Python для сетевых инженеров. Том 2: Продвинутые возможности Python с примерами на сетевой тематике

Release 0.0.1

Nov 23, 2019

Оглавление

Warning: Книга в процессе написания!

1

I. Объектно-ориентированное программирование

1. 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

Рассмотрим практический пример использования ООП.

В разделе “18. Работа с базами данных” первое, что нужно было сделать для работы с БД, подключиться к ней:

```
In [1]: import sqlite3

In [2]: conn = sqlite3.connect('dhcp_snooping.db')
```

Переменная conn - это объект, который представляет реальное соединение с БД. Благодаря функции type, можно выяснить экземпляром какого класса является объект conn:

```
In [3]: type(conn)
Out[3]: sqlite3.Connection
```

У conn есть свои методы и переменные, которые зависят от состояния текущего объекта. Например, переменная экземпляра conn.in_transaction доступна у каждого экземпляра класса sqlite3.Connection и возвращает True или False, в зависимости от того все ли изменения закоммичены:

```
In [15]: conn.in_transaction
Out[15]: False
```

Метод execute выполняет команду SQL:

```
In [19]: query = 'insert into dhcp (mac, ip, vlan, interface) values (?, ?, ?, ?)'

In [5]: conn.execute(query, ('0000.1111.7777', '10.255.1.1', '10', 'Gi0/7'))
Out[5]: <sqlite3.Cursor at 0xb57328a0>
```

При этом, объект conn сохраняет состояние: теперь переменная экзепляра conn.in_transaction, возвращает True:

```
In [6]: conn.in_transaction
Out[6]: True
```

После вызова метода commit, она опять равна False:

```
In [7]: conn.commit()

In [8]: conn.in_transaction
Out[8]: False
```

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 на данными 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, это 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======='
```

Note: На примере со сторокой понятно, что класс не обязан хранить состояние - строка неизменяемый тип данных и все методы возвращают новые строки и не изменяют исходную строку.

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

Note: 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
   ...:
```

Note: 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()

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 mode:

```
In [1]: def info(sw_obj):
   ...:     print('Hostname: {}\nModel: {}'.format(sw_obj.hostname, sw_obj.model))
   ...:

In [2]: sw1 = Switch()

In [3]: sw1.hostname = 'sw1'

In [4]: sw1.model = 'Cisco 3850'

In [5]: info(sw1)
Hostname: sw1
Model: Cisco 3850
```

В функции info параметр `sw_obj` ожидает экземпляр класса `Switch`. Скорее всего, в это 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.

Note: При вызове метода экземпляра класса, ссылка на экземпляр передается первым 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('Hostname: {}\nModel: {}'.format(sw_object.hostname, 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()
Hostname: sw1
Model: Cisco 3850
```

Warning: 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 = ['{}{}'.format(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('Hostname: {}\nModel: {}'.format(self.hostname, self.model))
   ...:
   ...:     def generate_interfaces(self, intf_type, number_of_intf):
   ...:         interfaces = ['{}{}'.format(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)
```

(continues on next page)

(continued from previous page)

```
      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('Hostname: {}\nModel: {}'.format(self.hostname, 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
```

Note: 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.

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 функция, 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:

```
In [32]: 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:

```
In [1]: class CiscoSSH:
   ...:     device_type = 'cisco_ios'
   ...:
   ...:     def send_command(self, command):
   ...:         pass
   ...:
```

Теперь не только у класса, но и у каждого экземпляра класса будет переменная `device_type`:

```
In [2]: CiscoSSH.device_type
Out[2]: 'cisco_ios'

In [3]: r1 = CiscoSSH()

In [4]: r1.device_type
Out[4]: 'cisco_ios'
```

(continues on next page)

(continued from previous page)

```
In [5]: r2 = CiscoSSH()

In [6]: r2.device_type
Out[6]: 'cisco_ios'
```

Важный момент при использовании переменных класса, то что внутри метода к ним все равно надо обращаться через имя класса. Для начала, вариант обращения без имени класса:

```
In [7]: class CiscoSSH:
   ...:     device_type = 'cisco_ios'
   ...:
   ...:     def send_command(self, command):
   ...:         print(device_type)
   ...:

In [8]: r1 = CiscoSSH()

In [9]: r1.send_command()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-921b8733dbee> in <module>()
----> 1 r1.send_command()

<ipython-input-7-ef923c4e39d3> in send_command(self, command)
      3
      4     def send_command(self, command):
----> 5         print(device_type)
      6

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

И правильный вариант:

```
In [10]: class CiscoSSH:
    ...:     device_type = 'cisco_ios'
    ...:
    ...:     def send_command(self, command):
    ...:         print(CiscoSSH.device_type)
    ...:

In [11]: r1 = CiscoSSH()

In [12]: r1.send_command()
```

(continues on next page)

(continued from previous page)

```
'cisco_ios'
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

Задание 1.1

Создать класс IPv4Network, который представляет сеть. При создании экземпляра класса, как аргумент передается строка с адресом сети.

Пример создания экземпляра класса:

```
In [3]: net1 = IPv4Network('10.1.1.0/29')
```

После этого, должны быть доступны переменные address и mask:

```
In [5]: net1.address
Out[5]: '10.1.1.0'

In [6]: net1.mask
Out[6]: 29
```

Broadcast адрес должен быть записан в атрибуте broadcast:

```
In [7]: net1.broadcast
Out[7]: '10.1.1.7'
```

Также должен быть создан атрибут allocated в котором будет храниться кортеж с адресами, которые назначены на каком-то устройстве/хосте. Изначально атрибут равен пустому кортежу:

```
In [8]: print(net1.allocated)
()
```

Метод hosts должен возвращать кортеж IP-адресов, которые входят в сеть, не включая адрес сети и broadcast:

```
In [9]: net1.hosts()
Out[9]: ('10.1.1.1', '10.1.1.2', '10.1.1.3', '10.1.1.4', '10.1.1.5', '10.1.1.6')
```

Метод allocate ожидает как аргумент IP-адрес. Указанный адрес должен быть записан в кортеж в атрибуте net1.allocated:

```
In [10]: net1 = IPv4Network('10.1.1.0/29')

In [11]: print(net1.allocated)
()

In [12]: net1.allocate('10.1.1.6')

In [13]: net1.allocate('10.1.1.3')

In [14]: print(net1.allocated)
('10.1.1.6', '10.1.1.3')
```

Метод unassigned возвращает возвращает кортеж со свободными адресами:

```
In [15]: net1 = IPv4Network('10.1.1.0/29')

In [16]: net1.allocate('10.1.1.4')
    ...: net1.allocate('10.1.1.6')
    ...: net1.allocate('10.1.1.8')
    ...:

In [17]: net1.unassigned()
Out[17]: ('10.1.1.1', '10.1.1.2', '10.1.1.3', '10.1.1.5')
```

Задание 1.2

Создать класс PingNetwork. При создании экземпляра класса PingNetwork, как аргумент передается экземпляр класса IPv4Network.

У класса PingNetwork должны быть методы _ping и scan. Метод _ping с параметром ip: должен пинговать один IP-адрес и возвращать

- True - если адрес пингуется
- False - если адрес не пингуется

Метод scan с таким параметрами:

- workers - значение по умолчанию 5

- include_unassigned - значение по умолчанию False

Метод scan:

- Пингует адреса из сети, которая передается как аргумент при создании экземпляра.
- Адреса должны пинговаться в разных потоках, для этого использовать concurrent.futures.
- По умолчанию, пингуются только адреса, которые находятся в атрибуте allocated. Если параметр include_unassigned равен True, должны пинговаться и адреса unassigned.
- Метод должен возвращать кортеж с двумя списками: список доступных IP-адресов и список недоступных IP-адресов

Пример работы с классом PingNetwork. Сначала создаем сеть:

```
In [3]: net1 = IPv4Network('8.8.4.0/29')
```

И выделяем несколько адресов:

```
In [4]: net1.allocate('8.8.4.2')
   ...: net1.allocate('8.8.4.4')
   ...: net1.allocate('8.8.4.6')
   ...:

In [5]: net1.allocated
Out[5]: ('8.8.4.2', '8.8.4.4', '8.8.4.6')

In [6]: net1.unassigned()
Out[6]: ('8.8.4.1', '8.8.4.3', '8.8.4.5')
```

Затем создается экземпляр класса PingNetwork, а сеть передается как аргумент:

```
In [8]: ping_net = PingNetwork(net1)
```

Пример работы метода scan:

```
In [9]: ping_net.scan()
Out[9]: (['8.8.4.4'], ['8.8.4.2', '8.8.4.6'])

In [10]: ping_net.scan(include_unassigned=True)
Out[10]: (['8.8.4.4'], ['8.8.4.2', '8.8.4.6', '8.8.4.1', '8.8.4.3', '8.8.4.5'])
```

2. 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”.

Note: 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,
```

(continues on next page)

(continued from previous page)

```
            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:
            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)
```

(continues on next page)

(continued from previous page)

```
        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
```

Note: 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
    ...:

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
   ...:
```

(continues on next page)

(continued from previous page)

```
   ...:     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)
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')
```

(continues on next page)

(continued from previous page)

```
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}')"
   ...:
```

(continues on next page)

(continued from previous page)

```
   ...:     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
    ...:
```

(continues on next page)

(continued from previous page)

```
...:     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 экземплярами 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'
```

(continues on next page)

(continued from previous page)

```
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'
```

See also:

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.

Note: 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__
>>>> 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):
```

(continues on next page)

(continued from previous page)

```
        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__')
```

(continues on next page)

(continued from previous page)

```
        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

    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
```

(continues on next page)

(continued from previous page)

```
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
    ...:
```

(continues on next page)

(continued from previous page)

```
    ...:     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()]
   ...:
   ...:     def __iter__(self):
```

(continues on next page)

(continued from previous page)

```
   ...:         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)

(continued from previous page)

```
 '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')
```

(continues on next page)

(continued from previous page)

```
        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
```

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,
```

(continues on next page)

(continued from previous page)

```
            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)
        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__
```

(continues on next page)

(continued from previous page)

```
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

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

Задание 2.1

Скопировать класс IPv4Network из задания 1.1 и добавить ему все методы, которые необходимы для реализации протокола последовательности (sequence):

- __getitem__, __len__, __contains__, __iter__
- index, count - должны работать аналогично методам в списках и кортежах

Плюс оба метода, которые отвечают за строковое представление экземпляров класса IPv4Network.

Пример создания экземпляра класса:

```
In [2]: net1 = IPv4Network('8.8.4.0/29')
```

Проверка методов:

```
In [3]: for ip in net1:
   ...:     print(ip)
   ...:
8.8.4.1
8.8.4.2
8.8.4.3
8.8.4.4
8.8.4.5
8.8.4.6

In [4]: net1[2]
Out[4]: '8.8.4.3'

In [5]: net1[-1]
Out[5]: '8.8.4.6'
```

(continues on next page)

(continued from previous page)

```
In [6]: net1[1:4]
Out[6]: ('8.8.4.2', '8.8.4.3', '8.8.4.4')

In [7]: '8.8.4.4' in net1
Out[7]: True

In [8]: net1.index('8.8.4.4')
Out[8]: 3

In [9]: net1.count('8.8.4.4')
Out[9]: 1

In [10]: len(net1)
Out[10]: 6
```

Строковое представление:

```
In [13]: net1
Out[13]: IPv4Network(8.8.4.0/29)

In [14]: str(net1)
Out[14]: 'IPv4Network 8.8.4.0/29'
```

Задание 2.2

Скопировать класс PingNetwork из задания 1.2 и изменить его таким образом, чтобы адреса пинговались не при вызове метода scan, а при вызове экземпляра.

Вся функциональность метода scan должна быть перенесена в метод, который отвечает за вызов экземпляра.

Пример работы с классом PingNetwork. Сначала создаем сеть:

```
In [2]: net1 = IPv4Network('8.8.4.0/29')
```

И выделяем несколько адресов:

```
In [3]: net1.allocate('8.8.4.2')
   ...: net1.allocate('8.8.4.4')
   ...: net1.allocate('8.8.4.6')
   ...:
```

Затем создается экземпляр класса PingNetwork, сеть передается как аргумент:

```
In [6]: ping_net = PingNetwork(net1)
```

После этого экземпляр должен быть вызываемым объектом (callable):

```
In [7]: ping_net()
Out[7]: (['8.8.4.4'], ['8.8.4.2', '8.8.4.6'])

In [8]: ping_net(include_unassigned=True)
Out[8]: (['8.8.4.4'], ['8.8.4.2', '8.8.4.6', '8.8.4.1', '8.8.4.3', '8.8.4.5'])
```

3. Classmethod, staticmethod, property

В Python есть ряд полезных встроенных декораторов, которые позволяют менять поведение методов класса. Декораторы рассматриваются позже довольно подробно. На данном этапе достаточно знать, что декоратор это синтаксический сахар, который упрощает запись `func = decorator(func)` и позволяет писать так:

```
@decorator
def func():
    pass
```

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 сдедать переменную неищменяемой 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")
```

(continues on next page)

(continued from previous page)

```
   ...:         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
   ...:

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)
```

(continues on next page)

(continued from previous page)

```
<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):
```

(continues on next page)

(continued from previous page)

```
---> 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


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 чте-

ние/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:
            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
```

(continues on next page)

(continued from previous page)

```
    @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'

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
```

(continues on next page)

(continued from previous page)

```
    # 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)

    # а 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
```

(continues on next page)

(continued from previous page)

```
    # 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')
        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)):
```

(continues on next page)

(continued from previous page)

```
            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):
        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,
                 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:

- Один и тот же метод используется для всех экземпляров класса, то есть, метод не нужно инициализировать для каждого экземпляра.
- 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

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'):
```

(continues on next page)

(continued from previous page)

```
        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',
  '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'}]
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

Задание 3.1

Скопировать класс IPv4Network из задания 1.1. Переделать класс таким образом, чтобы методы hosts и unassigned стали переменными, но при этом значение переменной экземпляра вычислялось каждый раз при обращении и запись переменной была запрещена.

Пример создания экземпляра класса:

```
In [1]: net1 = IPv4Network('8.8.4.0/29')

In [2]: net1.hosts
Out[2]: ('8.8.4.1', '8.8.4.2', '8.8.4.3', '8.8.4.4', '8.8.4.5', '8.8.4.6')

In [3]: net1.allocate('8.8.4.2')

In [4]: net1.allocate('8.8.4.3')

In [5]: net1.unassigned
Out[5]: ('8.8.4.1', '8.8.4.4', '8.8.4.5', '8.8.4.6')
```

Запись переменной:

```
In [6]: net1.unassigned = 'test'
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-6-c98e898835e1> in <module>
----> 1 net1.unassigned = 'test'

AttributeError: can't set attribute
```

Задание 3.1a

Скопировать класс IPv4Network из задания 3.1. Добавить метод from_tuple, который позволяет создать экземпляр класса IPv4Network из кортежа вида ('10.1.1.0', 29).

Пример создания экземпляра класса:

```
In [3]: net2 = IPv4Network.from_tuple(('10.1.1.0', 29))

In [4]: net2
Out[4]: IPv4Network(10.1.1.0/29)
```

Задание 3.2

Скопировать класс PingNetwork из задания 1.2. Один из методов класса зависит только от значения аргумента и не зависит от значений переменных экземпляра или другого состояния объекта.

Сделать этот метод статическим и проверить работу метода.

Задание 3.3

Создать класс User, который представляет пользователя. При создании экземпляра класса, как аргумент передается строка с именем пользователя.

Пример создания экземпляра класса:

```
In [3]: nata = User('nata')
```

После этого, должна быть доступна переменная username:

```
In [4]: nata.username
Out[4]: 'nata'
```

Переменная username должна быть доступна только для чтения:

```
In [5]: nata.username = 'user'
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-5-eba76ef1ed86> in <module>
----> 1 nata.username = 'user'

AttributeError: can't set attribute
```

Также в экземпляре должа быть создана переменная password, но пока пользователь не установил пароль, при обращении к переменной должно генерироваться исключение ValueError:

```
In [6]: nata.password
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-7527817bf03d> in <module>
----> 1 nata.password
...
ValueError: Надо установить пароль!
```

При установке пароля должны выполняться проверки:

- длины пароля - минимальная разрешенная длина пароля 8 символов
- содержится ли имя пользователя в пароле

Если проверки не прошли, надо вывести сообщение об ошибке и запросить пароль еще раз:

```
In [7]: nata.password = 'sadf'
Пароль слишком короткий. Введите пароль еще раз: sdlkjfksnatasdfsd
Пароль содержит имя пользователя. Введите пароль еще раз: asdfkpeorti2435
Пароль установлен
```

Если пароль прошел проверки, должно выводиться сообщение “Пароль установлен”

```
In [8]: nata.password = 'sadfsadfsadf'
Пароль установлен
```

4. 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.

Принцип подстановки Барбары Лисков

В Python синтаксис наследования используется с абстрактными классами для наследования интерфейса/протокола. Кроме того, синтаксис наследования используется с Mixin.

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
        self._MAX_READ = 10000

        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
```

(continues on next page)

(continued from previous page)

```
            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)

(continued from previous page)

```
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

Пользовательские исключения

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()
```

(continues on next page)

(continued from previous page)

```
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()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
```

(continues on next page)

(continued from previous page)

```
            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):
        self._ssh.send(command + '\n')
        time.sleep(0.5)
```

(continues on next page)

(continued from previous page)

```
        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

```

(continues on next page)

(continued from previous page)

```
In [3]: 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()]
   ...:
```

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',
```

(continues on next page)

(continued from previous page)

```
 '10.1.1.195',
 '10.1.1.196',
 '10.1.1.197',
 '10.1.1.198']

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.

Note: What is a mixin, and why are they useful? Перевод “Что такое mixin и почему они полезны?”

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
```

(continues on next page)

(continued from previous page)

```
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>
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
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

Задание 4.1

Создать класс CiscoTelnet, который наследует класс TelnetBase из файла base_telnet_class.py.

Переписать метод __init__ в классе CiscoTelnet таким образом:

- добавить параметры:
- enable - пароль на режим enable
- disable_paging - отключает постраничный вывод команд, по умолчанию равен True
- после подключения по Telnet должен выполняться переход в режим enable: для этого в методе __init__ должен сначала вызываться метод __init__ класса TelnetBase, а затем выполняться переход в режим enable.

Добавить в класс CiscoTelnet метод send_show_command, который отправляет команду show и возвращает ее вывод в виде строки. Метод ожидает как аргумент одну команду.

Добавить в класс CiscoTelnet метод send_config_commands, который отправляет одну или несколько команд на оборудование в конфигурационном режиме и возвращает ее вывод в виде строки. Метод ожидает как аргумент одну команду (строку) или несколько команд (список).

Пример работы класса:

```
In [1]: r1 = CiscoTelnet('192.168.100.1', 'cisco', 'cisco', 'cisco')

Метод send_show_command:
In [2]: r1.send_show_command('sh clock')
Out[2]: 'sh clock\r\n*09:39:38.633 UTC Thu Oct 10 2019\r\nR1#'

Метод send_config_commands:
In [3]: r1.send_config_commands('logging 7.7.7.7')
```

(continues on next page)

(continued from previous page)

```
Out[3]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.
↪\r\nR1(config)#logging 7.7.7.7\r\nR1(config)#end\r\nR1#'

In [4]: r1.send_config_commands(['interface loop77', 'ip address 107.7.7.7 255.
↪255.255.255'])
Out[4]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.
↪\r\nR1(config)#interface loop77\r\nR1(config-if)#ip address 107.7.7.7 255.255.
↪255.255\r\nR1(config-if)#end\r\nR1#'
```

Задание 4.1a

Скопировать класс CiscoTelnet из задания 4.1 и добавить проверку на ошибки.

Добавить метод _check_error_in_command, который выполняет проверку на такие ошибки:

- Invalid input detected, Incomplete command, Ambiguous command

Создать исключение ErrorInCommand, которое будет генерироваться при возникновении ошибки на оборудовании.

Метод ожидает как аргумент команду и вывод команды. Если в выводе не обнаружена ошибка, метод ничего не возвращает. Если в выводе найдена ошибка, метод генерирует исключение ErrorInCommand с сообщением о том какая ошибка была обнаружена, на каком устройстве и в какой команде.

Добавить проверку на ошибки в методы send_show_command и send_config_commands.

Пример работы класса с ошибками:

```
In [1]: r1 = CiscoTelnet('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [2]: r1.send_show_command('sh clck')
---------------------------------------------------------------------------
ErrorInCommand                            Traceback (most recent call last)
<ipython-input-2-e26d712f3ad3> in <module>
----> 1 r1.send_show_command('sh clck')
...
ErrorInCommand: При выполнении команды "sh clck" на устройстве 192.168.100.1␣
↪возникла ошибка "Invalid input detected at '^' marker.

In [3]: r1.send_config_commands('loggg 7.7.7.7')
---------------------------------------------------------------------------
ErrorInCommand                            Traceback (most recent call last)
<ipython-input-3-ab4a1ce52554> in <module>
----> 1 r1.send_config_commands('loggg 7.7.7.7')
```

(continues on next page)

(continued from previous page)

```
...
ErrorInCommand: При выполнении команды "loggg 7.7.7.7" на устройстве 192.168.100.
↪1 возникла ошибка "Invalid input detected at '^' marker.
```

Без ошибок:

```
In [4]: r1.send_show_command('sh clock')
Out[4]: 'sh clock\r\n*09:39:38.633 UTC Thu Oct 10 2019\r\nR1#'

In [5]: r1.send_config_commands('logging 7.7.7.7')
Out[5]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.
↪\r\nR1(config)#logging 7.7.7.7\r\nR1(config)#end\r\nR1#'

In [6]: r1.send_config_commands(['interface loop77', 'ip address 107.7.7.7 255.
↪255.255.255'])
Out[6]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.
↪\r\nR1(config)#interface loop77\r\nR1(config-if)#ip address 107.7.7.7 255.255.
↪255.255\r\nR1(config-if)#end\r\nR1#'
```

Примеры команд с ошибками:

```
R1(config)#logging 0255.255.1
                   ^
% Invalid input detected at '^' marker.
R1(config)#logging
% Incomplete command.

R1(config)#sh i
% Ambiguous command:  "sh i"
```

Задание 4.2

Скопировать класс IPv4Network из задания 3.1 и изменить его таким образом, чтобы класс IPv4Network наследовал абстрактный класс Sequence. Создать все необходимые абстрактные методы для работы IPv4Network как Sequence.

Проверить, что работают все методы характерные для последовательности (sequence):

- __getitem__, __len__, __contains__, __iter__, index, count

Пример создания экземпляра класса:

```
In [1]: net1 = IPv4Network('8.8.4.0/29')
```

Проверка методов:

```
In [2]: len(net1)
Out[2]: 6

In [3]: net1[0]
Out[3]: '8.8.4.1'

In [4]: '8.8.4.1' in net1
Out[4]: True

In [5]: '8.8.4.10' in net1
Out[5]: False

In [6]: net1.count('8.8.4.1')
Out[6]: 1

In [7]: net1.index('8.8.4.1')
Out[7]: 0

In [8]: for ip in net1:
   ...:     print(ip)
   ...:
8.8.4.1
8.8.4.2
8.8.4.3
8.8.4.4
8.8.4.5
8.8.4.6
```

Задание 4.3

Создать класс Topology, который представляет топологию сети. Класс Topology должен наследовать абстрактный класс MutableMapping и для всех абстрактных методов класса MutableMapping должна быть написана рабочая реализация в классе Topology.

Проверить, что после реализации абстрактных методов, работают также такие методы: keys, items, values, get, pop, popitem, clear, update, setdefault.

При создании экземпляра класса, как аргумент передается словарь, который описывает топологию. В каждом экземпляре должна быть создана переменная topology, в которой содержится словарь топологии.

Пример создания экземпляра класса:

```
In [1]: t1 = Topology(example1)

In [2]: t1.topology
Out[2]:
{('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')}
```

Проверка реализации абстрактных методов:

Получение элемента:

```
In [3]: t1[('R1', 'Eth0/0')]
Out[3]: ('SW1', 'Eth0/1')
```

Перезапись/добавление элемента:

```
In [5]: t1[('R1', 'Eth0/0')] = ('SW1', 'Eth0/12')

In [6]: t1.topology
Out[6]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/12'),
 ('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')}

In [7]: t1[('R6', 'Eth0/0')] = ('SW1', 'Eth0/17')

In [8]: t1.topology
Out[8]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/12'),
 ('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'),
 ('R6', 'Eth0/0'): ('SW1', 'Eth0/17')}
```

Удаление:

```
In [11]: del t1[('R6', 'Eth0/0')]

In [12]: t1.topology
Out[12]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/12'),
 ('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')}
```

Итерация:

```
In [13]: for item in t1:
    ...:     print(item)
    ...:
('R1', 'Eth0/0')
('R2', 'Eth0/0')
('R2', 'Eth0/1')
('R3', 'Eth0/0')
('R3', 'Eth0/1')
('R3', 'Eth0/2')
```

Длина:

```
In [14]: len(t1)
Out[14]: 6
```

После реализации абстрактных методов, должны работать таким методы:

```
In [1]: t1.topology
Out[1]:
{('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')}
```

keys, values, items:

```
In [2]: t1.keys()
Out[2]: KeysView(<__main__.Topology object at 0xb562f82c>)

```

(continues on next page)

(continued from previous page)

```
In [3]: t1.values()
Out[3]: ValuesView(<__main__.Topology object at 0xb562f82c>)
```

Метод get:

```
In [4]: t1.get(('R2', 'Eth0/0'))
Out[4]: ('SW1', 'Eth0/2')

In [6]: print(t1.get(('R2', 'Eth0/4')))
None
```

Метод pop:

```
In [8]: t1.pop(('R2', 'Eth0/0'))
Out[8]: ('SW1', 'Eth0/2')

In [9]: t1.topology
Out[9]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Метод update:

```
In [10]: t2.topology
Out[10]: {('R1', 'Eth0/4'): ('R7', 'Eth0/0'), ('R1', 'Eth0/6'): ('R9', 'Eth0/0')}

In [11]: t1.update(t2)

In [13]: t1.topology
Out[13]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0'),
 ('R1', 'Eth0/4'): ('R7', 'Eth0/0'),
 ('R1', 'Eth0/6'): ('R9', 'Eth0/0')}
```

Метод clear:

```
In [14]: t1.clear()

In [15]: t1.topology
Out[15]: {}
```

Задание 4.3a

Скопировать класс Topology из задания 4.3. Переделать класс Topology таким образом, чтобы абстрактные методы могли удалять соединение и в том случае, когда вместо ключа, передается значение из словаря.

При создании экземпляра класса, как аргумент теперь передается словарь, который может содержать дублирующиеся соединения.

Дублем считается ситуация, когда в словаре есть такие пары:

```
('R1', 'Eth0/0'): ('SW1', 'Eth0/1') и ('SW1', 'Eth0/1'): ('R1', 'Eth0/0')
```

В каждом экземпляре должна быть создана переменная topology, в которой содержится словарь топологии, но уже без дублей. При удалении дублей надо оставить ту пару, где key < value.

То есть ключом должно быть меньший кортеж, а значением больший. Из таких двух пар:

```
('R1', 'Eth0/0'): ('SW1', 'Eth0/1') и ('SW1', 'Eth0/1'): ('R1', 'Eth0/0')
```

должна остаться первая `('R1', 'Eth0/0'): ('SW1', 'Eth0/1')`.

Пример создания экземпляра класса:

```
In [1]: t1 = Topology(example1)

In [2]: t1.topology
Out[2]:
{('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')}
```

Проверка реализации абстрактных методов:

получение элемента:

```
In [3]: t1[('R1', 'Eth0/0')]
Out[3]: ('SW1', 'Eth0/1')

In [4]: t1[('SW1', 'Eth0/2')]
Out[4]: ('R2', 'Eth0/0')
```

Перезапись/запись элемента:

```
In [5]: t1[('R1', 'Eth0/0')] = ('SW1', 'Eth0/12')

In [6]: t1.topology
Out[6]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/12'),
 ('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')}

In [7]: t1[('R6', 'Eth0/0')] = ('SW1', 'Eth0/17')

In [8]: t1.topology
Out[8]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/12'),
 ('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'),
 ('R6', 'Eth0/0'): ('SW1', 'Eth0/17')}

In [9]: t1[('SW1', 'Eth0/21')] = ('R7', 'Eth0/0')

In [10]: t1.topology
Out[10]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/12'),
 ('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'),
 ('R6', 'Eth0/0'): ('SW1', 'Eth0/17'),
 ('R7', 'Eth0/0'): ('SW1', 'Eth0/21')}
```

Удаление:

```
In [11]: del t1[('R7', 'Eth0/0')]

In [12]: t1.topology
Out[12]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/12'),
 ('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'),
 ('R6', 'Eth0/0'): ('SW1', 'Eth0/17')}

In [13]: del t1[('SW1', 'Eth0/17')]

In [14]: t1.topology
Out[14]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/12'),
 ('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')}
```

Итерация:

```
In [15]: for item in t1:
    ...:     print(item)
    ...:
('R1', 'Eth0/0')
('R2', 'Eth0/0')
('R2', 'Eth0/1')
('R3', 'Eth0/0')
('R3', 'Eth0/1')
('R3', 'Eth0/2')
```

Длина:

```
In [16]: len(t1)
Out[16]: 6
```

Задание 4.4

Создать класс OrderingMixin, который будет автоматически добавлять к объекту методы:

- __ge__ - операция >=
- __ne__ - операция !=
- __le__ - операция <=
- __gt__ - операция >

OrderingMixin предполагает, что в классе уже определены методы:

- __eq__ - операция ==
- __lt__ - операция <

Проверить работу примеси можно на примере класса IPAddress (класс находится в файле задания). Определение класса можно менять. OrderingMixin не должен использовать переменные класса IPAddress. Для работы методов должны использоваться только существующие методы __eq__ и __lt__. OrderingMixin должен работать и с любым другим классом у которого есть методы __eq__ и __lt__.

Пример проверки методов с классом IPAddress:

```
In [4]: ip1 = IPAddress('10.10.1.1')

In [5]: ip2 = IPAddress('10.2.1.1')

In [6]: ip1 < ip2
Out[6]: False

In [7]: ip1 > ip2
Out[7]: True

In [8]: ip1 >= ip2
Out[8]: True

In [9]: ip1 <= ip2
Out[9]: False

In [10]: ip1 == ip2
Out[10]: False

In [11]: ip1 != ip2
Out[11]: True
```

Задание 4.5

Создать примесь InheritanceMixin с двумя методами:

- subclasses - отображает дочерние классы
- superclasses - отображает родительские классы

Методы должны отрабатывать и при вызове через класс и при вызове через экземпляр:

```
In [2]: A.subclasses()
Out[2]: [__main__.B, __main__.D]

In [3]: A.superclasses()
Out[3]: [__main__.A, __main__.InheritanceMixin, object]

In [4]: a.subclasses()
Out[4]: [__main__.B, __main__.D]

In [5]: a.superclasses()
Out[5]: [__main__.A, __main__.InheritanceMixin, object]
```

В задании заготовлена иерархия классов, надо сделать так, чтобы у всех этих классов появились методы subclasses и superclasses. Определение классов можно менять.

5. Data classes

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

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)

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):
    ...:     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)
```

(continues on next page)

(continued from previous page)

```
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)>
```

Data classes

Data classes во многом похожи на именованные кортежи, но имеют более широкие возможности. Например, атрибуты класса могут быть изменяемые.

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.

Note: 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,
```

(continues on next page)

(continued from previous page)

```
    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.

Note: 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):
            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
```

(continues on next page)

(continued from previous page)

```
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
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)
```

(continues on next page)

(continued from previous page)

```
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)

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
```

(continues on next page)

(continued from previous page)

```
    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
   ...:

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)


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

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

Задание 5.1

Создать класс Route с использованием dataclass. У экземпляров класса должны быть доступны переменные: prefix, nexthop и protocol. В строковом представлении экземпляров не должна выводиться информация о протоколе.

Пример создания экземпляра класса:

```
In [2]: route1 = Route('10.1.1.0/24', '10.2.2.2', 'OSPF')
```

После этого, должны быть доступны переменные prefix, nexthop и protocol:

```
In [3]: route1.nexthop
Out[3]: '10.2.2.2'

In [4]: route1.prefix
Out[4]: '10.1.1.0/24'

In [5]: route1.protocol
Out[5]: 'OSPF'
```

Строковое представление:

```
In [6]: route1
Out[6]: Route(prefix='10.1.1.0/24', nexthop='10.2.2.2')
```

Задание 5.2

Дополнить класс IPAddress: добавить метод, который позволит выполнять сложение экземпляра класса IPAddress и числа. В результате сложения должен возвращаться новый экземпляр класса IPAddress.

Пример создания экземпляра класса:

```
In [7]: ip1 = IPAddress('10.10.1.1', 24)
```

Суммирование:

```
In [8]: ip1
Out[8]: IPAddress(ip='10.10.1.1', mask=24)

In [9]: ip1 + 5
Out[9]: IPAddress(ip='10.10.1.6', mask=24)

In [10]: ip2 = ip1 + 5

In [11]: isinstance(ip2, IPAddress)
Out[11]: True
```

Дополнить такой класс:

```
import ipaddress
from dataclasses import dataclass, field


@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))
```

Задание 5.3

Дополнить класс Book: добавить метод to_dict. Метод to_dict должен возвращать словарь в котором:

- ключи - имена переменных экземпляра
- значения - значения переменных

В словаре должны быть все переменные, кроме тех, которые начинаются на _.

Пример создания экземпляра класса:

```
In [4]: b1 = Book('Good Omens', 35, 5)
```

В этом случае должен возвращаться такой словарь:

```
In [5]: b1.to_dict()
Out[5]: {'title': 'Good Omens', 'price': 35.0, 'quantity': 124, 'total': 4340.0}
```

Обратите внимание, что в словаре не только простые переменные, но и переменные, которые созданы через property.

```
from dataclasses import dataclass, field

@dataclass
class Book:
    title: str
    price: float
    _price: float = field(init=False, repr=False)
    quantity: int = 0

    @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)
```

Задание 5.4

Создать класс IPv4Network с использованием dataclass. У экземпляров класса должны быть доступны:

- переменные: network, broadcast, hosts, allocated, unassigned
- метод allocate

Пример создания экземпляра класса:

```
In [3]: net1 = IPv4Network('10.1.1.0/29')
```

После этого, должна быть доступна переменная network:

```
In [6]: net1.network
Out[6]: '10.1.1.0/29'
```

Broadcast адрес должен быть записан в атрибуте broadcast:

```
In [7]: net1.broadcast
Out[7]: '10.1.1.7'
```

Также должен быть создан атрибут allocated в котором будет храниться список с адресами, которые назначены на каком-то устройстве/хосте. Изначально атрибут равен пустому списку:

```
In [8]: print(net1.allocated)
[]
```

Атрибут hosts должен возвращать список IP-адресов, которые входят в сеть, не включая адрес сети и broadcast:

```
In [9]: net1.hosts
Out[9]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.1.1.4', '10.1.1.5', '10.1.1.6']
```

Метод allocate ожидает как аргумент IP-адрес. Указанный адрес должен быть записан в список в атрибуте net1.allocated и удален из списка unassigned:

```
In [10]: net1 = IPv4Network('10.1.1.0/29')

In [11]: print(net1.allocated)
[]

In [12]: net1.allocate('10.1.1.6')

In [13]: net1.allocate('10.1.1.3')

In [14]: print(net1.allocated)
['10.1.1.6', '10.1.1.3']
```

Атрибут unassigned возвращает возвращает список со свободными адресами:

```
In [15]: net1 = IPv4Network('10.1.1.0/29')

In [16]: net1.allocate('10.1.1.4')
    ...: net1.allocate('10.1.1.6')
```

(continues on next page)

(continued from previous page)

```
    ...: net1.allocate('10.1.1.8')
    ...:

In [17]: net1.unassigned
Out[17]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.1.1.5']
```

2

II. Декораторы

6. Closure

Функции 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:

```
In [2]: import time

In [3]: 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:

```
In [4]: 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
```

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)
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>)
```

(continues on next page)

(continued from previous page)

```
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:

```
In [31]: def func1():
    ...:     a = 1
    ...:     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
```

(continues on next page)

(continued from previous page)

```
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:

```
In [40]: def func1():
    ...:     a = 1
    ...:     b = 'line'
    ...:     c = [1, 2, 3]
    ...:
    ...:     def func2():
    ...:         nonlocal a
    ...:         c.append(4)
    ...:         a += 1
    ...:         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

Так как замыкания позволяют сохранять состояние (значения свободных переменных), их можно использовать для создания функции, которая отчасти похожа на класс:

```
def func_as_object(a,b):
    def add():
        return a+b
    def sub():
        return a-b
    def mul():
        return a*b
    def replace():
        pass
    replace.add = add
    replace.sub = sub
    replace.mul = mul
    return replace
```

(continues on next page)

(continued from previous page)

```
In [13]: obj1 = func_as_object(5,2)

In [14]: obj1.add()
Out[14]: 7

In [15]: obj2 = func_as_object(15,2)

In [16]: obj2.add()
Out[16]: 17

In [17]: obj1.add()
Out[17]: 7
```

В таких случая обязательно надо делать внутреннюю функцию которой присваиваются атрибуты и возвращать ее вместо исходной. Если сделать как в примере ниже, все будет работать корректно только с одним объектом:

```
def func_as_object(a,b):
    def add():
        return a+b
    def sub():
        return a-b
    def mul():
        return a*b
    func_as_object.add = add
    func_as_object.sub = sub
    func_as_object.mul = mul
    return func_as_object

In [18]: obj1 = func_as_object(5, 2)

In [19]: obj1.add()
Out[19]: 7
```

Как только добавляется второй объект, атрибуты функции подменяются на другие вложенные функции, которые помнят значения переменных для последнего объекта и первый объект теперь возвращает неправильные значения:

```
In [9]: obj2 = func_as_object(15,2)

In [10]: obj2.add()
Out[10]: 17
```

(continues on next page)

(continued from previous page)

```
In [11]: obj1.add()
Out[11]: 17
```

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'
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

Задание 6.1

Переделать функцию netmiko_ssh таким образом, чтобы при отправке строки “close”, вместо отправки “close” как команды на оборудование, закрывалось соединение к устройству и выводилось сообщение ‘Соединение закрыто’.

Пример работы функции:

```
In [1]: r1 = netmiko_ssh(**device_params)

In [2]: r1('sh clock')
Out[2]: '*08:07:44.267 UTC Thu Oct 17 2019'

In [3]: r1('close')
Соединение закрыто
```

Тест берет значения из словаря device_params в этом файле, поэтому если для заданий используются другие адреса/логины, надо заменить их в словаре.

```
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):
```

(continues on next page)

(continued from previous page)

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


if __name__ == "__main__":
    r1 = netmiko_ssh(**device_params)
    print(r1('sh clock'))
```

Задание 6.2

Создать функцию count_total, которая вычисляет сумму потраченную на категорию товаров. После вызова функции count_total, должна возвращаться внутренняя функция. При вызове внутренней функции надо передавать аргумент - число. Как результат должна возвращаться текущая сумма чисел.

Пример использования функции count_total:

```
In [2]: books = count_total()

In [3]: books(25)
Out[3]: 25

In [4]: books(15)
Out[4]: 40

In [5]: books(115)
Out[5]: 155

In [6]: books(25)
Out[6]: 180

In [7]: toys = count_total()

In [8]: toys(67)
Out[8]: 67

In [9]: toys(17)
Out[9]: 84
```

(continues on next page)

(continued from previous page)

```
In [10]: toys(24)
Out[10]: 108
```

Задание 6.2a

Изменить функцию count_total из задания 6.2a. После вызова функции count_total, должна быть доступна возможность обращаться к атрибуту buy и передавать ему аргумент - число. Как результат должна возвращаться текущая сумма чисел.

Пример использования функции count_total:

```
In [2]: books = count_total()

In [3]: books.buy(25)
Out[3]: 25

In [4]: books.buy(15)
Out[4]: 40

In [5]: books.buy(115)
Out[5]: 155

In [6]: books.buy(25)
Out[6]: 180

In [7]: toys = count_total()

In [8]: toys.buy(67)
Out[8]: 67

In [9]: toys.buy(17)
Out[9]: 84

In [10]: toys.buy(24)
Out[10]: 108
```

Задание 6.3

Создать функцию queue, которая работает как очередь. После вызова функции queue, должна быть доступна возможность обращаться к атрибутам:

- put - добавляет элемент в очередь

- get - удаляет элемент с начала очереди и возвращает None, если элементы закончились

Пример работы функции queue:

```
In [2]: tasks = queue()

In [3]: tasks.put('a')

In [4]: tasks.put('b')

In [5]: tasks.put('c')

In [6]: tasks.get()
Out[6]: 'a'

In [7]: tasks.get()
Out[7]: 'b'

In [8]: tasks.get()
Out[8]: 'c'

In [9]: tasks.get()

In [10]: tasks.get()
```

7. Декораторы

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.

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()

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()
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
```

(continues on next page)

(continued from previous page)

```
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:

```
In [5]: from functools import wraps

In [6]: 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
```

(continues on next page)

(continued from previous page)

```
   ...: 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.

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

(continues on next page)

(continued from previous page)

```
@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)
******************************
------------------------------
==============================
Out[30]: 9

In [31]: @equals
    ...: @lines
    ...: @stars
    ...: def func(a, b):
    ...:     return a + b
    ...:

In [32]: func(4,5)
==============================
------------------------------
******************************
Out[32]: 9

In [33]: def func(a, b):
    ...:     return a + b
    ...: func = equals(lines(stars(func)))

In [34]: func(4,5)
==============================
------------------------------
```

(continues on next page)

(continued from previous page)

```
****************************
Out[34]: 9
```

Для некоторых декораторов порядок важен и тогда он будет указан в документации. Например, декоратор abstractmethod должен стоять первым над методом (быть самым внутренним):

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, ...):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...
```

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


@debugger_with_args
```

(continues on next page)

(continued from previous page)

```
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
```

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


@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)
```

(continues on next page)

(continued from previous page)

```
@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
    ...:
```

(continues on next page)

(continued from previous page)

```
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']

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)
```

(continues on next page)

(continued from previous page)

```
      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
```

(continues on next page)

(continued from previous page)

```
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


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
```

(continues on next page)

(continued from previous page)

```
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:
        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:
```

(continues on next page)

(continued from previous page)

```
        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')
    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:
```

(continues on next page)

(continued from previous page)

```
        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))
```

Декоратор класса

```
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
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

Задание 7.1

Создать декоратор timecode, который засекает время выполнения декорируемой функции и выводит время на стандартный поток вывода. Декоратор должен работать с любой функцией.

Проверить работу декоратора на функции send_show_command.

Пример вывода:

```
In [3]: @timecode
   ...: def send_show_command(params, command):
   ...:     with ConnectHandler(**params) as ssh:
   ...:         ssh.enable()
   ...:         result = ssh.send_command(command)
   ...:     return result
   ...:

In [4]: print(send_show_command(device_params, 'sh clock'))
>>> Функция выполнялась: 0:00:05.527703
*13:02:49.080 UTC Mon Feb 26 2018
```

Тест берет значения из словаря device_params в этом файле, поэтому если для заданий используются другие адреса/логины, надо заменить их в словаре.

```
from netmiko import ConnectHandler

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

(continues on next page)

(continued from previous page)

```
    'secret': 'cisco'
}


def send_show_command(params, command):
    with ConnectHandler(**params) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
    return result


if __name__ == "__main__":
    print(send_show_command(device_params, 'sh clock'))
```

Задание 7.2

Переделать декоратор all_args_str таким образом, чтобы он проверял не только позиционные аргументы, но и ключевые тоже.

```
In [2]: concat_str(str1='b', str2='a')
Out[2]: 'ba'

In [3]: concat_str(str1='b', str2=1)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-08af972add0a> in <module>
----> 1 concat_str(str1='b', str2=1)
...
ValueError: Все аргументы должны быть строками


In [4]: concat_str('b', 1)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-864f6fda8c8b> in <module>
----> 1 concat_str('b', 1)
...
ValueError: Все аргументы должны быть строками
```

Задание 7.3

Создать декоратор add_verbose, который добавляет в функцию дополнительный параметр verbose. Когда параметру передано значение True, на стандартный поток вывода должна отображаться информация о вызове функции и ее аргументах (пример работы декоратора показан ниже).

По умолчанию, значение параметра должно быть равным False.

Проверить работу декоратора на функции send_show_command.

Пример вывода:

```
In [3]: @add_verbose
   ...: def send_show_command(params, command):
   ...:     with ConnectHandler(**params) as ssh:
   ...:         ssh.enable()
   ...:         result = ssh.send_command(command)
   ...:     return result
   ...:

In [4]: print(send_show_command(device_params, 'sh clock', verbose=True))
Вызываем send_show_command
Позиционные аргументы: ({'device_type': 'cisco_ios', 'ip': '192.168.100.1',
↪'username': 'cisco', 'password': 'cisco', 'secret': 'cisco'}, 'sh clock')
*14:01:07.353 UTC Mon Feb 26 2018

In [5]: print(send_show_command(device_params, 'sh clock', verbose=True))
Вызываем send_show_command
Позиционные аргументы: ({'device_type': 'cisco_ios', 'ip': '192.168.100.1',
↪'username': 'cisco', 'password': 'cisco', 'secret': 'cisco'}, 'sh clock')
*15:09:45.152 UTC Fri Oct 18 2019

In [6]: print(send_show_command(device_params, command='sh clock', verbose=True))
Вызываем send_show_command
Позиционные аргументы: ({'device_type': 'cisco_ios', 'ip': '192.168.100.1',
↪'username': 'cisco', 'password': 'cisco', 'secret': 'cisco'},)
Ключевые аргументы: {'command': 'sh clock'}
*15:10:09.222 UTC Fri Oct 18 2019

In [7]: print(send_show_command(params=device_params, command='sh clock',␣
↪verbose=True))
Вызываем send_show_command
Ключевые аргументы: {'params': {'device_type': 'cisco_ios', 'ip': '192.168.100.1',
↪ 'username': 'cisco', 'password': 'cisco', 'secret': 'cisco'}, 'command': 'sh␣
↪clock'}
```

(continues on next page)

(continued from previous page)

```
*15:10:28.524 UTC Fri Oct 18 2019

In [8]: print(send_show_command(device_params, 'sh clock', verbose=False))
*14:01:18.141 UTC Mon Feb 26 2018
```

Тест берет значения из словаря device_params в этом файле, поэтому если для заданий используются другие адреса/логины, надо заменить их в словаре.

Задание 7.4

Создать декоратор retry, который выполняет декорируемую функцию повторно, заданное количество раз, если результат функции не был истинным.

Пример работы декоратора:

```
In [2]: @retry(times=3)
   ...: def send_show_command(device, show_command):
   ...:     print('Подключаюсь к', device['ip'])
   ...:     try:
   ...:         with ConnectHandler(**device) as ssh:
   ...:             ssh.enable()
   ...:             result = ssh.send_command(show_command)
   ...:         return result
   ...:     except SSHException:
   ...:         return None
   ...:

In [3]: send_show_command(device_params, 'sh clock')
Подключаюсь к 192.168.100.1
Out[3]: '*14:22:01.566 UTC Mon Mar 5 2018'

In [4]: device_params['password'] = '123123'

Обратите внимание, что если указано, что повторить попытку надо 3 раза,
то это три раза в дополнение к первому, то есть все подключение будет 4 раза:
In [5]: send_show_command(device_params, 'sh clock')
Подключаюсь к 192.168.100.1
Подключаюсь к 192.168.100.1
Подключаюсь к 192.168.100.1
Подключаюсь к 192.168.100.1
```

Тест берет значения из словаря device_params в этом файле, поэтому если для заданий используются другие адреса/логины, надо заменить их в словаре.

Задание 7.4a

Переделать декоратор retry из задания 7.4: добавить параметр delay, который контролирует через какое количество секунд будет выполняться повторная попытка.

Пример работы декоратора:

```
In [2]: @retry(times=3, delay=5)
    ..: def send_show_command(device, show_command):
    ..:     print('Подключаюсь к', device['ip'])
    ..:     try:
    ..:         with ConnectHandler(**device) as ssh:
    ..:             ssh.enable()
    ..:             result = ssh.send_command(show_command)
    ..:         return result
    ..:     except (NetMikoAuthenticationException, NetMikoTimeoutException):
    ..:         return None
    ..:

In [3]: send_show_command(device_params, 'sh clock')
Подключаюсь к 192.168.100.1
Out[4]: '*16:35:59.723 UTC Fri Oct 18 2019'

In [5]: device_params['password'] = '123123'

In [6]: send_show_command(device_params, 'sh clock')
Подключаюсь к 192.168.100.1
Повторное подключение через 5 сек
Подключаюсь к 192.168.100.1
Повторное подключение через 5 сек
Подключаюсь к 192.168.100.1
Повторное подключение через 5 сек
Подключаюсь к 192.168.100.1
```

Тест берет значения из словаря device_params в этом файле, поэтому если для заданий используются другие адреса/логины, надо заменить их в словаре.

Задание 7.5

Создать декоратор count_calls, который считает сколько раз декорируемая функция была вызвана. При вызове функции должно отображаться количество вызовов этой функции.

Пример работы декоратора:

```
In [11]: @count_calls
    ...: def f1():
    ...:     return True
    ...:

In [12]: @count_calls
    ...: def f2():
    ...:     return False
    ...:

In [14]: for _ in range(5):
    ...:     f1()
    ...:
Всего вызовов: 1
Всего вызовов: 2
Всего вызовов: 3
Всего вызовов: 4
Всего вызовов: 5

In [15]: for _ in range(5):
    ...:     f2()
    ...:
Всего вызовов: 1
Всего вызовов: 2
Всего вызовов: 3
Всего вызовов: 4
Всего вызовов: 5

In [16]: for _ in range(5):
    ...:     f1()
    ...:
Всего вызовов: 6
Всего вызовов: 7
Всего вызовов: 8
Всего вызовов: 9
Всего вызовов: 10
```

Задание 7.5a

Переделать декоратор count_calls из задания 7.5. Вместо вывода количества вызовов на стандартный поток вывода, надо записать его в атрибут total_calls.

Пример работы декоратора:

```
In [10]: @count_calls
    ...: def f1():
    ...:     return False
    ...:

In [11]: @count_calls
    ...: def f2():
    ...:     return False
    ...:

In [12]: for _ in range(5):
    ...:     f1()
    ...:

In [13]: for _ in range(5):
    ...:     f2()
    ...:

In [14]: for _ in range(5):
    ...:     f1()
    ...:

In [15]: f1.total_calls
Out[15]: 10

In [16]: f2.total_calls
Out[16]: 5
```

Задание 7.6

Создать декоратор total_order, который добавляет к классу методы:

- __ge__ - операция >=
- __ne__ - операция !=
- __le__ - операция <=
- __gt__ - операция >

Декоратор total_order полагается на то, что в классе уже определены методы:

- __eq__ - операция ==
- __lt__ - операция <

Если методы __eq__ и __lt__ не определены, сгенерировать исключение ValueError при декорации.

Проверить работу декоратора можно на примере класса IPAddress. Определение класса нельзя менять, можно только декорировать. Декоратор не должен использовать переменные класса IPAddress. Для работы методов должны использоваться только существующие методы __eq__ и __lt__. Декоратор должен работать и с любым другим классом у которого есть методы __eq__ и __lt__.

Пример проверки методов с классом IPAddress после декорирования:

```
In [4]: ip1 = IPAddress('10.10.1.1')

In [5]: ip2 = IPAddress('10.2.1.1')

In [6]: ip1 < ip2
Out[6]: False

In [7]: ip1 > ip2
Out[7]: True

In [8]: ip1 >= ip2
Out[8]: True

In [9]: ip1 <= ip2
Out[9]: False

In [10]: ip1 == ip2
Out[10]: False

In [11]: ip1 != ip2
Out[11]: True
```

3

III. Генераторы

8. 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
```

(continues on next page)

(continued from previous page)

```
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:
   ...:             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:
```

(continues on next page)

(continued from previous page)

```
                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()
```

(continues on next page)

(continued from previous page)

```
    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

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

Задание 8.1

Создать генератор get_ip_from_cfg, который ожидает как аргумент имя файла, в котором находится конфигурация устройства и возвращает все IP-адреса, которые настроены на интерфейсах.

Генератор должен обрабатывать конфигурацию и возвращать кортеж на каждой итерации: * первый элемент кортежа - IP-адрес * второй элемент кортежа - маска

Например: (‘10.0.1.1’, ‘255.255.255.0’)

Проверить работу генератора на примере файла config_r1.txt.

Задание 8.1a

Создать генератор get_intf_ip, который ожидает как аргумент имя файла, в котором находится конфигурация устройства и возвращает все интерфейсы и IP-адреса, которые настроены на интерфейсах.

Генератор должен обрабатывать конфигурацию и возвращать кортеж на каждой итерации:

- первый элемент кортежа - имя интерфейса
- второй элемент кортежа - IP-адрес
- третий элемент кортежа - маска

Например: `('FastEthernet', '10.0.1.1', '255.255.255.0')`

Проверить работу генератора на примере файла config_r1.txt.

Задание 8.1b

Создать генератор get_intf_ip_from_files, который ожидает как аргумент произвольное количество файлов с конфигурацией устройств и возвращает интерсейсы и IP-адреса, которые настроены на интерфейсах.

Генератор должен обрабатывать конфигурацию и возвращать словарь для каждого файла на каждой итерации:

- ключ - hostname
- значение словарь, в котором:
  - ключ - имя интерфейса
  - значение - кортеж с IP-адресом и маской

Пример:

```
{'r1': {'FastEthernet0/1': ('10.0.1.1', '255.255.255.0'),
        'FastEthernet0/2': ('10.0.2.2', '255.255.255.0')}}
```

Проверить работу генератора на примере конфигураций config_r1.txt и config_r2.txt.

Задание 8.2

Создать генератор read_file_in_chunks, который считывает файл по несколько строк.

Генератор ожидает как аргумент имя файла и количество строк, которые нужно считать за раз и должен возвращать указанное количество строк одной строкой на каждой итерации.

Проверить работу генератора на примере файла config_r1.txt.

Убедиться, что если в последней итерации строк меньше, чем в указанном аргументе, не возникает исключения.

Ограничение: нельзя использовать функции из модуля itertools.

Пример использования функции:

```
In [1]: g = read_file_in_chunks('config_r1.txt', 10)

In [2]: next(g)
Out[2]: 'Current configuration : 4052 bytes\n!\n! Last configuration change at␣
↪13:13:40 UTC Tue Mar 1 2016\nversion 15.2\nno service timestamps debug␣
↪uptime\nno service timestamps log uptime\nno service password-encryption\n!
↪\nhostname PE_r1\n!\n'
```

Задание 8.3

Создать генератор filter_data_by_attr, который фильтрует данные на основании указанного атрибута и значения.

Аргументы генератора:

- итерируемый объект
- имя атрибута
- значение атрибута

Заменить генераторы filter_by_nexthop и filter_by_mask генератором filter_data_by_attr в коде ниже. Проверить работу генератора на объектах Route. Генератор не должен быть привязан к конкретным объектам, то есть должен работать не только с экземплярами класса Route.

Пример использования функции:

```
In [1]: import csv
   ...: from collections import namedtuple
   ...:
   ...: f = open('rib_table.csv')
   ...: reader = csv.reader(f)
   ...:
   ...: headers = next(reader)
   ...: Route = namedtuple("Route", headers)
   ...: route_tuples = map(Route._make, reader)
   ...:

In [2]: nhop_23 = filter_data_by_attr(route_tuples, 'nexthop', '200.219.145.23')

In [3]: mask_20 = filter_data_by_attr(nhop_23, 'netmask', '20')

In [4]: next(mask_20)
Out[4]: Route(status='*>', network='23.36.48.0', netmask='20', nexthop='200.219.
↪145.23', metric='NA', locprf='NA', weight='0', path='53242 12956 2914', origin=
↪'i')

In [5]: next(mask_20)
Out[5]: Route(status='*>', network='23.36.64.0', netmask='20', nexthop='200.219.
↪145.23', metric='NA', locprf='NA', weight='0', path='53242 12956 1299 20940',␣
↪origin='i')
```

Начальный код:

```
import csv
from collections import namedtuple


def filter_by_nexthop(iterable, nexthop):
    for line in iterable:
        if line[3] == nexthop:
            yield line


def filter_by_mask(iterable, mask):
    for line in iterable:
        if line[2] == mask:
            yield line


if __name__ == "__main__":
    with open('rib_table.csv') as f:
        reader = csv.reader(f)
        headers = next(reader)
        Route = namedtuple("Route", headers)
        route_tuples = map(Route._make, reader)

        nhop_23 = filter_by_nexthop(route_tuples, '200.219.145.23')
        mask_20 = filter_by_mask(nhop_23, '20')
```

Задание 8.4

Переделать код функции send_show_command_to_devices таким образом, чтобы она была генератором и возвращала вывод с одного устройства на каждой итерации.

Переделать соответственно код, который вызывает send_show_command_to_devices таким образом, чтобы результат, который генерирует send_show_command_to_devices записывался в файл.

Проверить работу генератора на устройствах из файла devices.yaml. Для этого задания нет теста!

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

from netmiko import ConnectHandler
import yaml
```

(continues on next page)

(continued from previous page)

```
def send_show_command(device, command):
    with ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        prompt = ssh.find_prompt()
    return f"{prompt}{command}\n{result}\n"


def send_show_command_to_devices(devices, command, filename, limit=3):
    with ThreadPoolExecutor(max_workers=limit) as executor:
        results = executor.map(send_show_command, devices, repeat(command))
    with open(filename, 'w') as f:
        for output in results:
            f.write(output)

if __name__ == "__main__":
    command = "sh ip int br"
    with open('devices.yaml') as f:
        devices = yaml.load(f)
    send_show_command_to_devices(devices, command, 'result.txt')
```

9. 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')
```

(continues on next page)

(continued from previous page)

```
(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
```

(continues on next page)

(continued from previous page)

```
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)

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)

```

(continues on next page)

(continued from previous page)

```
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'),
 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')]
```

(continues on next page)

(continued from previous page)

```
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'),
 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')],
```

(continues on next page)

(continued from previous page)

```
 '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+),')

    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]]

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]
```

(continues on next page)

(continued from previous page)

```
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')
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'),
```

(continues on next page)

(continued from previous page)

```
   ...:         ('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',
 '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):
    ...:             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>
```

(continues on next page)

(continued from previous page)

```
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')
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
```

4

IV. Основы asyncio

10. Основы модуля asyncio

Модуль asyncio можно разделить на две части: высокоуровневый интерфейс для пользователей, которые пишут программы и низкоуровневый интерфейс для авторов модулей, библиотек и фреймворков на основе asyncio. В книге рассматривается только первая часть и чаще всего, вторая не понадобится.

Основная причина использования asyncio - улучшить время работы программы, уменьшив время ожидания ответа от операций ввода-вывода. В эти моменты будет выполняться переключение на другие задачи.

Отличия от многопоточной работы:

- при работе с потоками, планировщик может прервать работу потока в любой момент, не всегда это “удобный” момент, поэтому надо явно указывать, что в какие-то моменты прерывать поток нельзя.
- при использовании asyncio работа идет в одном потоке
- при работы с сопрограммами, мы явно указываем в каком месте надо сделать переключение
- await приостанавливает работу текущей сопрограммы и вызывает указанный объект. То есть, если в текущей сопрограмме написано `await run_coro`, текущая сопрограмма останавливается и планирует запуск сопрограммы run_coro в цикле событий

Терминология

- цикл событий (event loop) - менеджер управляющий различными задачами и сопрограммами
- сопрограмма (coroutine) в asyncio - специальный тип функций, которые создаются со словом async перед определением.
- future - это объект, который представляет отложенное вычисление.
- задача (task) - отвечает за управление работой сопрограммы, запрашивает статус сопрограммы. Является подклассом Future.

Awaitables

Awaitables (ожидаемые объекты) - это объекты, которые можно использовать в выражении вместе с `await`. Три базовых типа awaitables:

- сопрограммы (coroutines)
- задачи (tasks)

- future

Coroutine (сопрограмма)

Как и с генераторами, различают:

- функцию сопрограмму - функция, которая создается с помощью `async def`
- объект сопрограмму - объект, который возвращается при вызове функции сопрограммы

Функции-сопрограммы возвращает объекты сопрограммы, и которые запускаются менеджером (циклом событий). Сопрограмма может периодически прерывать выполнение и отдавать управление менеджеру, но при этом она не теряет состояние.

Task (задача)

Объекты класса Task используются для запуска сопрограмм в цикле событий и для отслеживания их состояния. Как только сопрограмма “обернута” в Task, например, с помощью функции `asyncio.create_task`, сопрограмма автоматически запущена для выполнения.

asyncio.Future

Future - это специальный низкоуровневый объект, который представляет отложенное вычисление асинхронных операций. Чаще всего, при работе с модулем asyncio, нет необходимости создавать Future напрямую, но некоторые функции могут возвращать Future. Task является подклассом Future. Менеджер может следить за future и ожидать их завершения.

Сопрограммы и задачи

Создание сопрограммы (coroutine):

```
import asyncio

async def main():
    print(f'Start {datetime.now()}')
    await asyncio.sleep(3)
    print(f'End   {datetime.now()}')


In [6]: coro = main()

In [7]: coro
Out[7]: <coroutine object main at 0xb449fdac>
```

Как и с генераторами, различают:

- функцию сопрограмму - функция, которая создается с помощью `async def`
- объект сопрограмму - объект, который возвращается при вызове функции сопрограммы

Создать сопрограмму недостаточно для того чтобы она запускалась параллельно с другими сопрограммами - для управления сопрограммами нужен менеджер - event loop. Также по умолчанию в сопрограмме код выполняется последовательно и надо явно указывать в каких местах можно переключаться - await.

Запустить сопрограмму можно несколькими способами:

- asyncio.run
- await
- asyncio.create_task

asyncio.run

Функция asyncio.run запускает сопрограмму и возвращает результат:

```
asyncio.run(coro, *, debug=False)
```

Функция asyncio.run всегда создает новый цикл событий и закрывает его в конце. В идеале, функция asyncio.run должна вызываться в программе только один раз и использоваться как основная точка входа. Эту функцию нельзя вызвать, когда в том же потоке запущен другой цикл событий.

Запуск с помощью asyncio.run:

```
In [8]: asyncio.run(coro)
Start 2019-10-30 06:36:03.396389
End   2019-10-30 06:36:06.399606


In [9]: asyncio.run(main())
Start 2019-10-30 06:46:22.162731
End   2019-10-30 06:46:25.166902
```

await

Второй вариант запуска сопрограммы - ожидание ее результата в другой сопрограмме с помощью `await`.

Сопрограмма delay_message выводит указанное сообщение с задержкой:

```
In [10]: from datetime import datetime

In [11]: async def delay_message(delay, message):
    ...:     await asyncio.sleep(delay)
    ...:     print(message)
    ...:
```

Для запуска сопрограммы delay_message, ее результат ожидается в сопрограмме main:

```
In [12]: async def main():
    ...:     print(f'Start {datetime.now()}')
    ...:     await delay_message(4, 'Hello')
    ...:     await delay_message(2, 'world')
    ...:     print(f'End   {datetime.now()}')
    ...:

In [13]: asyncio.run(main())
Start 2019-10-30 06:29:43.828145
Hello
world
End   2019-10-30 06:29:49.835494
```

Обратите внимание на время выполнения main - в данном случае сопрограммы выполнились последовательно и суммарное время 6 секунд.

asyncio.create_task

Еще один вариант запуска сопрограммы - это создание задачи (task). Обернуть сопрограмму в задачу и запланировать ее выполнение можно с помощью функции asyncio.create_task. Она возвращает объект Task, который можно ожидать с await, как и сопрограммы.

```
asyncio.create_task(coro)
```

Функция asyncio.create_task позволяет запускать сопрограммы одновременно, так как создание задачи означает для цикла, что надо запустить эту сопрограмму при первой возможности.

Пример создания задач:

```
In [42]: async def delay_message(delay, message):
    ...:     print('>>> start delay_message')
    ...:     await asyncio.sleep(delay)
    ...:     print('<<<', message)
    ...:
```

(continues on next page)

(continued from previous page)

```
In [43]: async def main():
    ...:     print(f'Start {datetime.now()}')
    ...:     task1 = asyncio.create_task(delay_message(4, 'Hello'))
    ...:     task2 = asyncio.create_task(delay_message(2, 'world'))
    ...:
    ...:     await task1
    ...:     await task2
    ...:     print(f'End {datetime.now()}')
    ...:

In [44]: asyncio.run(main())
Start 2019-10-30 10:18:39.489131
>>> start delay_message
>>> start delay_message
<<< world
<<< Hello
End 2019-10-30 10:18:43.494321
```

При выполнении строк с созданием задач, выполнение сопрограмм уже запланировано и цикл событий их запустит, как только появится возможность.

```
task1 = asyncio.create_task(delay_message(4, 'Hello'))
task2 = asyncio.create_task(delay_message(2, 'world'))
```

Запуск нескольких awaitables

Тут рассматриваются функции, которые позволяют запускать несколько сопрограмм или задач:

- asyncio.gather
- asyncio.wait
- asyncio.wait_for
- asyncio.as_completed

asyncio.gather

Функция gather запускает на выполнение awaitable объекты, которые перечислены в последовательности aws:

```
asyncio.gather(*aws, loop=None, return_exceptions=False)
```

Если какие-то из объектов являются сопрограммами, они автоматически оборачиваются в задачи и планируются на выполнение уже как объекты Task.

В данном примере функция connect_ssh якобы делает подключение к устройству по SSH и отправляет команду. Все реальные действия пока заменены на asyncio.sleep. В зависимости от числа, которое передается как аргумент, выполнение сопрограмм, которые возвращает функция connect_ssh, занимает разное время. Функция send_command_to_devices создает сопрограммы с помощью map и запускает их на выполнение с помощью asyncio.gather:

```
async def connect_ssh(ip, command):
    print(f'Подключаюсь к {ip}')
    await asyncio.sleep(ip)
    print(f'Отправляю команду {command} на устройство {ip}')
    await asyncio.sleep(1)
    return f"{command} {ip}"


async def send_command_to_devices(ip_list, command):
    coroutines = map(connect_ssh, ip_list, repeat(command))
    result = await asyncio.gather(*coroutines)
    return result
```

Если все объекты отработали корректно, asyncio.gather вернет список со значениями, которые вернули объекты. Порядок значений в списке соответствует порядку объектов:

```
In [2]: ip_list = [5, 2, 3, 7]

In [3]: result = asyncio.run(send_command_to_devices(ip_list, 'test'))
Подключаюсь к 5
Подключаюсь к 2
Подключаюсь к 3
Подключаюсь к 7
Отправляю команду test на устройство 2
Отправляю команду test на устройство 3
Отправляю команду test на устройство 5
Отправляю команду test на устройство 7

In [4]: result
Out[4]: ['test 5', 'test 2', 'test 3', 'test 7']
```

Если return_exceptions равно False (по умолчанию), при возникновении исключения, оно появляется в том месте, где ожидается (await) результат asyncio.gather:

```
async def connect_ssh(ip, command):
    print(f'Подключаюсь к {ip}')
    await asyncio.sleep(ip)
    if ip == 3:
        raise OSError(f'Не могу подключиться к {ip}')
    print(f'Отправляю команду {command} на устройство {ip}')
    await asyncio.sleep(1)
    return f"{command} {ip}"


In [11]: result = asyncio.run(send_command_to_devices(ip_list, 'test'))
Подключаюсь к 5
Подключаюсь к 2
Подключаюсь к 3
Подключаюсь к 7
Отправляю команду test на устройство 2
---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
<ipython-input-11-4c2a35eaf7cd> in <module>
----> 1 result = asyncio.run(send_command_to_devices(ip_list, 'test'))
...

<ipython-input-1-7f470cb98776> in send_command_to_devices(ip_list, command)
     13 async def send_command_to_devices(ip_list, command):
     14     coroutines = map(connect_ssh, ip_list, repeat(command))
---> 15     result = await asyncio.gather(*coroutines)
     16     return result

<ipython-input-10-5e26dce87ca7> in connect_ssh(ip, command)
      3     await asyncio.sleep(ip)
      4     if ip == 3:
----> 5         raise OSError(f'Не могу подключиться к {ip}')
      6     print(f'Отправляю команду {command} на устройство {ip}')
      7     await asyncio.sleep(1)

OSError: Не могу подключиться к 3
```

Если return_exceptions равно True, исключение попадает в список как результат:

```
async def connect_ssh(ip, command):
    print(f'Подключаюсь к {ip}')
    await asyncio.sleep(ip)
    if ip == 3:
        raise OSError(f'Не могу подключиться к {ip}')
```

(continues on next page)

(continued from previous page)

```
    print(f'Отправляю команду {command} на устройство {ip}')
    await asyncio.sleep(1)
    return f"{command} {ip}"


async def send_command_to_devices(ip_list, command):
    coroutines = map(connect_ssh, ip_list, repeat(command))
    result = await asyncio.gather(*coroutines, return_exceptions=True)
    return result


In [14]: result = asyncio.run(send_command_to_devices(ip_list, 'test'))
Подключаюсь к 5
Подключаюсь к 2
Подключаюсь к 3
Подключаюсь к 7
Отправляю команду test на устройство 2
Отправляю команду test на устройство 5
Отправляю команду test на устройство 7

In [15]: result
Out[15]: ['test 5', 'test 2', OSError('Не могу подключиться к 3'), 'test 7']

In [16]: result[2]
Out[16]: OSError('Не могу подключиться к 3')

In [17]: isinstance(result[2], Exception)
Out[17]: True
```

Дополнительные материалы

Документация:

- Модуль asyncio

Статьи:

- Overview of Async IO in Python 3.7. Перевод статьи

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

11. Модули async

- asyncssh
- netdev
- aiofiles
- aiohttp
- async-timeout

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

Задание 11.1

Создать сопрограмму (coroutine) send_config_commands. Сопрограмма должна подключаться по SSH с помощью asyncssh к одному устройству, переходить в режим enable, в конфигурационный режим, выполнять указанные команды, а затем выходить из конфигурационного режима.

Параметры функции:

- host - IP-адрес устройства
- username - имя пользователя
- password - пароль
- enable_password - пароль на режим enable
- config_commands - список команд или одна команда (строка), которые надо выполнить

Функция возвращает строку с результатами выполнения команды:

```
In [1]: import asyncio

In [2]: from task_11_1 import send_config_commands

In [3]: commands = ['interface loopback55', 'ip address 10.5.5.5 255.255.255.255']

In [4]: print(asyncio.run(send_config_commands('192.168.100.1', 'cisco', 'cisco',
↪'cisco', commands)))
conf t
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#interface loopback55
R1(config-if)#ip address 10.5.5.5 255.255.255.255
R1(config-if)#end
```

(continues on next page)

(continued from previous page)

```
R1#

In [5]: asyncio.run(send_config_commands(*r1, config_commands=commands))
Out[5]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.
↪\r\nR1(config)#interface loopback55\r\nR1(config-if)#ip address 10.5.5.5 255.
↪255.255.255\r\nR1(config-if)#end\r\nR1#'
```

Запустить сопрограмму и проверить, что она работает корректно. При необходимости можно создавать дополнительные функции.

Для заданий в этом разделе нет тестов!

```
import asyncio
import asyncssh


r1 = {'host': '192.168.100.1',
      'username': 'cisco',
      'password': 'cisco',
      'enable_password': 'cisco'}
```

Задание 11.2

Создать сопрограмму (coroutine) configure_devices. Сопрограмма должна настраивать одни и те же команды на указанных устройствах с помощью asyncssh. Все устройства должны настраиваться параллельно.

Параметры функции:

- devices - список словарей с параметрами подключения к устройствам
- config_commands - команды конфигурационного режима, которые нужно отправить на каждое устройство

Функция возвращает список строк с результатами выполнения команды на каждом устройстве. Запустить сопрограмму и проверить, что она работает корректно с устройствами в файле devices.yaml и командами в списке commands.

При необходимости, можно использовать функции из предыдущих заданий и создавать дополнительные функции.

Для заданий в этом разделе нет тестов!

```
commands = ['router ospf 55',
            'auto-cost reference-bandwidth 1000000',
            'network 0.0.0.0 255.255.255.255 area 0']
```

Задание 11.3

Создать сопрограмму (coroutine) config_device_and_check. Сопрограмма должна подключаться по SSH с помощью netdev к одному устройству, переходить в режим enable, в конфигурационный режим, выполнять указанные команды, а затем выходить из конфигурационного режима. После настройки команд, функция должна проверять, что они настроены корректно. Для проверки используется словарь (пояснение ниже). Если проверка не прошла, должно генерироваться исключение ValueError с текстом на каком устройстве не прошла проверка. Если проверка прошла, функция должна возвращать строку с результатами выполнения команды.

Параметры функции:

- device - словарь с параметрами подключения к устройству
- config_commands - список команд или одна команда (строка), которые надо выполнить
- check - словарь, который указывает как проверить настройку команд config_commands. По умолчанию значение None.

Словарь, который передается в параметр check должен содержать две пары ключ-значение:

- command - команда, которая используется для проверки конфигурации
- search_line - какая строка должна присутствовать в выводе команды command

Запустить сопрограмму и проверить, что она работает корректно одним из устройств в файле devices_netmiko.yaml и командами в списке commands. Пример команд и словаря для проверки настройки есть в задании.

При необходимости, можно использовать функции из предыдущих заданий и создавать дополнительные функции.

Для заданий в этом разделе нет тестов!

```
commands = ['router ospf 55',
            'auto-cost reference-bandwidth 1000000',
            'network 0.0.0.0 255.255.255.255 area 0']

check_ospf = {'command': 'sh ip ospf',
              'search_line': 'Routing Process "ospf 55" with ID'}
```

Задание 11.4

Создать сопрограмму (coroutine) configure_network_device. Сопрограмма должна подключаться по SSH к одному устройству, переходить в режим enable, в конфигурационный режим, выполнять указанные команды, а затем выходить из конфигурационного режима.

Для подключения должен функция должна использовать модуль netdev, если device_type есть среди поддерживаемых платформ в netdev и использовать asyncssh, если его среди платформ нет. Для проверки второй ситуации можно прямо внутри функции удалить cisco_ios из устройств.

Параметры функции:

- device - словарь с параметрами подключения к устройству
- config_commands - список команд или одна команда (строка), которые надо выполнить

Функция возвращает строку с результатами выполнения команд (как в 11.1).

Как получить платформы netdev:

```
In [3]: netdev.platforms
Out[3]:
['arista_eos',
 'aruba_aos_6',
 'aruba_aos_8',
 'cisco_asa',
 'cisco_ios',
 'cisco_ios_xe',
 'cisco_ios_xr',
 'cisco_nxos',
 'fujitsu_switch',
 'hp_comware',
 'hp_comware_limited',
 'hw1000',
 'juniper_junos',
 'mikrotik_routeros',
 'terminal',
 'ubiquity_edge']
```

Запустить сопрограмму и проверить, что она работает корректно одним из устройств в файле devices_netmiko.yaml и командами в списке commands.

При необходимости, можно использовать функции из предыдущих заданий и создавать дополнительные функции. Для заданий в этом разделе нет тестов!

```
commands = ['router ospf 55',
            'auto-cost reference-bandwidth 1000000',
            'network 0.0.0.0 255.255.255.255 area 0']
```

Задание 11.5

Создать сопрограмму (coroutine) configure_router. Сопрограмма подключается по SSH (с помощью netdev) к устройству и выполняет перечень команд в конфигурационном режиме на основании переданных аргументов.

При выполнении каждой команды, скрипт должен проверять результат на такие ошибки:

- Invalid input detected, Incomplete command, Ambiguous command

Если при выполнении какой-то из команд возникла ошибка, должно генерироваться исключение ValueError с информацией о том, какая ошибка возникла, при выполнении какой команды и на каком устройстве, например: Команда “logging” выполнилась с ошибкой “Incomplete command” на устройстве 192.168.100.1

Параметры функции:

- device - словарь с параметрами подключения к устройству
- config_commands - список команд или одна команда (строка), которые надо выполнить

Функция возвращает строку с результатами выполнения команды.

Примеры команд с ошибками:

```
R1(config)#logging 0255.255.1
                   ^
% Invalid input detected at '^' marker.

R1(config)#logging
% Incomplete command.

R1(config)#a
% Ambiguous command:  "a"
```

Запустить сопрограмму и проверить, что она работает корректно одним из устройств в файле devices_netmiko.yaml.

При необходимости, можно использовать функции из предыдущих заданий и создавать дополнительные функции.

Для заданий в этом разделе нет тестов!

Списки команд с ошибками и без:

```
commands_with_errors = ['logging 0255.255.1', 'logging', 'a']
correct_commands = ['logging buffered 20010', 'ip http server']
```

Задание 11.5a

Создать сопрограмму (coroutine) log_device_configuration. Сопрограмма log_device_configuration должна использовать configure_router из задания 11.5 для настройки оборудования. Оборудование должно настраиваться параллельно.

После настройки оборудования, log_device_configuration должна записывать результаты в log-файл (все результаты записываются в один log-файл):

- если настройка прошла успешно, записать в файл строку "Успешно настроен 192.168.100.1", а затем записать результат, который вернула config_device_and_check
- если настройка не прошла проверку, записать в файл строку "Не получилось настроить 192.168.100.1" и записать сообщение из исключения

Параметры функции log_device_configuration:

- log_file - имя файла, в который будут записываться сообщения (сообщения могут быть в любом порядке)
- devices - список словарей с параметрами подключения к устройствам
- device_commands_map - словарь в котором указано на какое устройство отправлять какие команды. Пример словаря - commands

Пример команд и словаря для проверки настройки есть в задании.

При необходимости, можно использовать функции из предыдущих заданий и создавать дополнительные функции.

Для заданий в этом разделе нет тестов!

```
ospf = ['router ospf 55',
        'auto-cost reference-bandwidth 1000000',
        'network 0.0.0.0 255.255.255.255 area 0']
logging_with_error = 'logging 0255.255.1'
logging_correct = 'logging buffered 20010'

commands = {'192.168.100.2': logging_correct,
            '192.168.100.3': logging_with_error,
            '192.168.100.1': ospf}
```

12. Использование модуля asyncio

- loop
- wait
- cancel
- Генераторы
- list comprehensions
- async with
- async for
- run in thread

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

5

V. Полезные модули

14. Основы pytest

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
```

(continues on next page)

(continued from previous page)

```
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]
---------------------------------------------------------------------------
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:
        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

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 ис-

ключение.

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)
        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
```

(continues on next page)

(continued from previous page)

```
test_check_ip_function.py .                                          [100%]

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

Примеры тестов

Тест функции

```
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
```

```
In [3]: check_passwd('nata', '12345', min_length=3)
Пароль для пользователя nata прошел все проверки
Out[3]: True

In [4]: check_passwd('nata', '12345nata', min_length=3)
Пароль содержит имя пользователя
Out[4]: False

In [5]: check_passwd('nata', '12345nata', min_length=3, check_username=False)
Пароль для пользователя nata прошел все проверки
Out[5]: True

In [6]: check_passwd('nata', '12345nata', min_length=3, check_username=True)
Пароль содержит имя пользователя
Out[6]: False
```

Тест класса

```
import ipaddress

```

(continues on next page)

(continued from previous page)

```
class IPv4Network:
    def __init__(self, network):
        self._net = ipaddress.ip_network(network)
        self.address = str(self._net.network_address)
        self.mask = self._net.prefixlen
        self.allocated = tuple()

    def hosts(self):
        return tuple([str(ip) for ip in self._net.hosts()])

    def allocate(self, ip):
        self.allocated += (ip,)

    def unassigned(self):
        return tuple([ip for ip in self.hosts() if ip not in self.allocated])
```

```
import pytest
import task_1_1
from common_functions import check_class_exists, check_attr_or_method


def test_class_created():
    '''Проверяем, что класс создан'''
    check_class_exists(task_1_1, 'IPv4Network')


def test_attributes_created():
    '''
    Проверяем, что у объекта есть атрибуты:
        address, mask, broadcast, allocated
    '''
    net = task_1_1.IPv4Network('100.7.1.0/26')
    check_attr_or_method(net, attr='address')
    check_attr_or_method(net, attr='mask')
    check_attr_or_method(net, attr='broadcast')
    check_attr_or_method(net, attr='allocated')
    assert net.allocated == tuple(), "По умолчанию allocated должен содержать␣
↪пустой кортеж"

def test_methods_created():
    '''
```

(continues on next page)

(continued from previous page)

```
    Проверяем, что у объекта есть методы:
        allocate, unassigned
    '''
    net = task_1_1.IPv4Network('100.7.1.0/26')
    check_attr_or_method(net, method='allocate')
    check_attr_or_method(net, method='unassigned')

def test_return_types():
    '''Проверяем работу объекта'''
    net = task_1_1.IPv4Network('100.7.1.0/26')
    assert type(net.hosts()) == tuple, "Метод hosts должен возвращать кортеж"
    assert type(net.unassigned()) == tuple, "Метод unassigned должен возвращать␣
↪кортеж"


def test_address_allocation():
    '''Проверяем работу объекта'''
    net = task_1_1.IPv4Network('100.7.1.0/26')
    assert len(net.hosts()) == 62, "В данной сети должно быть 62 хоста"
    assert net.broadcast == '100.7.1.63', "Broadcast адрес для этой сети 100.7.1.
↪63"

    net.allocate('100.7.1.45')
    net.allocate('100.7.1.15')
    net.allocate('100.7.1.60')

    assert len(net.hosts()) == 62, "Метод hosts должен возвращать все хосты"
    assert len(net.allocated) == 3, "Переменная allocated должна содержать 3 хоста
↪"
    assert len(net.unassigned()) == 59, "Метод unassigned должен возвращать на 3␣
↪хоста меньше"
```

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
```

(continues on next page)

(continued from previous page)

```
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

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
```

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
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 ==========================
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
```

(continues on next page)

(continued from previous page)

```
====================== 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>
  <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

```
from check_password_function import check_passwd


def test_password_min_length():
    assert check_passwd('nata', '12345', min_length=3)
    assert not check_passwd('nata', '12345nata', min_length=3)
```

Параметризация:

```
import pytest
from check_password_function import check_passwd

@pytest.mark.parametrize("username,password,min_length,result",[
```

(continues on next page)

(continued from previous page)

```
    ('nata', '12345', 3, True),
    ('nata', '12345nata', 3, False)
])
def test_password_min_length(username, password, min_length, result):
    assert result == check_passwd(username, password, min_length=min_length)
```

Пример из базового курса: https://github.com/pyneng/pyneng-online-may-aug-2019/blob/master/exercises/19_ssh_telnet/tests/test_task_19_2b.py

Fixture

Fixtures - это функции, которые pytest вызывает.

scope:

- function (default)
- module
- session

Источник pyneng-online-may-aug-2019/exercises/26_oop_special_methods/conftest.py

```
import re
import yaml
import pytest
from netmiko import ConnectHandler


@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():
```

(continues on next page)

(continued from previous page)

```
    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
```

Источник pyneng-online-may-aug-2019/exercises/19_ssh_telnet/conftest.py

```
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]
        #options = {'timeout': 5, 'fast_cli': True}
        r1.update(options)
    return r1


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


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


@pytest.fixture(scope='module')
def first_router_wrong_ip(first_router_from_devices_yaml):
    r1 = first_router_from_devices_yaml.copy()
```

(continues on next page)

(continued from previous page)

```
    r1['ip'] = 'unreachable'
    return r1
```

Встроенные fixture

capsys

```
import pytest
import task_19_2a
import sys
sys.path.append('..')

from common_functions import check_function_exists


def test_functions_created():
    check_function_exists(task_19_2a, 'send_config_commands')


def test_function_return_value(capsys, r1_test_connection,
                               first_router_from_devices_yaml):
    test_commands = [
        'logging 10.255.255.1', 'logging buffered 20010', 'no logging console'
    ]
    correct_return_value = r1_test_connection.send_config_set(test_commands)
    return_value = task_19_2a.send_config_commands(
        first_router_from_devices_yaml, test_commands)
    # проверяем возвращаемое значение
    assert return_value != None, "Функция ничего не возвращает"
    assert type(return_value) == str, "Функция должна возвращать строку"
    assert return_value == correct_return_value, "Функция возвращает неправильное␣
↪значение"

    # по умолчанию, verbose должно быть равным True
    # и на stdout должно выводиться сообщение
    correct_stdout = f'{r1_test_connection.host}'
    out, err = capsys.readouterr()
    assert out != '', "Сообщение об ошибке не выведено на stdout"
    assert correct_stdout in out, "Выведено неправильное сообщение об ошибке"

    # проверяем, что с verbose=False вывода в stdout нет
```

(continues on next page)

(continued from previous page)

```
    return_value = task_19_2a.send_config_commands(
        first_router_from_devices_yaml, test_commands, verbose=False)
    correct_stdout = ''
    out, err = capsys.readouterr()
    assert out == correct_stdout,\
            "Сообщение об ошибке не должно выводиться на stdout, когда␣
↪verbose=False"
```

tmpdir

```
import pytest
import task_20_2
import sys
sys.path.append('..')

from common_functions import check_function_exists



def test_functions_created():
    check_function_exists(task_20_2, 'send_show_command_to_devices')


def test_function_return_value(three_routers_from_devices_yaml,
                               r1_r2_r3_test_connection, tmpdir):
    command = 'sh ip int br'
    out1, out2, out3 = [r.send_command(command) for r in r1_r2_r3_test_connection]
    dest_filename = tmpdir.mkdir("test_tasks").join("task_20_2.txt")

    return_value = task_20_2.send_show_command_to_devices(
        devices=three_routers_from_devices_yaml,
        command=command, filename=dest_filename, limit=3)
    assert return_value == None, "Функция должна возвращать None"

    dest_file_content = dest_filename.read().strip()

    # проверяем, что вывод с каждого устройства есть в файле
    assert out1.strip() in dest_file_content, "В итоговом файле нет вывода с␣
↪первого устройства"
    assert out2.strip() in dest_file_content, "В итоговом файле нет вывода со␣
↪второго устройства"
```

(continues on next page)

(continued from previous page)

```
    assert out3.strip() in dest_file_content, "В итоговом файле нет вывода с␣
↪третьего устройства"
```

```
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 add_user_to_users_file(user, users_filename='users.txt'):
    while True:
        passwd = input(f'Введите пароль для пользователя {user}: ')
        if check_passwd(user, passwd):
            break
    with open(users_filename, 'a') as f:
        f.write(f'{user},{passwd}\n')
```

Подделка функций (Mocking)

Код

```
import getpass

def check_passwd(min_length=8, check_username=True):
    username = input('Username: ')
    password = getpass.getpass('Password: ')
    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
```

Тест:

```
from check_password_function_input import check_passwd


def test_password_min_length():
    assert check_passwd(min_length=3)
```

Результат:

```
$ pytest test_check_password_input.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 1 item

test_check_password_input.py F                                   [100%]

============================== FAILURES ==============================
_________________________ test_password_min_length _________________________

    def test_password_min_length():
>       assert check_passwd(min_length=3)

test_check_password_input.py:5:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
check_password_function_input.py:2: in check_passwd
    username = input('Username: ')
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <_pytest.capture.DontReadFromInput object at 0xb68f424c>
args = ()

    def read(self, *args):
>       raise IOError("reading from stdin while output is captured")
E       OSError: reading from stdin while output is captured

/home/vagrant/venv/pyneng-py3-7/lib/python3.7/site-packages/_pytest/capture.
↪py:706: OSError
---------------------- Captured stdout call ----------------------
Username:
======================== 1 failed in 0.09s =========================
```

monkeypatch

```
from check_password_function_input import check_passwd


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

Проверка

```
$ pytest test_check_password_input.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 1 item

test_check_password_input.py .                                   [100%]

======================== 1 passed in 0.03s =========================
```

Проверка нескольких сценариев с parametrize

```
import pytest
from check_password_function_input import check_passwd


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

Проверка:

```
$ pytest test_check_password_input.py
======================= test session starts ========================
```

(continues on next page)

(continued from previous page)

```
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 2 items

test_check_password_input.py ..                                      [100%]

======================== 2 passed in 0.03s =========================
```

conftest

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

Запуск 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 0 items / 5 errors

==================================== ERRORS ====================================
_____________ ERROR collecting tests/test_check_ip_function.py _____________
ImportError while importing test module '/home/vagrant/repos/advanced-pyneng-1/
↪advpyneng-online-oct-nov-2019/examples/14_pytest_basics/tests/test_check_ip_
↪function.py'.
Hint: make sure your test modules/packages have valid Python names.
```

(continues on next page)

(continued from previous page)

```
Traceback:
tests/test_check_ip_function.py:1: in <module>
    from check_ip_functions import check_ip
E   ModuleNotFoundError: No module named 'check_ip_functions'
!!!!!!!!!!!!!!!!!!!! Interrupted: 5 errors during collection !!!!!!!!!!!!!!!!!!!!
=============================== 5 error in 0.18s ===============================
```

Достаточно создать пустой файл conftest.py и тесты заработают

```
$ touch conftest.py
$ 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 13 items

tests/test_check_ip_function.py .                                        [  7%]
tests/test_check_password.py ...                                         [ 30%]
tests/test_check_password_input.py ..                                    [ 46%]
tests/test_check_password_parametrize.py ..                              [ 61%]
tests/test_ipv4_network.py .....                                         [100%]

============================== 13 passed in 0.09s ==============================
```

Дополнительные возможности

pytest.raises

```
import pytest
import task_27_2a
from netmiko.cisco.cisco_ios import CiscoIosBase
import sys
sys.path.append('..')

from common_functions import check_class_exists, check_attr_or_method


def test_class_created():
    check_class_exists(task_27_2a, 'MyNetmiko')
```

(continues on next page)

(continued from previous page)

```
def test_class_inheritance(first_router_from_devices_yaml):
    r1 = task_27_2a.MyNetmiko(**first_router_from_devices_yaml)
    assert isinstance(r1, CiscoIosBase), "Класс MyNetmiko должен наследовать␣
↪CiscoIosBase"
    check_attr_or_method(r1, method='send_command')
    with pytest.raises(task_27_2a.ErrorInCommand) as excinfo:
        return_value = r1.send_command('sh ip br')
    r1.disconnect()
```

pytest-html

https://github.com/pytest-dev/pytest-html

Использование pytest для тестирования сети

```
import pytest
import subprocess


def ping_ip(ip):
    result = subprocess.run(f'ping -c 2 {ip}', shell=True)
    return result.returncode == 0


@pytest.mark.parametrize('ip',
                         ['192.168.100.1', '192.168.100.2', '192.168.100.3'])
def test_ip(ip):
    assert ping_ip(ip)
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

16. Модуль logging

Базовый пример

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
```

Пример вывода информации о потоках:

```
from concurrent.futures import ThreadPoolExecutor
from pprint import pprint
from datetime import datetime
```

(continues on next page)

(continued from previous page)

```
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__':
    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

Вывод на стандартный поток ошибок logging_api_example_1.py

```
import logging

logger = logging.getLogger('My Script')

## 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
По умолчанию вывод идет в stderr и уровень 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
```

(continues on next page)

(continued from previous page)

```
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')
```

(continues on next page)

(continued from previous page)

```
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')
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)
```

(continues on next page)

(continued from previous page)

```
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 - 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):
```

(continues on next page)

(continued from previous page)

```
    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')
    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
```

(continues on next page)

(continued from previous page)

```
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)

## 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
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

15. Основы аннотации типов

Преимущества:

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

Недостатки/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

Основы

Аннотация типов - это дополнительное описание в классах, функциях, переменных, которое указывает какой тип данных должен быть в этом месте. Это новый функционал, который был добавлен в последних версиях Python (Python 3.6+).

При этом указанные типы не проверяются и не форсируются самим Python. Для проверки типов данных надо использовать отдельные модули, например, mypy.

Аннотация функции

Пример аннотации функции:

```
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:
```

(continues on next page)

(continued from previous page)

```
    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
```

Атрибут __annotations__:

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

Аннотация переменных

Пример переменной:

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

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

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

Например, этот функционал используется в Data classes:

```
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)
```

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
```

Аннотация классов

- не пишем аннотацию для self

```
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

    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)

Основы mypy

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)
```

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:
```

(continues on next page)

(continued from previous page)

```
        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

Note: Работает в Python 3.7+ с импортом __future__

Использование имени класса в аннотации внутри этого же класса:

```
from __future__ import annotations
import ipaddress


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
```

pydantic

pydantic использует аннотацию типов для проверки данных.

Пример создания dataclass:

```
In [9]: from dataclasses import dataclass

In [10]: @dataclass
    ...: class Book:
    ...:     title: str
    ...:     price: int
    ...:
```

В этом случае, аннотация переменных используется для создания атрибутов, но при этом тип данных не проверяется и все эти варианты отработают:

```
In [11]: book = Book('Good Omens', price=35)

In [12]: book = Book('Good Omens', price='35')

In [13]: book = Book('Good Omens', price='a')
```

При использовании декоратора dataclass из модуля pydantic, типы проверяются:

```
In [14]: from pydantic.dataclasses import dataclass

In [15]: @dataclass
    ...: class Book:
    ...:     title: str
    ...:     price: int
    ...:

In [16]: book = Book('Good Omens', price=35)

In [17]: book = Book('Good Omens', price='35')

In [18]: book = Book('Good Omens', price='a')
---------------------------------------------------------------------------
ValidationError                           Traceback (most recent call last)
<ipython-input-18-c21f0df3a6ac> in <module>
----> 1 book = Book('Good Omens', price='a')

<string> in __init__(self, title, price)

~/venv/pyneng-py3-7/lib/python3.7/site-packages/pydantic/dataclasses.cpython-37m-
↪i386-linux-gnu.so in pydantic.dataclasses._process_class._pydantic_post_init()

ValidationError: 1 validation error for Book
price
  value is not a valid integer (type=type_error.integer)

In [19]: book = Book('Good Omens', price='35')

In [20]: book.price
Out[20]: 35
```

Примеры использования pydantic

Дополнительные материалы

- Шпаргалка по аннотации типов
- Модуль typing
- pydantic
- Type hinting in PyCharm

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

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

17. Модуль click

Задания

Все задания и вспомогательные файлы можно скачать в репозитории. Если в заданиях раздела есть задания с буквами (например, 5.2a), то лучше выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают или усложняют идею в соответствующем задании без буквы.

Note: Например, в разделе есть задания 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала лучше выполнить задания 5.1, 5.2, 5.3, а затем 5.2a, 5.2b, 5.3a

---

Если задания с буквами получается сделать сразу, лучше делать их по порядку.

18. Модуль pdb

19. Code formatters

20. Collections

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 Interger:
    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
```

(continues on next page)

(continued from previous page)

```
class String:
    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, str):
            raise TypeError('Wrong data type, expected str')
        instance.__dict__[self.name] = value
```

Дескриптор обязательно должен быть указан на уровне класса:

```
class IPAddress:
    mask = Interger('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)
```

(continues on next page)

(continued from previous page)

```
<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):
---> 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 Interger(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)
    ...:
```

(continues on next page)

(continued from previous page)

```
    ...:     @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
    ...:

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>
```

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()
```

(continues on next page)

(continued from previous page)

```
    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


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()
```

(continues on next page)

(continued from previous page)

```
        result += super().send_config_commands(commands)
        result += self.exit_config_mode()
        return result


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__ = ()
```

(continues on next page)

(continued from previous page)

```
    def __int__(self):
        return self._ip

    def __eq__(self, other):
        try:
            return (self._ip == other._ip
                    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)
```

(continues on next page)

(continued from previous page)

```
class IPv4Address(_BaseV4, _BaseAddress):

    """Represent and manipulate single IPv4 Addresses."""

    __slots__ = ('_ip', '__weakref__')

    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.
        """
```