fields defined in the class.

Examines PEP 526 __annotations__ to determine fields.

If init is true, an __init__() method is added to the class. If
repr is true, a __repr__() method is added. If order is true, rich
comparison dunder methods are added. If unsafe_hash is true, a
__hash__() method function is added. If frozen is true, fields may
not be assigned to after instance creation.
File:      /usr/local/lib/python3.7/dataclasses.py
Type:      function
```

Пример класса IPAddress:

```
class IPAddress:
    def __init__(self, ip, mask):
        self._ip = ip
        self._mask = mask

    def __repr__(self):
        return f"IPAddress({self.ip}/{self.mask})"
```

И соответствующего класса созданного с помощью dataclass:

```
In [11]: @dataclass
    ...: class IPAddress:
    ...:     ip: str
    ...:     mask: int
    ...:

In [12]: ip1 = IPAddress('10.1.1.1', 28)

In [13]: ip1
Out[13]: IPAddress(ip='10.1.1.1', mask=28)
```

Для создания класса данных используется аннотация типов. Декоратор dataclass использует указанные переменные и дополнительные настройки для создания атрибутов для экземпляров класса, а также методов __init__, __repr__ и других.

Все переменные, которые определены на уровне класса, по умолчанию, будут прописаны в методе __init__ и будут ожидаться как аргументы при создании экземпляра.

---

**Примечание:** Типы указанные в определении класса не преобразуют атрибуты и не проверяют реальный тип данных аргументов.

---

### Метод __post_init__

Метод __post_init__ позволяет добавлять дополнительную логику работы с переменными экземпляра. Например, можно проверить тип данных или сделать дополнительные вычисления:

```
@dataclass
class IPAddress:
    ip: str
    mask: int

    def __post_init__(self):
        if not isinstance(self.mask, int):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            self.mask = int(self.mask)


In [46]: ip1 = IPAddress('10.10.1.1', '24')

In [47]: ip1.mask
Out[47]: 24
```

### Параметры order и frozen

При декорировании класса можно указать дополнительные параметры:

- frozen - контролирует можно ли менять значения переменных
- order - если равен True, добавляет к классу методы __lt__, __le__, __gt__, __ge__

Если параметр order равен True, экземпляры класса можно сравнивать и упорядочивать:

```
@dataclass(order=True)
class IPAddress:
    ip: str
    mask: int


In [12]: ip1 = IPAddress('10.1.1.1', 28)

In [14]: ip1 == ip2
Out[14]: False

In [15]: ip1 < ip2
Out[15]: True
```

В данном случае, при сравнении и сортировке экземпляров класса возникает проблема из-за лексикографической сортировки - экземпляры сортируются не так как хотелось бы:

```
In [24]: ip1 = IPAddress('10.10.1.1', 24)

In [25]: ip2 = IPAddress('10.2.1.1', 24)

In [26]: ip2 > ip1
Out[26]: True

In [27]: ip_list = [ip1, ip2]

In [28]: ip_list
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Out[28]: [IPAddress(ip='10.10.1.1', mask=24), IPAddress(ip='10.2.1.1', mask=24)]

In [30]: sorted(ip_list)
Out[30]: [IPAddress(ip='10.10.1.1', mask=24), IPAddress(ip='10.2.1.1', mask=24)]
```

### Функция field

Функция field позволяет указывать параметры работы с отдельными переменными.

```
dataclasses.field(*, default=MISSING, default_factory=MISSING,
                  repr=True, hash=None, init=True, compare=True, metadata=None)
```

Например, с помощью field можно указать, что какая-то переменная не должна отображаться в __repr__:

```
@dataclass
class User:
    username: str
    password: str = field(repr=False)


In [49]: user1 = User('John', '12345')

In [50]: user1
Out[50]: User(username='John')
```

Все переменные, которые определены на уровне класса, по умолчанию, будут прописаны в методе __init__ и будут ожидаться как аргументы при создании экземпляра. Иногда в классе могут присутствовать переменные, которые вычисляются на основании аргументов __init__, а не передаются как аргументы. В этом случае, можно воспользоваться параметром init в field и вычислить значение динамически в__post_init__:

```
@dataclass
class Book:
    title: str
    price: int
    quantity: int
    total: int = field(init=False)

    def __post_init__(self):
        self.total = self.price * self.quantity


In [52]: book = Book('Good Omens', 35, 5)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [53]: book.total
Out[53]: 175

In [54]: book
Out[54]: Book(title='Good Omens', price=35, quantity=5, total=175)
```

Функция field также поможет исправить ситуацию с сортировкой в классе IPAddress. Указав compare=False при создании переменной, можно исключить ее из сравнения и сортировки. Также в классе добавлена дополнительная переменная _ip, которая содержит IP-адрес в виде числа. Для этой переменной init=False, так как это значение не надо передавать при создании экземпляра, и repr=False, так как переменная не должна отображаться в строковом представлении:

```
@dataclass(order=True)
class IPAddress:
    ip: str = field(compare=False)
    _ip: int = field(init=False, repr=False)
    mask: int

    def __post_init__(self):
        self._ip = int(ipaddress.ip_address(self.ip))


In [40]: ip1 = IPAddress('10.10.1.1', 24)

In [41]: ip2 = IPAddress('10.2.1.1', 24)

In [42]: ip_list = [ip1, ip2]

In [43]: sorted(ip_list)
Out[43]: [IPAddress(ip='10.2.1.1', mask=24), IPAddress(ip='10.10.1.1', mask=24)]

In [44]: ip1 > ip2
Out[44]: True
```

### Функции asdict, astuple, replace

```
In [2]: from dataclasses import asdict, astuple, replace, dataclass

In [3]: @dataclass(order=True, frozen=True)
   ...: class IPAddress:
   ...:     ip: str
   ...:     mask: int = 24
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...:

In [4]: ip1 = IPAddress('10.1.1.1', 28)

In [5]: asdict(ip1)
Out[5]: {'ip': '10.1.1.1', 'mask': 28}

In [6]: astuple(ip1)
Out[6]: ('10.1.1.1', 28)

In [8]: replace(ip1, mask=24)
Out[8]: IPAddress(ip='10.1.1.1', mask=24)

In [9]: ip3 = replace(ip1, mask=24)

In [10]: ip3
Out[10]: IPAddress(ip='10.1.1.1', mask=24)
```

### Работа с property

```
@dataclass
class Book:
    title: str
    price: float
    _price: float = field(init=False, repr=False)
    quantity: int = 0 # TypeError: non-default argument 'quantity' follows default␣
↪argument

    @property
    def total(self):
        return round(self.price * self.quantity, 2)

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        if not isinstance(value, (int, float)):
            raise TypeError('Значение должно быть числом')
        if not value >= 0:
            raise ValueError('Значение должно быть положительным')
        self._price = float(value)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [79]: b1 = Book('Good Omens', 35, 5)

In [80]: b1.price
Out[80]: 35.0

In [81]: b1.total
Out[81]: 175.0

In [82]: b1.price = 30

In [83]: b1.total
Out[83]: 150.0
```

## Дополнительные материалы

Документация:

- Data Classes

Полезные статьи:

- Python dataclasses: A revolution
- Reconciling Dataclasses And Properties In Python

Видео:

- Raymond Hettinger - Dataclasses: The code generator to end all code generators - PyCon 2018

Дополнительные возможности:

- pydantic for full type validation for dataclasses
- Модуль attrs похож на dataclasses, но у него больше возможностей

# 4

# IV. Генераторы

# 14. Генераторы

Генератор - функция, которая позволяет легко создавать свои итераторы. В отличии от обычных функций, генератор не просто возвращает значение и завершает работу, а возвращает итератор, который отдает элементы по одному.

## Создание генератора

Функция-генератор - это функция, в которой присутствует ключевое слово yield. При вызове, эта функция возвращает объект генератор.

Обычная функция завершает работу если:

- встретилось выражение return
- закончился код функции (это срабатывает как выражение `return None`)
- возникло исключение

После выполнения функции, управление возвращается и программа выполняется дальше. Все аргументы, которые передавались в функцию, локальные переменные, все это теряется. Остается только результат, который вернула функция.

Функция может возвращать список элементов, несколько объектов или возвращать разные результаты, в зависимости от аргументов, но она всегда возвращает какой-то один результат.

С точки зрения синтаксиса, генератор выглядит как обычная функция, но, вместо return, используется оператор `yield`. Каждый раз, когда внутри функции встречается yield, генератор приостанавливается и возвращает значение. При следующем запросе, генератор начинает работать с того же места, где он завершил работу в прошлый раз. Так как yield не завершает работу генератора, он может использоваться несколько раз.

## Генератор

Генераторы - это специальный класс функций, который позволяет легко создавать свои итераторы. В отличии от обычных функций, генератор не просто возвращает значение и завершает работу, а возвращает итератор, который отдает элементы по одному.

Более корректное определение: функция-генератор - это функция, в которой присутствует ключевое слово yield. При вызове, эта функция возвращает объект генератор. Так как и сама функция и объект, который она возвращает, называется генератор, возникает путанница, о чем идет речь. В документации Python очень часто объект генератор называется итератором. Поэтому тут я тоже буду называть возвращенный объект итератором, а функцию - генератором.

Обычная функция завершает работу если:

- встретилось выражение return
- закончился код функции (это срабатывает как выражение return None)
- возникло исключение

После выполнения функции, управление возвращается и программа выполняется дальше. Все аргументы, которые передавались в функцию, локальные переменные, все это теряется. Остается только результат, который вернула функция. Функция может возвращать список элементов, несколько объектов или возвращать разные результаты, в зависимости от аргументов, но она всегда возвращает какой-то один результат.

Генератор же генерирует значения. При этом, значения возвращаются по запросу и после возврата одного значения, выполнение функции-генератора приостанавливается до запроса следующего значения. Между запросами генератор сохраняет свое состояние.

С точки зрения синтаксиса, генератор выглядит как обычная функция, но, вместо return, используется оператор yield.

Каждый раз, когда внутри функции встречается yield, генератор приостанавливается и возвращает значение. При следующем запросе, генератор начинает работать с того же места, где он завершил работу в прошлый раз.

### Базовый пример

Рассмотрим простой пример генератора:

```
In [1]: def generate_nums(number):
   ...:     print('Start of generation')
   ...:     yield number
   ...:     print('Next number')
   ...:     yield number+1
   ...:     print('The end')
   ...:
```

Если вызвать генератор и присвоить результат в переменную, его код еще не будет выполняться:

```
In [3]: result = generate_nums(100)
```

Теперь в переменной result находится итератор:

```
In [4]: result
Out[4]: <generator object generate_nums at 0xb5788e9c>
```

Раз result это итератор, можно вызвать функцию next, чтобы получить значение:

```
In [5]: next(result)
Start of generation

Out[5]: 100
```

После первого вызова next, генератор выполнил все строки до первого yield. В данном случае, отобразилась строка „Start of generation“. Затем yield вернул значение - число 100.

Второй вызов next:

```
In [6]: next(result)
Next number

Out[6]: 101
```

Выполнение продолжилось с предыдущего места - выведена строка „Next number“ и вернулось значение 101.

Следующий next:

```
In [7]: next(result)
The end
---------------------------------------------------------------
StopIteration             Traceback (most recent call last)
<ipython-input-7-1b214ba10814> in <module>()
----> 1 next(result)

StopIteration:
```

Так как в result находится итератор, когда элементы заканчиваются, он генерирует исключение StopIteration, но, до этого, вывелась строка „The end“.

Раз функция-генератор возвращает итератор, его можно использовать в цикле:

```
In [8]: for num in generate_nums(100):
   ...:     print('Number:', num)
   ...:
Start of generation
Number: 100
Next number
Number: 101
The end
```

### Обычная функция и аналогичный генератор

С помощью генераторов зачастую можно написать ту же функцию с меньшим количеством промежуточных переменных. Например, функцию такого вида:

```
In [14]: def work_with_items(items):
    ...:     result = []
    ...:     for item in items:
    ...:         result.append('Changed {}'.format(item))
    ...:     return result
    ...:

In [15]: for i in work_with_items(range(10)):
    ...:     print(i)
    ...:
Changed 0
Changed 1
Changed 2
Changed 3
Changed 4
Changed 5
Changed 6
Changed 7
Changed 8
Changed 9
```

Можно заменить таким генератором:

```
In [16]: def yield_items(items):
    ...:     for item in items:
    ...:         yield 'Changed {}'.format(item)
    ...:

In [17]: for i in yield_items(range(10)):
    ...:     print(i)
    ...:
Changed 0
Changed 1
Changed 2
Changed 3
Changed 4
Changed 5
Changed 6
Changed 7
Changed 8
Changed 9
```

При этом, генератор yield_items возвращает элементы по одному, а функция work_with_items - собирает их в список, а потом возвращает. Если количество элементов небольшое, это не существенно, но при обработке больших объемов данных, лучше работать с элементами по одному.

При этом, в любой момент, если действительно нужно получить все элементы, например, в виде списка, это можно сделать применив функцию list:

```
In [20]: result =  yield_items(range(10))

In [21]: result
Out[21]: <generator object yield_items at 0xb579053c>

In [22]: list(result)
Out[22]:
['Changed 0',
 'Changed 1',
 'Changed 2',
 'Changed 3',
 'Changed 4',
 'Changed 5',
 'Changed 6',
 'Changed 7',
 'Changed 8',
 'Changed 9']
```

### Использование генератора, при работе с файлами

Например, при обработке большого log-файла, лучше обрабатывать его построчно, не выгружая все содержимое в память.

Допустим, нам нужно часто фильтровать определенные строки из файла. Например, надо получить только строки, которые соответствуют регулярному выражению. Конечно, можно каждый раз это делать в процессе обработки строк. Но можно вынести подобную функциональность и в отдельную функцию.

Но только, в случае обычной функции, придется опять возвращать список или подобный объект. А, если файл очень большой, то, скорее всего, придется отказаться от этой затеи.

Однако, если использовать генератор, файл будет обрабатываться построчно. Это может быть, например, такой генератор:

```
In [3]: import re

In [5]: def filter_lines(filename, regex):
   ...:     with open(filename) as f:
   ...:         for line in f:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...:             if re.search(regex, line):
   ...:                 yield line.rstrip()
   ...:
```

Генератор проходится по указанному файлу и отдает те строки, которые совпали с регулярным выражением.

Пример использования:

```
In [7]: for line in filter_lines('config_r1.txt', '^interface'):
   ...:     print(line)
   ...:
interface Loopback0
interface Tunnel0
interface Ethernet0/0
interface Ethernet0/1
interface Ethernet0/2
interface Ethernet0/3
interface Ethernet0/3.100
interface Ethernet1/0
```

## Пример использования генератора для обработки вывода sh cdp neighbors detail

Генераторы могут использоваться не только в том случае, когда надо возвращать элементы по одному.

Например, генератор get_cdp_neighbor читает файл с выводом sh cdp neighbor detail и выдает вывод частями, по одному соседу:

```
def get_one_neighbor(filename):
    with open(filename) as f:
        line = ''
        while True:
            while not 'Device ID' in line:
                line = f.readline()
            neighbor = line
            for line in f:
                if '----------' in line:
                    break
                neighbor += line
            yield neighbor
            line = f.readline()
            if not line:
                return
```

Полный скрипт выглядит таким образом (файл parse_cdp_neighbors.py):

```
import re
from pprint import pprint


def get_one_neighbor(filename):
    with open(filename) as f:
        line = ''
        while True:
            while not 'Device ID' in line:
                line = f.readline()
            neighbor = line
            for line in f:
                if '----------' in line:
                    break
                neighbor += line
            yield neighbor
            line = f.readline()
            if not line:
                return


def parse_neighbor(output):
    regex = (
        r'Device ID: (\S+).+?'
        r' IP address: (?P<ip>\S+).+?'
        r'Platform: (?P<platform>\S+ \S+), .+?'
        r', Version (?P<ios>\S+),')

    result = {}
    match = re.search(regex, output, re.DOTALL)
    if match:
        device = match.group(1)
        result[device] = match.groupdict()
    return result

if __name__ == "__main__":
    data = get_one_neighbor('sh_cdp_neighbors_detail.txt')
    for n in data:
        pprint(parse_neighbor(n), width=120)
```

Так как генератор get_cdp_neighbor выдает каждый раз вывод про одного соседа, можно проходиться по результату в цикле и передавать каждый вывод функции parse_cdp. И конечно же, полученный результат тоже можно не собирать в один большой словарь, а передавать куда-то дальше на обработку или запись.

Результат выполнения:

```
$ python parse_cdp_neighbors.py
{'SW2': {'ios': '12.2(55)SE9', 'ip': '10.1.1.2', 'platform': 'cisco WS-C2960-8TC-L'}}
{'R1': {'ios': '12.4(24)T1', 'ip': '10.1.1.1', 'platform': 'Cisco 3825'}}
{'R2': {'ios': '15.2(2)T1', 'ip': '10.2.2.2', 'platform': 'Cisco 2911'}}
{'R3': {'ios': '15.2(2)T1', 'ip': '10.3.3.3', 'platform': 'Cisco 2911'}}
```

## generator expression (генераторное выражение)

Генераторное выражение использует такой же синтаксис, как list comprehentions, но возвращает итератор, а не список.

Генераторное выражение выглядит точно так же, как list comprehentions, но используются круглые скобки:

```
In [1]: genexpr = (x**2 for x in range(10000))

In [2]: genexpr
Out[2]: <generator object <genexpr> at 0xb571ec8c>

In [3]: next(genexpr)
Out[3]: 0

In [4]: next(genexpr)
Out[4]: 1

In [5]: next(genexpr)
Out[5]: 4
```

Обратите внимание, что это не tuple comprehentions, а генераторное выражение.

Оно полезно в том случае, когда надо работать с большим итерируемым объектом или бесконечным итератором.

## Дополнительные материалы

Документация:

- Iterator types
- Functional Programming HOWTO

Статьи:

- Iterables vs. Iterators vs. Generators
- Improve Your Python: „yield“ and Generators Explained - generator and generator expressions

- Generator Tricks for Systems Programmers. David Beazley

Ответ на stackoverflow:

- Difference between Python’s Generators and Iterators
- Understanding Generators in Python
- What can you use Python generator functions for?

В книге Fluent Python этой теме посвящен 14 раздел:

- Fluent Python. Chapter 14 Iterables, Iterators, and Generators
- Примеры из книги

# 15. Модули itertools, more-itertools

В этом разделе рассматриваются два модуля: модуль из стандартной библиотеки itertools и сторонний модуль more-itertools. Оба модуля предоставляют набор функций для работы с итераторами.

Оба модуля содержат большое количество функций, поэтому в этом разделе рассматриваются только некоторые из них.

## itertools

### repeat

Функция repeat возвращает итератор, который повторяет указанный объект бесконечно или указанное количество раз:

```
itertools.repeat(object[, times])
```

Пример использования repeat для повторения команды:

```
from itertools import repeat
from concurrent.futures import ThreadPoolExecutor

import netmiko
import yaml


def send_show(device, show):
    with netmiko.ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_command(show)
        return result


with open('devices.yaml') as f:
    devices = yaml.safe_load(f)

with ThreadPoolExecutor(max_workers=3) as executor:
    result = executor.map(send_show, devices, repeat('sh clock'))
    for device, output in zip(devices, result):
        print(device['ip'], output)
```

### cycle

Функция cycle создает итератор, которые возвращает элементы итерируемого объекта по кругу:

```
itertools.cycle(iterable)
```

Пример использования cycle:

```
from itertools import cycle


spinner = it.cycle('\|/-')
for _ in range(20):
    print(f'\r{next(spinner)}', end='')
    time.sleep(0.5)
```

### count

Функция count возвращает итератор, который генерирует числа бесконечно, начиная с указанного в start и используя шаг step:

```
itertools.count(start=0, step=1)
```

Пример использования count:

```
from itertools import count


In [13]: ip_list
Out[13]:
['192.168.100.1',
 '192.168.100.2',
 '192.168.100.3',
 '192.168.100.4',
 '192.168.100.5']

In [18]: for num, ip in zip(count(1), ip_list):
    ...:     print((num, ip))
    ...:
(1, '192.168.100.1')
(2, '192.168.100.2')
(3, '192.168.100.3')
(4, '192.168.100.4')
(5, '192.168.100.5')
```

### zip_longest

Функция zip_longest работает аналогично встроенной функции zip, но не останавливается на самом коротком итерируемом объекте.

```
itertools.zip_longest(*iterables, fillvalue=None)
```

Пример использования:

```
In [20]: list(zip([1,2,3,4,5], [10,20]))
Out[20]: [(1, 10), (2, 20)]

In [21]: list(zip_longest([1,2,3,4,5], [10,20]))
Out[21]: [(1, 10), (2, 20), (3, None), (4, None), (5, None)]

In [22]: list(zip_longest([1,2,3,4,5], [10,20], fillvalue=0))
Out[22]: [(1, 10), (2, 20), (3, 0), (4, 0), (5, 0)]
```

### chain

Функция chain ожидает несколько итерируемых объектов как аргумент и возвращает единый итератор, который перебирает элементы каждого итерируемого объекта так, как будто они составляют единый объект:

```
itertools.chain(*iterables)
```

Пример использования:

```
In [4]: line = 'test'

In [5]: items = [1, 2, 3]

In [6]: mapping = {'ios': '15.4', 'vendor': 'Cisco'}

In [7]: for item in chain(line, items, mapping):
   ...:     print(item)
   ...:
t
e
s
t
1
2
3
ios
vendor
```

### compress

Функция compress позволяет фильтровать данные: она возвращает те элементы из data, которые соответствуют истинному значению в selectors:

```
itertools.compress(data, selectors)
```

Пример использования compress для фильтрации полей с ненулевым значением:

```
In [9]: headers = ['tx_packets', 'rx_packets', 'tx_bytes', 'rx_bytes', 'broadcasts']

In [10]: data = [294785, 0, 22275381, 0, 253218]

In [12]: list(compress(headers, data))
Out[12]: ['tx_packets', 'tx_bytes', 'broadcasts']

In [14]: list(compress(zip(headers, data), data))
Out[14]: [('tx_packets', 294785), ('tx_bytes', 22275381), ('broadcasts', 253218)]

In [24]: dict(compress(zip(headers, data), data))
Out[24]: {'tx_packets': 294785, 'tx_bytes': 22275381, 'broadcasts': 253218}
```

Пример фильтрации None:

```
In [25]: data2
Out[25]: [294785, 0, 22275381, None, None]

In [26]: headers
Out[26]: ['tx_packets', 'rx_packets', 'tx_bytes', 'rx_bytes', 'broadcasts']

In [27]: list(compress(zip(headers, data2), selectors=map(lambda x: x != None, data2)))
Out[27]: [('tx_packets', 294785), ('rx_packets', 0), ('tx_bytes', 22275381)]
```

### tee

Функция tee создает несколько независимых итераторов на основе исходных данных:

```
itertools.tee(iterable, n=2)
```

Пример использования:

```
In [30]: data = [1,2,3,4,5,6]

In [31]: data_iter = iter(data)

In [32]: duplicate_1, duplicate_2 = tee(data_iter)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [33]: list(duplicate_1)
Out[33]: [1, 2, 3, 4, 5, 6]

In [34]: list(duplicate_2)
Out[34]: [1, 2, 3, 4, 5, 6]
```

Важная особенность tee - исходный итератор лучше не использовать, иначе полученные итераторы начнут перебор не с начала:

```
In [35]: data_iter = iter(data)

In [36]: duplicate_1, duplicate_2 = tee(data_iter)

In [37]: next(data_iter)
Out[37]: 1

In [38]: next(data_iter)
Out[38]: 2

In [39]: list(duplicate_1)
Out[39]: [3, 4, 5, 6]

In [40]: list(duplicate_2)
Out[40]: [3, 4, 5, 6]
```

При этом перебор одной копии, не влияет на вторую:

```
In [41]: data_iter = iter(data)

In [42]: duplicate_1, duplicate_2 = tee(data_iter)

In [43]: next(duplicate_1)
Out[43]: 1

In [44]: next(duplicate_1)
Out[44]: 2

In [45]: list(duplicate_1)
Out[45]: [3, 4, 5, 6]

In [46]: list(duplicate_2)
Out[46]: [1, 2, 3, 4, 5, 6]
```

### islice

Функция islice

```
itertools.islice(iterable, stop)
itertools.islice(iterable, start, stop[, step])
```

Пример использования:

```
In [59]: list(islice(range(100), 5))
Out[59]: [0, 1, 2, 3, 4]

In [60]: list(islice(range(100), 5, 10))
Out[60]: [5, 6, 7, 8, 9]

In [61]: list(islice(range(100), 5, 10, 2))
Out[61]: [5, 7, 9]

In [62]: list(islice(range(100), 5, 20, 2))
Out[62]: [5, 7, 9, 11, 13, 15, 17, 19]

In [63]: list(islice(range(100), 5, 20, 3))
Out[63]: [5, 8, 11, 14, 17]
```

### groupby

Функция groupby

```
itertools.groupby(iterable, key=None)
```

Пример использования:

```
from pprint import pprint
from dataclasses import dataclass
import operator

@dataclass(frozen=True)
class Book:
    title: str
    author: str


In [75]: books
Out[75]:
[Book(title='1984', author='George Orwell'),
 Book(title='The Martian Chronicles', author='Ray Bradbury'),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 Book(title='The Hobbit', author='J.R.R. Tolkien'),
 Book(title='Animal Farm', author='George Orwell'),
 Book(title='Fahrenheit 451', author='Ray Bradbury'),
 Book(title='The Lord of the Rings (1-3)', author='J.R.R. Tolkien'),
 Book(title='Harry Potter and the Sorcerer's Stone', author='J.K. Rowling'),
 Book(title='To Kill a Mockingbird', author='Harper Lee')]


In [76]: list(groupby(books, operator.attrgetter('author')))
Out[76]:
[('George Orwell', <itertools._grouper at 0xb473f3ec>),
 ('Ray Bradbury', <itertools._grouper at 0xb473f12c>),
 ('J.R.R. Tolkien', <itertools._grouper at 0xb473f98c>),
 ('George Orwell', <itertools._grouper at 0xb473f7cc>),
 ('Ray Bradbury', <itertools._grouper at 0xb473f40c>),
 ('J.R.R. Tolkien', <itertools._grouper at 0xb473f74c>),
 ('J.K. Rowling', <itertools._grouper at 0xb473ffcc>),
 ('Harper Lee', <itertools._grouper at 0xb473fbec>)]


In [81]: for key, item in groupby(books, operator.attrgetter('author')):
    ...:     print(key.ljust(20), list(item))
    ...:
George Orwell        [Book(title='1984', author='George Orwell')]
Ray Bradbury         [Book(title='The Martian Chronicles', author='Ray Bradbury')]
J.R.R. Tolkien       [Book(title='The Hobbit', author='J.R.R. Tolkien')]
George Orwell        [Book(title='Animal Farm', author='George Orwell')]
Ray Bradbury         [Book(title='Fahrenheit 451', author='Ray Bradbury')]
J.R.R. Tolkien       [Book(title='The Lord of the Rings (1-3)', author='J.R.R. Tolkien')]
J.K. Rowling         [Book(title='Harry Potter and the Sorcerer's Stone', author='J.K.␣
↪Rowling')]
Harper Lee           [Book(title='To Kill a Mockingbird', author='Harper Lee')]


In [83]: sorted_books = sorted(books, key=operator.attrgetter('author'))

In [84]: sorted_books
Out[84]:
[Book(title='1984', author='George Orwell'),
 Book(title='Animal Farm', author='George Orwell'),
 Book(title='To Kill a Mockingbird', author='Harper Lee'),
 Book(title='Harry Potter and the Sorcerer's Stone', author='J.K. Rowling'),
 Book(title='The Hobbit', author='J.R.R. Tolkien'),
 Book(title='The Lord of the Rings (1-3)', author='J.R.R. Tolkien'),
 Book(title='The Martian Chronicles', author='Ray Bradbury'),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 Book(title='Fahrenheit 451', author='Ray Bradbury')]

In [85]: for key, item in groupby(sorted_books, operator.attrgetter('author')):
    ...:     print(key.ljust(20), list(item))
    ...:
George Orwell        [Book(title='1984', author='George Orwell'), Book(title='Animal Farm
↪', author='George Orwell')]
Harper Lee            [Book(title='To Kill a Mockingbird', author='Harper Lee')]
J.K. Rowling          [Book(title='Harry Potter and the Sorcerer's Stone', author='J.K.␣
↪Rowling')]
J.R.R. Tolkien        [Book(title='The Hobbit', author='J.R.R. Tolkien'), Book(title='The␣
↪Lord of the Rings (1-3)', author='J.R.R. Tolkien')]
Ray Bradbury          [Book(title='The Martian Chronicles', author='Ray Bradbury'),␣
↪Book(title='Fahrenheit 451', author='Ray Bradbury')]

In [86]: books_by_author = {}

In [87]: for key, item in groupby(sorted_books, operator.attrgetter('author')):
    ...:     books_by_author[key] = list(item)
    ...:

In [90]: pprint(books_by_author)
{'George Orwell': [Book(title='1984', author='George Orwell'),
                   Book(title='Animal Farm', author='George Orwell')],
 'Harper Lee': [Book(title='To Kill a Mockingbird', author='Harper Lee')],
 'J.K. Rowling': [Book(title='Harry Potter and the Sorcerer's Stone', author='J.K. Rowling
↪')],
 'J.R.R. Tolkien': [Book(title='The Hobbit', author='J.R.R. Tolkien'),
                    Book(title='The Lord of the Rings (1-3)', author='J.R.R. Tolkien')],
 'Ray Bradbury': [Book(title='The Martian Chronicles', author='Ray Bradbury'),
                  Book(title='Fahrenheit 451', author='Ray Bradbury')]}
```

### dropwhile и takewhile

Функция dropwhile ожидает как аргументы функцию, которая возвращает True или False, в зависимости от условия, и итерируемый объект. Функция dropwhile отбрасывает элементы итерируемого объекта до тех пор, пока функция переданная как аргумент возвращает True. Как только dropwhile встречает False, он возвращает итератор с оставшимися объектами.

```
In [1]: from itertools import dropwhile

In [2]: list(dropwhile(lambda x: x < 5, [0,2,3,5,10,2,3]))
Out[2]: [5, 10, 2, 3]
```

В данном случае, как только функция dropwhile дошла до числа, которое больше или равно

пяти, она вернула все оставшиеся числа. При этом, даже если далее есть числа, которые меньше 5, функция уже не проверяет их.

Функция takewhile - противоположность функции dropwhile: она возвращает итератор с теми элементами, которые соответствуют условию, до первого ложного условия:

```
In [3]: from itertools import takewhile

In [4]: list(takewhile(lambda x: x < 5, [0,2,3,5,10,2,3]))
Out[4]: [0, 2, 3]
```

Пример использования takewhile и dropwhile

```
def get_cdp_neighbor(sh_cdp_neighbor_detail):
    with open(sh_cdp_neighbor_detail) as f:
        while True:
            begin = dropwhile(lambda x: not 'Device ID' in x, f)
            lines = takewhile(lambda y: not '-----' in y, begin)
            neighbor = ''.join(lines)
            if not neighbor:
                return
            yield neighbor
```

Файл parse_cdp_file.py:

```
import re
from pprint import pprint
from itertools import dropwhile, takewhile


def get_cdp_neighbor(sh_cdp_neighbor_detail):
    with open(sh_cdp_neighbor_detail) as f:
        while True:
            f = dropwhile(lambda x: not 'Device ID' in x, f)
            lines = takewhile(lambda y: not '-----' in y, f)
            neighbor = ''.join(lines)
            if not neighbor:
                return None
            yield neighbor


def parse_cdp_neighbor(output):
    regex = ('Device ID: (\S+)\n.*?'
             ' +IP address: (?P<ip>\S+).+?'
             'Platform: (?P<platform>\S+ \S+),.+?'
             'Version (?P<ios>\S+),')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    result = {}
    match = re.search(regex, output, re.DOTALL)
    if match:
        device = match.group(1)
        result[device] = match.groupdict()
    return result


def parse_cdp_output(filename):
    result = get_cdp_neighbor(filename)
    all_cdp = {}
    for neighbor in result:
        all_cdp.update(parse_cdp_neighbor(neighbor))
    return all_cdp


if __name__ == "__main__":
    filename = 'sh_cdp_neighbors_detail.txt'
    pprint(parse_cdp_output(filename), width=120)
```

Результат:

```
$ python parse_cdp_file.py
{'R1': {'ios': '12.4(24)T1', 'ip': '10.1.1.1', 'platform': 'Cisco 3825'},
 'R2': {'ios': '15.2(2)T1', 'ip': '10.2.2.2', 'platform': 'Cisco 2911'},
 'R3': {'ios': '15.2(2)T1', 'ip': '10.3.3.3', 'platform': 'Cisco 2911'},
 'SW2': {'ios': '12.2(55)SE9', 'ip': '10.1.1.2', 'platform': 'cisco WS-C2960-8TC-L'}}
```

## more-itertools

### Группировка

### chunked

Разбивает итерируемый объект на списки указанной длины:

```
more_itertools.chunked(iterable, n)
```

Пример:

```
In [6]: list(more_itertools.chunked(data, 2))
Out[6]: [[1, 2], [3, 4], [5, 6], [7]]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [7]: list(more_itertools.chunked(data, 3))
Out[7]: [[1, 2, 3], [4, 5, 6], [7]]
```

### divide

Разбивает итерируемый объект на n частей:

```
more_itertools.divide(n, iterable)
```

Пример:

```
In [25]: data
Out[25]: [1, 2, 3, 4, 5, 6, 7]

In [26]: g1, g2, g3 = more_itertools.divide(3, data)

In [27]: list(g1)
Out[27]: [1, 2, 3]

In [28]: list(g2)
Out[28]: [4, 5]

In [29]: list(g3)
Out[29]: [6, 7]
```

### split_at

Генерирует списки элементов из итерируемого объекта, где каждый список разделен тем значением, для которого pred возвращает True (разедлитель не включен).

```
more_itertools.split_at(iterable, pred)
```

Пример:

```
import time

def file_gen(filename):
    with open(filename) as f:
        for idx, line in enumerate(f):
            print(idx)
            yield line

f = file_gen('sh_cdp_neighbors_detail.txt')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
for items in more_itertools.split_at(f, lambda x: '------' in x):
    print(items)
    time.sleep(2)
```

### unzip

Выполняет операцию противоположную zip:

```
more_itertools.unzip(iterable)
```

Пример:

```
In [2]: data = [('status', '*'),
   ...:         ('network', '1.23.78.0'),
   ...:         ('netmask', '24'),
   ...:         ('nexthop', '200.219.145.45'),
   ...:         ('metric', 'NA'),
   ...:         ('locprf', 'NA'),
   ...:         ('weight', '0'),
   ...:         ('path', '28135 18881 3549 6453 4755 45528'),
   ...:         ('origin', 'i')]

In [3]: headers, values = more_itertools.unzip(data)

In [4]: list(headers)
Out[4]:
['status',
 'network',
 'netmask',
 'nexthop',
 'metric',
 'locprf',
 'weight',
 'path',
 'origin']

In [5]: list(values)
Out[5]:
['*',
 '1.23.78.0',
 '24',
 '200.219.145.45',
 'NA',
 'NA',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 '0',
 '28135 18881 3549 6453 4755 45528',
 'i']
```

### grouper

```
more_itertools.grouper(iterable, n, fillvalue=None)
```

Пример:

```
In [6]: data = [1, 2, 3, 4, 5, 6, 7]

In [8]: list(more_itertools.grouper(data, 3, 0))
Out[8]: [(1, 2, 3), (4, 5, 6), (7, 0, 0)]
```

### partition

```
more_itertools.partition(pred, iterable)
```

Пример:

```
In [10]: data = [1, 2, 'a', 'b', 5, 'c', 7]

In [15]: is_false, is_true = more_itertools.partition(lambda x: str(x).isdigit(), data)

In [16]: list(is_false)
Out[16]: ['a', 'b', 'c']

In [17]: list(is_true)
Out[17]: [1, 2, 5, 7]
```

### spy

```
more_itertools.spy(iterable, n=1)
```

Пример

```
In [19]: def file_gen(filename):
    ...:     with open(filename) as f:
    ...:         for idx, line in enumerate(f):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ...:             print(idx)
    ...:             yield line
    ...:

In [20]: f = file_gen('sh_cdp_neighbors_detail.txt')

In [21]: f
Out[21]: <generator object file_gen at 0xb28bd4ec>

In [23]: first, f = more_itertools.spy(f)
0

In [24]: first
Out[24]: ['SW1#show cdp neighbors detail\n']

In [25]: f
Out[25]: <itertools.chain at 0xb38c184c>

In [26]: next(f)
Out[26]: 'SW1#show cdp neighbors detail\n'
```

### windowed

```
more_itertools.windowed(seq, n, fillvalue=None, step=1)
```

```
In [33]: windows = more_itertools.windowed(f, 5)

In [34]: for win in windows:
    ...:     print(win)
    ...:
0
1
2
3
4
('SW1#show cdp neighbors detail\n', '------------------------\n', 'Device ID: SW2\n',
↪'Entry address(es):\n', '  IP address: 10.1.1.2\n')
5
('------------------------\n', 'Device ID: SW2\n', 'Entry address(es):\n', '  IP␣
↪address: 10.1.1.2\n', 'Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP\n')
6
('Device ID: SW2\n', 'Entry address(es):\n', '  IP address: 10.1.1.2\n', 'Platform: cisco␣
↪WS-C2960-8TC-L,  Capabilities: Switch IGMP\n', 'Interface: GigabitEthernet1/0/16,  Port␣
↪ID (outgoing port): GigabitEthernet0/1\n')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
7
('Entry address(es):\n', '  IP address: 10.1.1.2\n', 'Platform: cisco WS-C2960-8TC-L, ␣
↪Capabilities: Switch IGMP\n', 'Interface: GigabitEthernet1/0/16,  Port ID (outgoing␣
↪port): GigabitEthernet0/1\n', 'Holdtime : 164 sec\n')
8
('  IP address: 10.1.1.2\n', 'Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP\n
↪', 'Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1\n',
↪'Holdtime : 164 sec\n', '\n')
```

### collapse

```
more_itertools.collapse(iterable, base_type=None, levels=None)
```

Пример

```
In [37]: iterable = [(1, 2), ([3, 4], [[5], [6]])]

In [38]: list(more_itertools.collapse(iterable))
Out[38]: [1, 2, 3, 4, 5, 6]
```

### Агрегирование значений

### first и last

```
more_itertools.first(iterable[, default])
more_itertools.last(iterable[, default])
```

```
In [42]: data = [1, 2, 'a', 'b', 5, 'c', 7]

In [43]: more_itertools.first(data)
Out[43]: 1

In [44]: more_itertools.last(data)
Out[44]: 7
```

### all_equal

```
more_itertools.all_equal(iterable)
```

```
In [46]: more_itertools.all_equal([1, 1, 1])
Out[46]: True

In [47]: more_itertools.all_equal([1, 2, 1])
Out[47]: False
```

# 5

# V. Основы asyncio

Разделы 16-18 переехали в отдельную книгу Основы asyncio для сетевых инженеров

Если вы делаете задания из курса «Advanced Python для сетевых инженеров», то соответствие разделов такое:

- 17 раздел заданий - 1, 2 разделы книги
- 18 раздел заданий - 3, 4 и 5 разделы книги

# 6

# VI. Дополнительная информация

В этом разделе собрана информация, которая не вошла в основные разделы книги, но которая, тем не менее, может быть полезна.

## Использование памяти

```
import resource
from base_ssh import BaseSSH

memory_start = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss

ip_list = ['192.168.100.1', '192.168.100.2', '192.168.100.3']*5
sessions = [BaseSSH(ip, 'cisco', 'cisco') for ip in ip_list]

memory_end = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss

print('Start ', memory_start)
print('End   ', memory_end)
print(sessions)
```

# Дополнительные темы по ООП

## Дескриптор

```
class IPAddress:

    def __init__(self, ip, mask):
        self._ip = ip
        self._mask = mask

    @property
    def ip(self):
        return self._ip

    @ip.setter
    def ip(self, value):
        if not isinstance(value, str):
            raise TypeError('Wrong data type, expected str')
        self._ip = value

    @property
    def mask(self):
        return self._mask

    @mask.setter
    def mask(self, value):
        if not isinstance(value, int):
            raise TypeError('Wrong data type, expected int')
        self._mask = mask
```

```
class Integer:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Wrong data type, expected int')
        instance.__dict__[self.name] = value

class String:
    def __init__(self, name):
        self.name = name
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def __get__(self, instance, cls):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError('Wrong data type, expected str')
        instance.__dict__[self.name] = value
```

Дескриптор обязательно должен быть указан на уровне класса:

```
class IPAddress:
    mask = Integer('mask')
    ip = String('ip')

    def __init__(self, ip, mask):
        self._ip = ip
        self._mask = mask


In [90]: ip1 = IPAddress('10.1.1.1', 28)

In [96]: ip1.mask = '24'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-96-247b3f37d10f> in <module>
----> 1 ip1.mask = '24'

<ipython-input-93-5812cdd26ed1> in __set__(self, instance, value)
      8         def __set__(self, instance, value):
      9             if not isinstance(value, int):
---> 10                raise TypeError('Wrong data type, expected int')
     11            instance.__dict__[self.name] = value
     12

TypeError: Wrong data type, expected int

In [97]: ip1.ip = 142
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-97-24102e80dc3a> in <module>
----> 1 ip1.ip = 142

<ipython-input-93-5812cdd26ed1> in __set__(self, instance, value)
     20         def __set__(self, instance, value):
     21             if not isinstance(value, str):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---> 22                 raise TypeError('Wrong data type, expected str')
     23             instance.__dict__[self.name] = value

TypeError: Wrong data type, expected str
```

Оптимизированный вариант:

```
class Typed:
    attr_type = object

    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, self.attr_type):
            raise TypeError(f'Wrong data type, expected {self.attr_type}')
        instance.__dict__[self.name] = value

class Integer(Typed):
    attr_type = int

class String(Typed):
    attr_type = str
```

### Замыкания вместо дескриптора для проверки типа

```
In [74]: def typed(name, attr_type):
    ...:     value = '_' + name
    ...:
    ...:     @property
    ...:     def attribute(self):
    ...:         return getattr(self, value)
    ...:
    ...:     @attribute.setter
    ...:     def attribute(self, new_value):
    ...:         if not isinstance(new_value, attr_type):
    ...:             raise TypeError(f'Wrong data type, expected {attr_type}')
    ...:         self.value = new_value
    ...:
    ...:     return attribute
    ...:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [75]: class IPAddress:
    ...:     ip = typed('ip', str)
    ...:     mask = typed('mask', int)
    ...:
    ...:     def __init__(self, ip, mask):
    ...:         self.ip = ip
    ...:         self.mask = mask
    ...:

In [76]: ip1 = IPAddress('10.1.1.1', 28)

In [77]: ip1.mask = '24'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-77-247b3f37d10f> in <module>
----> 1 ip1.mask = '24'

<ipython-input-74-4348b0de06dc> in attribute(self, new_value)
      9     def attribute(self, new_value):
     10         if not isinstance(new_value, attr_type):
---> 11             raise TypeError(f'Wrong data type, expected {attr_type}')
     12         setattr(self, value, new_value)
     13

TypeError: Wrong data type, expected <class 'int'>

In [80]: ip1.mask?
Type:        property
String form: <property object at 0xb4203aa4>
Docstring:   <no docstring>
```

### Дополнительные материалы

- Invoking Descriptors
- Descriptor HowTo Guide

## Метаклассы

```
import paramiko
import time


CLASS_MAPPER_BASE = {}

class Base(type):
    def __init__(cls, clsname, bases, methods):
        super().__init__(clsname, bases, methods)
        if hasattr(cls, 'device_type'):
            CLASS_MAPPER_BASE[cls.device_type] = cls


class BaseSSH(metaclass=Base):
    def __init__(self, ip, username, password):
        self.ip = ip
        self.username = username
        self.password = password
        self._MAX_READ = 10000

        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self._ssh = client.invoke_shell()
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self._ssh.close()

    def close(self):
        self._ssh.close()

    def send_show_command(self, command):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        self._ssh.send(command + '\n')
        time.sleep(2)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def send_config_commands(self, commands):
        if isinstance(commands, str):
            commands = [commands]
        for command in commands:
            self._ssh.send(command + '\n')
            time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result


class CiscoSSH(BaseSSH):
    device_type = 'cisco_ios'
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def config_mode(self):
        self._ssh.send('conf t\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def exit_config_mode(self):
        self._ssh.send('end\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def send_config_commands(self, commands):
        result = self.config_mode()
        result += super().send_config_commands(commands)
        result += self.exit_config_mode()
        return result
```

(continues on next page)

(продолжение с предыдущей страницы)

```
class JuniperSSH(BaseSSH):
    device_type = 'juniper'
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        pass
```

## Атрибут __slots__

Пример кода из модуля ipaddress

```
class _IPAddressBase:

    """The mother class."""

    __slots__ = ()

    @property
    def exploded(self):
        """Return the longhand version of the IP address as a string."""
        return self._explode_shorthand_ip_string()

    @property
    def compressed(self):
        """Return the shorthand version of the IP address as a string."""
        return str(self)


@functools.total_ordering
class _BaseAddress(_IPAddressBase):

    """A generic IP object.
    This IP class contains the version independent methods which are
    used by single IP addresses.
    """

    __slots__ = ()

    def __int__(self):
        return self._ip

    def __eq__(self, other):
        try:
            return (self._ip == other._ip
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                    and self._version == other._version)
        except AttributeError:
            return NotImplemented

    def __lt__(self, other):
        if not isinstance(other, _BaseAddress):
            return NotImplemented
        if self._version != other._version:
            raise TypeError('%s and %s are not of the same version' % (
                             self, other))
        if self._ip != other._ip:
            return self._ip < other._ip
        return False

class _BaseV4:

    """Base IPv4 object.
    The following methods are used by IPv4 objects in both single IP
    addresses and networks.
    """

    __slots__ = ()
    _version = 4
    # Equivalent to 255.255.255.255 or 32 bits of 1's.
    _ALL_ONES = (2**IPV4LENGTH) - 1
    _DECIMAL_DIGITS = frozenset('0123456789')

    # the valid octets for host and netmasks. only useful for IPv4.
    _valid_mask_octets = frozenset({255, 254, 252, 248, 240, 224, 192, 128, 0})

    _max_prefixlen = IPV4LENGTH
    # There are only a handful of valid v4 netmasks, so we cache them all
    # when constructed (see _make_netmask()).
    _netmask_cache = {}

    def _explode_shorthand_ip_string(self):
        return str(self)


class IPv4Address(_BaseV4, _BaseAddress):

    """Represent and manipulate single IPv4 Addresses."""

    __slots__ = ('_ip', '__weakref__')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def __init__(self, address):

        """
        Args:
            address: A string or integer representing the IP
              Additionally, an integer can be passed, so
              IPv4Address('192.0.2.1') == IPv4Address(3221225985).
              or, more generally
              IPv4Address(int(IPv4Address('192.0.2.1'))) ==
                IPv4Address('192.0.2.1')
        Raises:
            AddressValueError: If ipaddress isn't a valid IPv4 address.
        """
```

# Collections

- namedtuple - factory функция для создания подклассов кортежа с именованными атрибутами
- deque - контейнер похожий на список с быстрыми добавляениями и удалениями с двух сторон
- ChainMap - класс похожий на словарь для создания единого интерфейса для доступа к нескольким словарям
- Counter - подкласс словаря для подсчета хешируемых объектов
- OrderedDict
- defaultdict
- UserDict, UserList, UserString

## Временная сложность алгоритма

---

**Примечание:** Source

---

### Список

| Operation | Average Case |
|---|---|
| Copy | O(n) |
| Append | O(1) |
| Pop last | O(1) |
| Pop intermediate | O(n) |
| Insert | O(n) |
| Get Item | O(1) |
| Set Item | O(1) |
| Delete Item | O(n) |
| Iteration | O(n) |
| Get Slice | O(k) |
| Del Slice | O(n) |
| Set Slice | O(k+n) |
| Extend | O(k) |
| Sort | O(n log n) |
| Multiply | O(nk) |
| x in s | O(n) |
| min(s), max(s) | O(n) |
| Get Length | O(1) |

### Deque

| Operation | Average Case |
|---|---|
| copy | O(n) |
| append | O(1) |
| appendleft | O(1) |
| pop | O(1) |
| popleft | O(1) |
| extend | O(k) |
| extendleft | O(k) |
| rotate | O(k) |
| remove | O(n) |

### Dict

| Operation | Average Case |
|---|---|
| k in d | O(1) |
| copy | O(n) |
| Get Item | O(1) |
| Set Item | O(1) |
| Delete Item | O(1) |
| Iteration | O(n) |

## Создание классов с помощью namedtuple

### collections.namedtuple

Функция namedtuple позволяет создавать новые классы, которые наследуют tuple и при этом:

- доступ к атрибутам может осуществляться по имени
- доступ к элементами по индексу
- экземпляр класса является итерируемым объектом
- атрибуты неизменяемы

Именованные кортежи присваивают имена каждому элементу кортежа и код выглядит более понятным, так как вместо индексов используются имена. При этом, все возможности обычных кортежей остаются.

```
In [1]: from collections import namedtuple

In [2]: RouterClass = namedtuple('Router', ['hostname', 'ip', 'ios'])

In [3]: r1 = RouterClass('r1', '10.1.1.1', '15.4')

In [30]: r1
Out[30]: Router(hostname='r1', ip='10.1.1.1', ios='15.4')


In [18]: r1.hostname
Out[18]: 'r1'

In [19]: r1.ip
Out[19]: '10.1.1.1'

In [20]: hostname, ip, ios = r1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [21]: hostname
Out[21]: 'r1'

In [22]: ip
Out[22]: '10.1.1.1'

In [23]: ios
Out[23]: '15.4'
```

Метод _as_dict возвращает OrderedDict:

```
In [9]: r1._asdict()
Out[9]: OrderedDict([('hostname', 'r1'), ('ip', '10.1.1.1'), ('ios', '15.4')])
```

Метод _replace возвращает новый экземпляр класса, в котором заменены указанные поля:

```
In [18]: r1 = RouterClass('r1', '10.1.1.1', '15.4')

In [19]: r1
Out[19]: Router(hostname='r1', ip='10.1.1.1', ios='15.4')

In [20]: r1._replace(ip='10.2.2.2')
Out[20]: Router(hostname='r1', ip='10.2.2.2', ios='15.4')
```

Метод _make создает новый экземпляр класса из последовательности полей (это метод класса):

```
In [22]: RouterClass._make(['r3', '10.3.3.3', '15.2'])
Out[22]: Router(hostname='r3', ip='10.3.3.3', ios='15.2')

In [23]: r3 = RouterClass._make(['r3', '10.3.3.3', '15.2'])
```

Пример использования namedtuple:

```
import sqlite3
from collections import namedtuple


key = 'vlan'
value = 10
db_filename = 'dhcp_snooping.db'

keys = ['mac', 'ip', 'vlan', 'interface', 'switch']
DhcpSnoopRecord = namedtuple('DhcpSnoopRecord', keys)

```

(continues on next page)

(продолжение с предыдущей страницы)

```
conn = sqlite3.connect(db_filename)
query = 'select {} from dhcp where {} = ?'.format(','.join(keys), key)

print('-' * 40)
for row in map(DhcpSnoopRecord._make, conn.execute(query, (value,))):
    print(row.mac, row.ip, row.interface, sep='\n')
    print('-' * 40)
```

Вывод:

```
$ python get_data.py
----------------------------------------
00:09:BB:3D:D6:58
10.1.10.2
FastEthernet0/1
----------------------------------------
00:07:BC:3F:A6:50
10.1.10.6
FastEthernet0/3
----------------------------------------
```

Параметр defaults позволяет указывать значения по умолчанию:

```
In [33]: IPAddress = namedtuple('IPAddress', ['address', 'mask'], defaults=[24])

In [34]: ip1 = IPAddress('10.1.1.1', 28)

In [35]: ip1
Out[35]: IPAddress(address='10.1.1.1', mask=28)

In [36]: ip2 = IPAddress('10.2.2.2')

In [37]: ip2
Out[37]: IPAddress(address='10.2.2.2', mask=24)
```

### typing.NamedTuple

Еще один вариант создания класса с помощью именнованного кортежа - наследование класса typing.NamedTuple. Базовые особенности namedtuple сохраняются, плюс есть возможность добавлять свои методы.

```
In [9]: from typing import NamedTuple

In [10]: class IPAddress(typing.NamedTuple):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ...:     ip: str
    ...:     mask: int = 24
    ...:

In [11]: ip1 = IPAddress('10.1.1.1', 28)

In [12]: ip1
Out[12]: IPAddress(ip='10.1.1.1', mask=28)

In [13]: class IPAddress(typing.NamedTuple):
    ...:     ip: str
    ...:     mask: int = 24
    ...:
    ...:     def convert_to_bin(self):
    ...:         pass
    ...:

In [14]: ip1 = IPAddress('10.1.1.1', 28)

In [15]: ip1.convert_to_bin
Out[15]: <bound method IPAddress.convert_to_bin of IPAddress(ip='10.1.1.1', mask=28)>
```

## collections.deque

Deque поддерживает потокобезопасные, эффективные с точки зрения памяти добавления и извлечения с обеих сторон двухсторонней очереди с примерно одинаковой производительностью O(1) в любом направлении.

```
class collections.deque([iterable[, maxlen]])
```

Методы deque:

- append(x)
- appendleft(x)
- clear()
- copy()
- count(x)
- extend(iterable)
- extendleft(iterable)
- index(x[, start[, stop]])

- insert(i, x)
- pop()
- popleft()
- remove(value)
- reverse()
- rotate(n=1)
- maxlen

### append

```
In [1]: from collections import deque

In [2]: d = deque([1, 2, 3])

In [3]: d.append(4)

In [4]: d
Out[4]: deque([1, 2, 3, 4])

In [5]: d.appendleft(0)

In [6]: d
Out[6]: deque([0, 1, 2, 3, 4])
```

### pop

```
In [7]: d.pop()
Out[7]: 4

In [9]: d
Out[9]: deque([0, 1, 2, 3])

In [10]: d.popleft()
Out[10]: 0

In [11]: d
Out[11]: deque([1, 2, 3])
```

### index

```
In [12]: d[0]
Out[12]: 1

In [13]: d[-1]
Out[13]: 3
```

### rotate

```
In [14]: d.rotate(1)

In [15]: d
Out[15]: deque([3, 1, 2])

In [16]: d.rotate(-2)

In [17]: d
Out[17]: deque([2, 3, 1])
```

### maxlen

```
In [19]: d = deque([1, 2, 3, 4, 5], maxlen=5)

In [20]: d
Out[20]: deque([1, 2, 3, 4, 5])

In [21]: d.append(6)

In [22]: d
Out[22]: deque([2, 3, 4, 5, 6])

In [23]: d.appendleft(100)

In [24]: d
Out[24]: deque([100, 2, 3, 4, 5])
```

### Пример использования

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

## collections.ChainMap

ChainMap - класс похожий на словарь для создания единого интерфейса для доступа к нескольким словарям

```
class collections.ChainMap(*maps)
```

Методы:

- maps
- new_child(m=None, **kwargs)
- parents

```
In [1]: from collections import ChainMap

In [2]: r1 = {
   ...:     "host": "192.168.100.1",
   ...:     "auth_username": "cisco",
   ...:     "auth_password": "cisco",
   ...:     "auth_secondary": "cisco",
   ...:     "platform": "cisco_iosxe",
   ...:     "timeout_socket": 15,
   ...: }

In [3]: default_params = {
   ...:     "auth_strict_key": False,
   ...:     "timeout_socket": 5,
   ...:     "timeout_transport": 10,
   ...: }

In [4]: scrapli_params = ChainMap(r1, default_params)

In [5]: pprint(scrapli_params)
ChainMap({'auth_password': 'cisco',
          'auth_secondary': 'cisco',
          'auth_username': 'cisco',
          'host': '192.168.100.1',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
          'platform': 'cisco_iosxe',
          'timeout_socket': 15},
         {'auth_strict_key': False,
          'timeout_socket': 5,
          'timeout_transport': 10})

In [6]: scrapli_params["timeout_socket"]
Out[6]: 15

In [7]: scrapli_params["timeout_transport"]
Out[7]: 10

In [8]: scrapli_params["host"]
Out[8]: '192.168.100.1'
```

### maps, parents

```
In [12]: scrapli_params.maps
Out[12]:
[{'host': '192.168.100.1',
  'auth_username': 'cisco',
  'auth_password': 'cisco',
  'auth_secondary': 'cisco',
  'platform': 'cisco_iosxe',
  'timeout_socket': 15},
 {'auth_strict_key': False, 'timeout_socket': 5, 'timeout_transport': 10}]

In [13]: scrapli_params.parents
Out[13]: ChainMap({'auth_strict_key': False, 'timeout_socket': 5, 'timeout_transport': 10}
↪)
```

### new_child

```
In [18]: pprint(scrapli_params)
ChainMap({'auth_password': 'cisco',
          'auth_secondary': 'cisco',
          'auth_username': 'cisco',
          'host': '192.168.100.1',
          'platform': 'cisco_iosxe',
          'timeout_socket': 15},
         {'auth_strict_key': False,
          'timeout_socket': 5,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
          'timeout_transport': 10})

In [19]: updated_info = {"host": "10.1.1.1"}

In [20]: new_params = scrapli_params.new_child(updated_info)

In [22]: pprint(new_params)
ChainMap({'host': '10.1.1.1'},
         {'auth_password': 'cisco',
          'auth_secondary': 'cisco',
          'auth_username': 'cisco',
          'host': '192.168.100.1',
          'platform': 'cisco_iosxe',
          'timeout_socket': 15},
         {'auth_strict_key': False,
          'timeout_socket': 5,
          'timeout_transport': 10})

In [23]: new_params["host"]
Out[23]: '10.1.1.1'

In [25]: pprint(scrapli_params)
ChainMap({'auth_password': 'cisco',
          'auth_secondary': 'cisco',
          'auth_username': 'cisco',
          'host': '192.168.100.1',
          'platform': 'cisco_iosxe',
          'timeout_socket': 15},
         {'auth_strict_key': False,
          'timeout_socket': 5,
          'timeout_transport': 10})
```

## Counter

Counter - подкласс словаря для подсчета хешируемых объектов

```
class collections.Counter([iterable-or-mapping])
```

- `elements()`
- `most_common([n])`
- `subtract([iterable-or-mapping])`
- `total()` (New in version 3.10)

- fromkeys(iterable)

```
In [2]: c1 = Counter("aaabbbccccddddd")

In [3]: c1
Out[3]: Counter({'a': 3, 'b': 3, 'c': 4, 'd': 5})

In [4]: list(c1.elements())
Out[4]: ['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c', 'c', 'd', 'd', 'd', 'd', 'd']

In [5]: c1.most_common(2)
Out[5]: [('d', 5), ('c', 4)]
```

Операции с Counter

```
In [3]: c1
Out[3]: Counter({'a': 3, 'b': 3, 'c': 4, 'd': 5})

In [6]: c2 = Counter(a=1, d=5)

In [7]: c2
Out[7]: Counter({'a': 1, 'd': 5})

In [9]: c1 - c2
Out[9]: Counter({'a': 2, 'b': 3, 'c': 4})

In [11]: c1.subtract(c2)

In [12]: c1 + c2
Out[12]: Counter({'a': 3, 'b': 3, 'c': 4, 'd': 5})
```

### Пример использования

```
import re
import string
from pprint import pprint
from collections import Counter


with open("text.txt") as f:
    content = f.read()

clear_text = re.sub(f"[{string.punctuation}»«]", " ", content.lower())
words = re.split("\s+", clear_text)
stats = Counter(words)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
pprint(stats.most_common(20))
```

Результат

```
[('с', 9),
 ('не', 8),
 ('потоков', 6),
 ('в', 6),
 ('из', 5),
 ('это', 5),
 ('print', 5),
 ('при', 4),
 ('потоками', 4),
 ('если', 4),
 ('будет', 4),
 ('на', 4),
 ('работает', 4),
 ('работать', 3),
 ('и', 3),
 ('например', 3),
 ('нормально', 3),
 ('что', 3),
 ('сообщения', 3),
 ('может', 3)]
```

## collections.OrderedDict

OrderedDict похож на обычные словари, но имеют некоторые дополнительные возможности, связанные с операциями упорядочивания. Начиная с Python 3.7, обычные словари также стали упорядоченными, но при этом в OrderedDict остались несколько полезных возможностей:

- Обычный dict оптимизирован на операции mapping (getitem, setitem, deleteitem)
- OrderedDict оптимизирован под операции переупорядочивание
- Алгоритмически OrderedDict может обрабатывать частые операции переупорядочения лучше, чем dict
- При сравнении равенства словарей, OrderedDict учитывает соответствие порядка
- У OrderedDict есть метод `move_to_end()` для эффективного перемещения элемента в конец словаря

```
class collections.OrderedDict([items])
```

- `popitem(last=True)`

- move_to_end(key, last=True)

```
In [10]: d1 = {1: 100, 2: 200}

In [11]: d2 = {2: 200, 1: 100}

In [12]: d1 == d2
Out[12]: True

In [13]: from collections import OrderedDict

In [14]: o1 = OrderedDict({1: 100, 2: 200})

In [15]: o2 = OrderedDict({2: 200, 1: 100})

In [16]: o1 == o2
Out[16]: False
```

### move_to_end

```
In [17]: o1
Out[17]: OrderedDict([(1, 100), (2, 200)])

In [18]: o1.move_to_end(1)

In [19]: o1
Out[19]: OrderedDict([(2, 200), (1, 100)])
```

## collections.defaultdict

defaultdict - подкласс словаря, который вызывает указанную функцию для подстановки несуществующих значений

```
class collections.defaultdict(default_factory=None, /[, ...])
```

```
In [2]: from collections import defaultdict

In [3]: data = "some very important text"

In [4]: d = defaultdict(int)

In [5]: d
Out[5]: defaultdict(int, {})
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [6]: for letter in data:
   ...:     d[letter] += 1
   ...:

In [7]: d
Out[7]:
defaultdict(int,
            {'s': 1,
             'o': 2,
             'm': 2,
             'e': 3,
             ' ': 3,
             'v': 1,
             'r': 2,
             'y': 1,
             'i': 1,
             'p': 1,
             't': 4,
             'a': 1,
             'n': 1,
             'x': 1})

In [9]: sorted(d.items(), key=lambda x: x[-1], reverse=True)
Out[9]:
[('t', 4),
 ('e', 3),
 (' ', 3),
 ('o', 2),
 ('m', 2),
 ('r', 2),
 ('s', 1),
 ('v', 1),
 ('y', 1),
 ('i', 1),
 ('p', 1),
 ('a', 1),
 ('n', 1),
 ('x', 1)]
```

### Пример использования

config_sw1.txt

```
interface FastEthernet0/0
 switchport mode access
 switchport access vlan 10
!
interface FastEthernet0/1
 switchport trunk encapsulation dot1q
 switchport trunk allowed vlan 100,200
 switchport mode trunk
!
interface FastEthernet0/2
 switchport mode access
 switchport access vlan 20
!
interface FastEthernet0/3
 switchport trunk encapsulation dot1q
 switchport trunk allowed vlan 100,300,400,500,600
 switchport mode trunk
```

```
{'FastEthernet0/0': ['switchport mode access', 'switchport access vlan 10'],
 'FastEthernet0/1': ['switchport trunk encapsulation dot1q',
                     'switchport trunk allowed vlan 100,200',
                     'switchport mode trunk'],
 'FastEthernet0/2': ['switchport mode access', 'switchport access vlan 20'],
 'FastEthernet0/3': ['switchport trunk encapsulation dot1q',
                     'switchport trunk allowed vlan 100,300,400,500,600',
                     'switchport mode trunk'],
 'FastEthernet1/0': ['switchport mode access', 'switchport access vlan 20'],
 'FastEthernet1/1': ['switchport mode access', 'switchport access vlan 30'],
 'FastEthernet1/2': ['switchport trunk encapsulation dot1q',
                     'switchport trunk allowed vlan 400,500,600',
                     'switchport mode trunk']}
```

```
from pprint import pprint
from collections import defaultdict


def get_ip_from_cfg(filename):
    result = defaultdict(list)
    with open(filename) as f:
        for line in f:
            if line.startswith("interface"):
                intf = line.split()[-1]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            elif line.startswith(" switchport"):
                result[intf].append(line.strip())
    return result


if __name__ == "__main__":
    pprint(get_ip_from_cfg("config_sw1.txt"))
```

## UserList, UserDict, UserStr

Класс UserList действует как оболочка для list. Это полезный базовый класс для создания своих классов, которые могут наследовать UserList и переопределять существующие методы или добавлять новые.

### Почему бы не наследовать встроенные list, dict, str?

При наследовании UserDict

```
from collections import UserDict

class MyDict(UserDict):
    def __setitem__(self, key, value):
        print("__setitem__", key)


In [20]: d1 = MyDict({1: 100, 2: 200})
__setitem__ 1
__setitem__ 2

In [21]: d1.update({3: 300})
__setitem__ 3
```

Встроенный dict

```
class MyDict(dict):
    def __setitem__(self, key, value):
        print("__setitem__", key)


In [12]: d1 = MyDict({1: 100, 2: 200})

In [15]: d1.update({3: 300})
```

### UserList

```
from collections import UserList


class Task:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f"Task('{self.name}')"


class ToDo(UserList):
    def postpone(self):
        first_task = self.data.pop(0)
        self.data.append(first_task)


task1 = Task("task1")
task2 = Task("task2")
task3 = Task("task3")
task4 = Task("task4")
task5 = Task("task5")

todo_list = [task1, task2, task3]
todo = ToDo(todo_list)
```

Использование

```
In [2]: todo
Out[2]: [Task('task1'), Task('task2'), Task('task3')]

In [3]: todo[0]
Out[3]: Task('task1')

In [4]: todo[:-1]
Out[4]: [Task('task1'), Task('task2')]

In [5]: todo.postpone()

In [6]: todo
Out[6]: [Task('task2'), Task('task3'), Task('task1')]
```

Для сравнения версия Todo с использованием collections.abc.MutableSequence

```
from collections.abc import MutableSequence


class Task:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f"Task('{self.name}')"


class ToDo(MutableSequence):
    def __init__(self, tasks):
        self.tasks = tasks

    def __repr__(self):
        return f"ToDo('{self.tasks}')"

    def __getitem__(self, index):
        print("__getitem__", index)
        return self.tasks[index]

    def __setitem__(self, index, value):
        print("__setitem__", index)
        self.tasks[index] = value

    def __delitem__(self, index):
        print("__detitem__", index)
        del self.tasks[index]

    def __len__(self):
        return len(self.tasks)

    def insert(self, index, value):
        self.tasks.insert(index, value)


task1 = Task("task1")
task2 = Task("task2")
task3 = Task("task3")
task4 = Task("task4")
task5 = Task("task5")
todo_list = [task1, task2, task3, task4, task5]
todo = ToDo(todo_list)
```

# Отладчик pdb

## Основы pdb

Отладчик (debugger) это отдельный модуль, софт или часть редактора/IDE, которая позволяет делать отладку кода.

Примеры отладчиков и редакторов с отладчиками:

- pdb - модуль стандартной библиотеки Python (есть много разновидностей pdbr, ipdb, pdbpp)
- отладчики в редакторах для начинающих: Mu, Thonny
- PyCharm
- VS Code

Как запустить pdb

```
python -m pdb script.py
```

Для выхода из pdb используется команда q.

В любой момент можно перезапустить скрипт, без потери breakpoint, с помощью команды `run`.

### Базовые команды передвижения по программе

- n (next) - выполнить все до следующей строки. Эта команда не заходит в функции, которые вызываются в строке
- s (step) - выполнить текущую строку, остановиться как можно раньше. Эта команда заходит в функции, которые вызываются в строке
- c (continue) - выполнить все до breakpoint. Также полезна, когда скрипт отрабатывает с исключением, позволяет дойти до строки, где возникло исключение

### Контекст в коде, переменные

- l (list) - показывает следующую строку, которая будет выполняться и 5 строк до и после нее. При добавлении диапазона показывает указанные строки, например, `list 1, 20` покажет код с 1 по 20 строку
- ll (longlist) - показывает весь метод или функцию в котором мы находимся
- a (args) - показывает аргументы функции (или метода) и их значения. Работает только внутри функции

- p - показывает значение переменной, работает как print. Синтаксис `p vara`, где vara имя переменной
- pp - показывает значение переменной, работает как pprint. Синтаксис `pp vara`, где vara имя переменной

### Выполнение Python команд в pdb

Любую команду можно выполнить указав ! перед ней:

```
!vara = 55
!result.append(vara)
```

Таким образом можно пробовать выполнить какие-то действия в текущем контексте программы, изменить значения переменных.

Также можно перейти в интерпретатор python из текущего контекста. Для этого используется команда interact:

```
(Pdb) interact
*interactive*
>>> print(cfg)
<_io.TextIOWrapper name='sh_cdp_n_sw1.txt' mode='r' encoding='UTF-8'>
>>> cfg.closed
False
>>>
>>> data = ['1','2','3']
>>> print(','.join(data))
1,2,3
>>>
now exiting InteractiveConsole...
(Pdb)
```

Для выхода из интерпретатор используется команда `Ctrl-d`.

### Дополнительные команды по передвижению

- until - выполнить все до указанной строки. Синтаксис `until 15`, где 15 номер строки
- return - выполняется внутри функции и выполняет все до return
- u (up) - передвинутся на один уровень выше в стеке вызовов. Например, если мы по цепоцке переходили в один вызов функции, затем в друго, чтобы вернуться назад надо использовать up
- d (down) - передвинутся на один уровень ниже в стеке вызовов

### Breakpoints

- b (break) - команда для установки breakpoint

Если команда указывается с аргументом, например, `break 12` или `break check_ip`, устанавливается breakpoint. Без аргументов, команда показывает все установленные breakpoint.

Удаление breakpoint под номером 1:

```
clear 1
```

Удалить все breakpoint можно clear без аргументов.

### Базовые варианты установки breakpoint

Установить breakpoint в строке 12:

```
break 12
```

Установить breakpoint в первой строке функции check_ip:

```
break check_ip
```

### Breakpoint с условием

Сделать breakpoint в строке 12, если значение переменной num будет больше 10:

```
break 12, num > 10
```

### Привязка команд к breakpoint

Создаем breakpoint (предполагаем, что он первый, поэтому его номер будет 1):

```
break 12
```

Добавляем команды, которые будут выполняться каждый раз, когда попадаем на breakpoint (var1, var2, result_dict должны быть заменены на ваши переменные)

```
commands 1
 pp var1
 pp var2
 pp result_dict
end
```

## ipdb

Модуль ipdb это одна из разновидностей pdb, которая добавляет подсветку синтаксиса, вызов ipython вместо встроенного интерпретатора, автопродолжение команд.

Установка ipdb:

```
pip install ipdb
```

Как запустить ipdb

```
python -m ipdb script.py
```

В остальном, команды те же, что в и pdb, только по команде interact откроется ipython, а не встроенный интепретатор python.

## Дополнительные материалы

- The Python Debugger (pdb) - основы работы с pdb
- Python 3 Module of the Week. pdb — Interactive Debugger. Перевод на русский
- Python Debugging With Pdb - в конце статьи есть PDF со списком команд
- Nathan Yergler: In Depth PDB - PyCon 2014

# 7

# Продолжение обучения

- Fluent Python первое издание (второе издание выйдет осенью 2021 года)

# 8

# Скачать PDF/Epub

- Epub
- PDF