Начало работы с Python

В этом разделе рассматриваются:

- синтаксис Python
- работа в интерактивном режиме
- переменные в Python

Синтаксис Python

Первое, что, как правило, бросается в глаза, если говорить о синтаксисе Python - это то, что отступы имеют значение:

- они определяют, какие выражения попадают в блок кода
- когда блок кода заканчивается

Пример кода Python:

```
a = 10
b = 5

if a > b:
    print("A больше B")
    print(a - b)
else:
    print("B больше или равно A")
    print(b - a)

print("The End")

def open_file(filename):
    print("Reading file", filename)
    with open(filename) as f:
        return f.read()
        print("Done")
```

> Обратите внимание, что тут код показан для демонстрации синтаксиса. В следующих разделах рассматриваются типы данных Python и создание скриптов.

Несмотря на то, что еще не рассматривалась конструкция if/else, всё должно быть понятно.

Python понимает, какие строки относятся к if на основе отступов. Выполнение части `if a > b` заканчивается, когда встречается строка с тем же отступом, что и сама строка `if a > b`.

Аналогично с блоком else.

Вторая особенность Python: после некоторых выражений должно идти двоеточие (например, после if a > b или после else).

Несколько правил и рекомендаций:

- В качестве отступов могут использоваться Tab или пробелы.

    - лучше использовать пробелы, а точнее, настроить редактор так, чтобы Tab был равен 4 пробелам (тогда при использовании Tab будут ставиться 4 пробела)
- Количество пробелов должно быть одинаковым в одном блоке.
    - лучше, чтобы количество пробелов было одинаковым во всем коде
    - популярный вариант - использовать 2-4 пробела (в курсе используются 4 пробела)

> Для того, чтобы проблем с отступами не было, надо сразу настроить редактор таким образом, чтобы отступы делались автоматически.

Еще одна особенность приведенного примера кода: пустые строки. Таким образом код форматируется, чтобы его было проще читать.

Остальные особенности синтаксиса будут показаны в процессе знакомства с структурами данных в Python.

Комментарии

При написании кода часто нужно оставить комментарий, например, чтобы описать особенности работы кода.

Комментарии в Python могут быть однострочными:

```
#Очень важный комментарий
a = 10
b = 5 #Очень нужный комментарий
```

Однострочные комментарии начинаются со знака #.

Обратите внимание, что комментарий может быть и в строке, где находится код, и сам по себе.

При необходимости написать несколько строк с комментариями, чтобы не ставить перед каждой решетку, можно сделать многострочный комментарий:

```
"""Очень важный
и длинный комментарий
"""
a = 10
b = 5
```

Многострочный комментарий может использовать три двойные или три одинарные кавычки.

Комментарии могут использоваться как для того, чтобы комментировать, что происходит в коде, так и для того, чтобы закомментировать временно какое-то выражение.

Интерпретатор Python. iPython

Интерпретатор позволяет получать моментальный отклик на выполненные действия.

Можно сказать, что интерпретатор работает как командная строка сетевых устройств: каждая команда будет выполняться сразу после нажатия enter (по крайней мере, похоже на cisco).

Но для интерпретатора Python есть исключение: более сложные объекты (например, циклы, функции) выполняются только после нажатия enter два раза.

В предыдущем разделе для проверки установки Python вызвался стандартный интерпретатор.

Но, кроме него, в Python есть усовершенствованный интерпретатор iPython (документация iPython).

iPython позволяет намного больше, чем стандартный интерпретатор, который вызывается по команде python.

Несколько примеров (возможности ipython намного шире):

- автопродолжение команд по Tab или подсказка, если вариантов команд несколько
- более структурированный и понятный вывод команд
- автоматические отступы в циклах и других объектах
- история выполнения команд
  - по ней можно передвигаться
  - или посмотреть "волшебной" командой %history

Установить iPython можно с помощью pip (установка в виртуальном окружении):

```
pip install ipython
```

Далее как интерпретатор будет использоваться iPython.

Для знакомства с интерпретатором можно попробовать его использовать как калькулятор:

```
In [1]: 1 + 2
Out[1]: 3

In [2]: 22*45
Out[2]: 990

In [3]: 2**3
Out[3]: 8
```

В iPython ввод и вывод подписаны:

- In - это то, что написал пользователь
- Out - это вывод команды (если он есть)
- Числа после In и Out - это нумерация выполненных команд в текущей сессии iPython

Пример вывода строки:

```
In [4]: print('Hello!')
Hello!
```

Когда в интерпретаторе создается, например, цикл, то внутри цикла приглашение меняется на `...`. Для выполнения цикла и выхода из этого подрежима необходимо дважды нажать Enter:

```
In [5]: for i in range(5):
   ...:     print(i)
   ...:
0
1
2
3
4
```

help

В ipython есть возможность посмотреть help по какому-то объекту, функции или методу:

```
In [1]: help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
...

In [2]: help(str.strip)
Help on method_descriptor:

strip(...)
    S.strip([chars]) -> str

    Return a copy of the string S with leading and trailing
    whitespace removed.
    If chars is given and not None, remove characters in chars instead.
```

Второй вариант:

```
In [3]: ?str
Init signature: str(self, /, *args, **kwargs)
Docstring:
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
Type:           type

In [4]: ?str.strip
Docstring:
S.strip([chars]) -> str

Return a copy of the string S with leading and trailing
whitespace removed.
If chars is given and not None, remove characters in chars instead.
Type:      method_descriptor
```

print

Функция print позволяет вывести информацию на стандартный поток вывода.

Если необходимо вывести строку, то ее нужно обязательно заключить в кавычки (двойные или одинарные). Если же нужно вывести, например, результат вычисления или просто число, то кавычки не нужны:

```
In [6]: print('Hello!')
Hello!

In [7]: print(5*5)
25
```

Если нужно вывести несколько значений, можно перечислить их через запятую:

```
In [8]: print(1*5, 2*5, 3*5, 4*5)
5 10 15 20

In [9]: print('one', 'two', 'three')
one two three
```

> Подробнее о функции print

По умолчанию в конце выражения будет перевод строки. Если необходимо, чтобы после вывода выражения не было перевода строки, надо указать дополнительный аргумент `end`.

По умолчанию он равен `\n`, поэтому к строке или строкам в print добавляется перевод строки.

Например, такое выражение выведет строки 'one' и 'two' в разных строках:

```
In [10]: print('one'), print('two')
one
two
Out[10]: (None, None)
```

Но если в первой функции print указать параметр end равным пустой строке, результат будет таким:

```
In [11]: print('one', end=''), print('two')
onetwo
Out[11]: (None, None)
```

dir

Команда dir() может использоваться для того, чтобы посмотреть, какие атрибуты и методы есть у объекта.

Например, для числа вывод будет таким (обратите внимание на различные методы, которые позволяют делать арифметические операции):

```
In [10]: dir(5)
Out[10]:
['__abs__',
 '__add__',
 '__and__',
 ...
 'bit_length',
 'conjugate',
 'denominator',
 'imag',
 'numerator',
 'real']
```

Для строки:

```
In [11]: dir('hello')
Out[11]:
['__add__',
 '__class__',
 '__contains__',
 ...
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

Если выполнить команду без передачи значения, то она показывает существующие методы, атрибуты и переменные, определенные в текущей сессии интерпретатора:

```
In [12]: dir()
Out[12]:
[ '__builtin__',
 '__builtins__',
 '__doc__',
 '__name__',
 '_dh',
 ...
 '_oh',
 '_sh',
 'exit',
 'get_ipython',
 'i',
 'quit']
```

Пример после создания переменной **a** и функции **test**:

```
In [13]: a = 'hello'

In [14]: def test():
   ....:     print('test')
   ....:

In [15]: dir()
Out[15]:
 ...
 'a',
 'exit',
 'get_ipython',
 'i',
 'quit',
 'test']
```

Magic commands

В IPython есть специальные команды, которые упрощают работу с интерпретатором. Все они начинаются на %.

`%history`

Например, команда `%history` позволяет просмотреть историю текущей сессии:

```
In [1]: a = 10

In [2]: b = 5

In [3]: if a > b:
   ...:     print("A is bigger")
   ...:
A is bigger

In [4]: %history
a = 10
b = 5
if a > b:
    print("A is bigger")
%history
```

Таким образом можно скопировать какой-то блок кода.

`%cpaste`

Еще одна очень полезная волшебная команда `%cpaste`

При вставке кода с отступами в IPython из-за автоматических отступов самого IPython начинает сдвигаться код:

```
In [1]: a = 10

In [2]: b = 5

In [3]: if a > b:
   ...:     print("A is bigger")
   ...: else:
   ...:     print("A is less or equal")
   ...:
A is bigger

In [4]: %hist
a = 10
b = 5
if a > b:
    print("A is bigger")
else:
    print("A is less or equal")
%hist

In [5]: if a > b:
   ...:         print("A is bigger")
   ...:     else:
   ...:             print("A is less or equal")
   ...:
  File "<ipython-input-8-4d18ff094f5c>", line 3
    else:
         ^
IndentationError: unindent does not match any outer indentation level
If you want to paste code into IPython, try the %paste and %cpaste magic functions.
```

Обратите внимание на последнюю строку. IPython подсказывает, какой командой воспользоваться, чтобы корректно вставить такой код.

Команды %paste и %cpaste работают немного по-разному.

%cpaste (после того, как все строки скопированы, надо завершить работу команды, набрав '--'):

```
In [9]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:if a > b:
:    print("A is bigger")
:else:
:    print("A is less or equal")
:--
A is bigger
```

%paste (требует установленного Tkinter):

```
In [10]: %paste
if a > b:
    print("A is bigger")
else:
    print("A is less or equal")

## -- End pasted text --
A is bigger
```

Подробнее об IPython можно почитать в документация IPython.

Коротко информацию можно посмотреть в самом IPython командой %quickref:

```
IPython -- An enhanced Interactive Python - Quick Reference Card
================================================================

obj?, obj??      : Get help, or more help for object (also works as
                   ?obj, ??obj).
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
%magic           : Information about IPython's 'magic' % functions.

Magic functions are prefixed by % or %%, and typically take their arguments
without parentheses, quotes or even commas for convenience.  Line magics take a
single % and cell magics are prefixed with two %%.

Example magic function calls:

%alias d ls -F   : 'd' is now an alias for 'ls -F'
alias d ls -F    : Works if 'alias' not a python name
alist = %alias   : Get list of aliases to 'alist'
cd /usr/share    : Obvious. cd -<tab> to choose from visited dirs.
%cd??            : See help AND source for magic %cd
%timeit x=10     : time the 'x=10' statement with high precision.
%%timeit x=2**100
x**100           : time 'x**100' with a setup of 'x=2**100'; setup code is not
                   counted.  This is an example of a cell magic.

System commands:

!cp a.txt b/     : System command escape, calls os.system()
cp a.txt b/      : after %rehashx, most system commands work without !
cp ${f}.txt $bar : Variable expansion in magics and system commands
files = !ls /usr : Capture sytem command output
files.s, files.l, files.n: "a b c", ['a','b','c'], 'a\nb\nc'

History:

_i, _ii, _iii    : Previous, next previous, next next previous input
_i4, _ih[2:5]    : Input history line 4, lines 2-4
exec _i81        : Execute input history line #81 again
%rep 81          : Edit input history line #81
_, __, ___       : previous, next previous, next next previous output
_dh              : Directory history
_oh              : Output history
%hist            : Command history of current session.
%hist -g foo     : Search command history of (almost) all sessions for 'foo'.
%hist -g         : Command history of (almost) all sessions.
%hist 1/2-8      : Command history containing lines 2-8 of session 1.
%hist 1/ ~2/     : Command history of session 1 and 2 sessions before current.
```

Переменные

Переменные в Python:

- не требуют объявления типа переменной (Python - язык с динамической типизацией)
- являются ссылками на область памяти

Правила именования переменных:

- имя переменной может состоять только из букв, цифр и знака подчеркивания
- имя не может начинаться с цифры
- имя не может содержать специальных символов @, $, %

Создавать переменные в Python очень просто:

```
In [1]: a = 3

In [2]: b = 'Hello'

In [3]: c, d = 9, 'Test'

In [4]: print(a,b,c,d)
3 Hello 9 Test
```

Обратите внимание, что в Python не нужно указывать, что a это число, a b это строка.

Переменные являются ссылками на область памяти. Это легко продемонстрировать с помощью функции `id()`, которая показывает идентификатор объекта:

```
In [5]: a = b = c = 33

In [6]: id(a)
Out[6]: 31671480

In [7]: id(b)
Out[7]: 31671480

In [8]: id(c)
Out[8]: 31671480
```

В этом примере видно, что все три имени ссылаются на один и тот же идентификатор. То есть, это один и тот же объект, на который указывают три ссылки a, b и c.

С числами у Python есть ещё одна особенность, которая может немного сбить. Числа от -5 до 256 заранее созданы и хранятся в массиве (списке). Поэтому при создании числа из этого диапазона фактически создается ссылка на число в созданном массиве.

> Эта особенность характерна именно для реализации CPython, которая рассматривается в курсе.

Это можно проверить таким образом:

```
In [9]: a = 3

In [10]: b = 3

In [11]: id(a)
Out[11]: 4400936168

In [12]: id(b)
Out[12]: 4400936168

In [13]: id(3)
Out[13]: 4400936168
```

Обратите внимание, что у `a`, `b` и числа `3` одинаковые идентификаторы. Все они просто являются ссылками на существующее число в списке.

Но если сделать то же самое с числом больше 256:

```
In [14]: a = 500

In [15]: b = 500

In [16]: id(a)
Out[16]: 140239990503056

In [17]: id(b)
Out[17]: 140239990503032

In [18]: id(500)
Out[18]: 140239990502960
```

Идентификаторы у всех разные.

При этом, если сделать присваивание такого вида:

```
In [19]: a = b = c = 500
```

Идентификаторы будут у всех одинаковые:

```
In [20]: id(a)
Out[20]: 140239990503080

In [21]: id(b)
Out[21]: 140239990503080

In [22]: id(c)
Out[22]: 140239990503080
```

Так как в таком варианте a, b и c просто ссылаются на один и тот же объект.

Имена переменных

Имена переменных не должны пересекаться с операторами и названиями модулей или других зарезервированных значений.

В Python есть рекомендации по именованию функций, классов и переменных:

- имена переменных обычно пишутся полностью большими или маленькими буквами
  - DB_NAME
  - db_name
- имена функций задаются маленькими буквами, с подчеркиваниями между словами
  - get_names
- имена классов задаются словами с заглавными буквами, без пробелов
  - CiscoSwitch

Типы данных в Python

В Python есть несколько стандартных типов данных:

- Numbers (числа)
- Strings (строки)
- Lists (списки)
- Dictionaries (словари)
- Tuples (кортежи)
- Sets (множества)
- Boolean

Эти типы данных можно, в свою очередь, классифицировать по нескольким признакам:

- Изменяемые:
  - Списки
  - Словари
  - Множества
- Неизменяемые
  - Числа
  - Строки
  - Кортежи
- Упорядоченные:
  - Списки
  - Кортежи
  - Строки
- Неупорядоченные:
  - Словари
  - Множества

Числа

С числами можно выполнять различные математические операции.

```
In [1]: 1 + 2
Out[1]: 3

In [2]: 1.0 + 2
Out[2]: 3.0

In [3]: 10 - 4
Out[3]: 6

In [4]: 2**3
Out[4]: 8
```

Деление int и float:

```
In [5]: 10/3
Out[5]: 3.3333333333333335

In [6]: 10/3.0
Out[6]: 3.3333333333333335
```

С помощью функции round можно округлять числа до нужного количества знаков:

```
In [9]: round(10/3.0, 2)
Out[9]: 3.33

In [10]: round(10/3.0, 4)
Out[10]: 3.3333
```

Остаток от деления:

```
In [11]: 10 % 3
Out[11]: 1
```

Операторы сравнения

```
In [12]: 10 > 3.0
Out[12]: True

In [13]: 10 < 3
Out[13]: False

In [14]: 10 == 3
Out[14]: False

In [15]: 10 == 10
Out[15]: True

In [16]: 10 <= 10
Out[16]: True

In [17]: 10.0 == 10
Out[17]: True
```

Функция int() позволяет выполнять конвертацию в тип int. Во втором аргументе можно указывать систему счисления:

```
In [18]: a = '11'

In [19]: int(a)
Out[19]: 11
```

Если указать, что строку a надо воспринимать как двоичное число, то результат будет таким:

```
In [20]: int(a, 2)
Out[20]: 3
```

Конвертация в int типа float:

```
In [21]: int(3.333)
Out[21]: 3

In [22]: int(3.9)
Out[22]: 3
```

Функция bin позволяет получить двоичное представление числа (обратите внимание, что результат - строка):

```
In [23]: bin(8)
Out[23]: '0b1000'

In [24]: bin(255)
Out[24]: '0b11111111'
```

Аналогично, функция hex() позволяет получить шестнадцатеричное значение:

```
In [25]: hex(10)
Out[25]: '0xa'
```

И, конечно же, можно делать несколько преобразований одновременно:

```
In [26]: int('ff', 16)
Out[26]: 255

In [27]: bin(int('ff', 16))
Out[27]: '0b11111111'
```

Для более сложных математических функций в Python есть модуль **math**:

```
In [28]: import  math

In [29]: math.sqrt(9)
Out[29]: 3.0

In [30]: math.sqrt(10)
Out[30]: 3.1622776601683795

In [31]: math.factorial(3)
Out[31]: 6

In [32]: math.pi
Out[32]: 3.141592653589793
```

Строки (Strings)

Строка в Python - это последовательность символов, заключенная в кавычки.

> Строки - это неизменяемый упорядоченный тип данных.

Примеры строк:

```
In [9]: 'Hello'
Out[9]: 'Hello'
In [10]: "Hello"
Out[10]: 'Hello'

In [11]: tunnel = """
   ....: interface Tunnel0
   ....:  ip address 10.10.10.1 255.255.255.0
   ....:  ip mtu 1416
   ....:  ip ospf hello-interval 5
   ....:  tunnel source FastEthernet1/0
   ....:  tunnel protection ipsec profile DMVPN
   ....: """

In [12]: tunnel
Out[12]: '\ninterface Tunnel0\n ip address 10.10.10.1 255.255.255.0\n ip mtu 1416\n ip
 ospf hello-interval 5\n tunnel source FastEthernet1/0\n tunnel protection ipsec profi
le DMVPN\n'

In [13]: print(tunnel)

interface Tunnel0
 ip address 10.10.10.1 255.255.255.0
 ip mtu 1416
 ip ospf hello-interval 5
 tunnel source FastEthernet1/0
 tunnel protection ipsec profile DMVPN
```

Строки можно суммировать. Тогда они объединяются в одну строку:

```
In [14]: intf = 'interface'

In [15]: tun = 'Tunnel0'

In [16]: intf + tun
Out[16]: 'interfaceTunnel0'

In [17]: intf + ' ' + tun
Out[17]: 'interface Tunnel0'
```

Строку можно умножать на число. В этом случае, строка повторяется указанное количество раз:

```
In [18]: intf * 5
Out[18]: 'interfaceinterfaceinterfaceinterfaceinterface'

In [19]: '#' * 40
Out[19]: '########################################'
```

То, что строки являются упорядоченным типом данных, позволяет обращаться к символам в строке по номеру, начиная с нуля:

```
In [20]: string1 = 'interface FastEthernet1/0'

In [21]: string1[0]
Out[21]: 'i'
```

Нумерация всех символов в строке идет с нуля. Но, если нужно обратиться к какому-то по счету символу, начиная с конца, то можно указывать отрицательные значения (на этот раз с единицы).

```
In [22]: string1[1]
Out[22]: 'n'

In [23]: string1[-1]
Out[23]: '0'
```

Кроме обращения к конкретному символу, можно делать срезы строки, указав диапазон номеров (срез выполняется по второе число, не включая его):

```
In [24]: string1[0:9]
Out[24]: 'interface'

In [25]: string1[10:22]
Out[25]: 'FastEthernet'
```

Если не указывается второе число, то срез будет до конца строки:

```
In [26]:  string1[10:]
Out[26]: 'FastEthernet1/0'
```

Срезать три последних символа строки:

```
In [27]: string1[-3:]
Out[27]: '1/0'
```

Строка в обратном порядке:

```
In [28]: a = '0123456789'

In [29]: a[::]
Out[29]: '0123456789'

In [30]: a[::-1]
Out[30]: '9876543210'
```

Записи `a[::]` и `a[:]` дают одинаковый результат, но двойное двоеточие позволяет указывать, что надо брать не каждый элемент, а, например, каждый второй.

Например, таким образом можно получить все четные числа строки a:

```
In [31]: a[::2]
Out[31]: '02468'
```

Так можно получить нечетные:

```
In [32]: a[1::2]
Out[32]: '13579'
```

Полезные методы для работы со строками

Так как конфигурационный файл - это просто текстовый файл, при автоматизации очень часто надо будет работать со строками.

Знание различных методов (то есть, действий), которые можно применять к строкам, помогает более эффективно работать с ними.

> Чаще всего эти методы будут применяться в циклах, при обработке файла.

`upper(), lower(), swapcase(), capitalize()`

Методы `upper()`, `lower()`, `swapcase()`, `capitalize()` выполняют преобразование регистра строки:

```
In [25]: string1 = 'FastEthernet'

In [26]: string1.upper()
Out[26]: 'FASTETHERNET'

In [27]: string1.lower()
Out[27]: 'fastethernet'

In [28]: string1.swapcase()
Out[28]: 'fASTeTHERNET'

In [29]: string2 = 'tunnel 0'

In [30]: string2.capitalize()
Out[30]: 'Tunnel 0'
```

Очень важно обращать внимание на то, что часто методы возвращают преобразованную строку. И, значит, надо не забыть присвоить ее какой-то переменной (можно той же).

```
In [31]: string1 = string1.upper()

In [32]: print(string1)
FASTETHERNET
```

Так происходит из-за того, что строка - это неизменяемый тип данных.

`count()`

Метод `count()` используется для подсчета того, сколько раз символ или подстрока встречаются в строке:

```
In [33]: string1 = 'Hello, hello, hello, hello'

In [34]: string1.count('hello')
Out[34]: 3

In [35]: string1.count('ello')
Out[35]: 4

In [36]: string1.count('l')
Out[36]: 8
```

find()

Методу `find()` можно передать подстроку или символ, и он покажет, на какой позиции находится первый символ подстроки (для первого совпадения):

```
In [37]: string1 = 'interface FastEthernet0/1'

In [38]: string1.find('Fast')
Out[38]: 10

In [39]: string1[string1.find('Fast')::]
Out[39]: 'FastEthernet0/1'
```

startswith(), endswith()

Проверка на то, начинается (или заканчивается) ли строка на определенные символы (методы `startswith()`, `endswith()`):

```
In [40]: string1 = 'FastEthernet0/1'

In [41]: string1.startswith('Fast')
Out[41]: True

In [42]: string1.startswith('fast')
Out[42]: False

In [43]: string1.endswith('0/1')
Out[43]: True

In [44]: string1.endswith('0/2')
Out[44]: False
```

`replace()`

Замена последовательности символов в строке на другую последовательность (метод `replace()`):

```
In [45]: string1 = 'FastEthernet0/1'

In [46]: string1.replace('Fast', 'Gigabit')
Out[46]: 'GigabitEthernet0/1'
```

`strip()`

Часто при обработке файла файл открывается построчно. Но в конце каждой строки, как правило, есть какие-то спецсимволы (а могут быть и в начале). Например, перевод строки.

Для того, чтобы избавиться от них, очень удобно использовать метод `strip()`:

```
In [47]: string1 = '\n\tinterface FastEthernet0/1\n'

In [48]: print(string1)

    interface FastEthernet0/1


In [49]: string1
Out[49]: '\n\tinterface FastEthernet0/1\n'

In [50]: string1.strip()
Out[50]: 'interface FastEthernet0/1'
```

По умолчанию, метод strip() убирает whitespace символы. Однако, ему можно передать как аргумент любые символы. Тогда в начале и в конце строки будут удалены все символы, которые были указаны в строке:

```
In [51]: ad_metric = '[110/1045]'

In [52]: ad_metric.strip('[]')
Out[52]: '110/1045'
```

Метод strip() убирает спецсимволы и в начале, и в конце строки. Если необходимо убрать символы только слева или только справа, можно использовать, соответственно, методы `lstrip()` и `rstrip()`.

`split()`

Метод **`split()`** разбивает строку на части, используя как разделитель какой-то символ (или символы). По умолчанию, в качестве разделителя используются пробелы. Но в скобках можно указать любой разделитель.

В результате, строка будет разбита на части по указанному разделителю и представлена в виде частей, которые содержатся в списке:

```
In [53]: string1 = ' switchport trunk allowed vlan 10,20,30,100-200\n'

In [54]: commands = string1.strip().split()

In [55]: print(commands)
['switchport', 'trunk', 'allowed', 'vlan', '10,20,30,100-200']

In [56]: vlans = commands[-1].split(',')

In [57]: print(vlans)
['10', '20', '30', '100-200']
```

В строке string1 был символ пробела в начале и символ перевода строки в конце. В строке номер 54 с помощью метода `strip()` эти символы удаляются.

Метод `strip()` возвращает строку, которая обрабатывается методом `split()` и разделяет строку на части, используя пробел как разделитель. Итоговая строка присваивается переменной commands.

Используя тот же способ, что и со строками, к последнему объекту в списке vlans применяется метод `split()`. Но на этот раз внутри скобок указывается другой разделитель - запятая. В итоге, в списке vlans находятся номера VLAN.

У метода `split()` есть ещё одна хорошая особенность: по умолчанию метод разбивает строку не по одному пробелу, а по любому количеству пробелов. Это будет очень полезным при обработке команд show. Например:

```
In [58]: sh_ip_int_br = "FastEthernet0/0            15.0.15.1       YES manual up
                up"

In [59]: sh_ip_int_br.split()
Out[59]: ['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up']
```

А вот так выглядит разделение той же строки, когда один пробел используется как разделитель:

```
In [60]: sh_ip_int_br.split(' ')
Out[60]:
['FastEthernet0/0', '', '', '', '', '', '', '', '', '', '', '', '15.0.15.1', '', '', '
', '', '', '', 'YES', 'manual', 'up', '', '', '', '', '', '', '', '', '', '', '', '',
'', '', '', '', '', '', '', 'up']
```

Форматирование строк

При работе со строками часто возникают ситуации, когда в шаблон строки надо подставить разные данные.

Это можно делать объединяя, части строки и данные, но в Python есть более удобный способ: форматирование строк.

Форматирование строк может помочь, например, в таких ситуациях:

- необходимо подставить значения в строку по определенному шаблону
- необходимо отформатировать вывод столбцами
- надо конвертировать числа в двоичный формат

Существует два варианта форматирования строк:

- с оператором `%` (более старый вариант)
- методом `format()` (новый вариант)

Несмотря на то, что рекомендуется использовать метод `format`, часто можно встретить форматирование строк и через оператор `%`.

Пример использования метода format:

```
In [1]: "interface FastEthernet0/{}".format('1')
Out[1]: 'interface FastEthernet0/1'
```

Специальный символ `{}` указывает, что сюда подставится значение, которое передается методу format. При этом, каждая пара фигурных скобок обозначает одно место для подстановки.

Кроме того, в форматировании строк специальные символы могут указывать, какой тип данных будет подставляться.

Аналогичный пример с оператором %:

```
In [2]: "interface FastEthernet0/%s" % '1'
Out[2]: 'interface FastEthernet0/1'
```

В старом синтаксисе форматирования строк используются такие обозначения:

- `%s` - строка или любой другой объект в котором есть строковое представление
- `%d` - integer
- `%f` - float

С помощью форматирования строк можно выводить результат столбцами. В форматировании строк можно указывать, какое количество символов выделено на данные. Если количество символов в данных меньше, чем выделенное количество символов, недостающие символы заполняются пробелами.

Например, таким образом можно вывести данные столбцами одинаковой ширины по 15 символов с выравниванием по правой стороне:

```
In [3]: vlan, mac, intf = ['100', 'aabb.cc80.7000', 'Gi0/1']

In [4]: print("%15s %15s %15s" % (vlan, mac, intf))
            100  aabb.cc80.7000           Gi0/1

In [5]: print("{:>15} {:>15} {:>15}".format(vlan, mac, intf))
            100  aabb.cc80.7000           Gi0/1
```

Выравнивание по левой стороне:

```
In [6]: print("%-15s %-15s %-15s" % (vlan, mac, intf))
100             aabb.cc80.7000  Gi0/1

In [7]: print("{:15} {:15} {:15}".format(vlan, mac, intf))
100             aabb.cc80.7000  Gi0/1
```

С помощью форматирования строк можно также влиять на отображение чисел.

Например, можно указать, сколько цифр после запятой выводить:

```
In [8]: print("%.3f" % (10.0/3))
3.333

In [9]: print("{:.3f}".format(10.0/3))
3.333
```

Конвертировать в двоичный формат, указать, сколько цифр должно быть в отображении числа, и дополнить недостающее нулями:

```
In [10]: '{:08b}'.format(10)
Out[10]: '00001010'
```

Аналогичный результат можно получить с помощью метода zfill:

```
In [11]: bin(10)[2:].zfill(8)
Out[11]: '00001010'
```

> У форматирования строк есть ещё много возможностей. Хорошие примеры и объяснения двух вариантов форматирования строк можно найти тут.

Объединение литералов строк

В Python есть очень удобная функциональность - объединение литералов строк.

```
In [1]: s = ('Test' 'String')

In [2]: s
Out[2]: 'TestString'

In [3]: s = 'Test' 'String'

In [4]: s
Out[4]: 'TestString'
```

Можно даже переносить составляющие строки на разные строки, но только если они в скобках

```
In [5]: s = ('Test'
   ...: 'String')

In [6]: s
Out[6]: 'TestString'
```

Этим очень удобно пользоваться в регулярных выражениях:

```
regex = ('(\S+) +(\S+) +'
         '\w+ +\w+ +'
         '(up|down|administratively down) +'
         '(\w+)')
```

Так регулярное выражение можно разбивать на части и его будет проще понять. Плюс можно добавлять поясняющие комментарии в строках.

```
regex = ('(\S+) +(\S+) +' #interface and IP
         '\w+ +\w+ +'
         '(up|down|administratively down) +' #Status
         '(\w+)') #Protocol
```

Также этим приемом удобно пользоваться, когда надо написать длинное сообщение:

```
In [7]: message = ('При выполнении команды "{}" '
   ...: 'возникла такая ошибка "{}".\n'
   ...: 'Исключить эту команду из списка? [y/n]')

In [8]: message
Out[8]: 'При выполнении команды "{}" возникла такая ошибка "{}".\nИсключить эту команд
y из списка? [y/n]'
```

Список (List)

Список - это изменяемый упорядоченный тип данных.

Список в Python - это последовательность элементов, разделенных между собой запятой и заключенных в квадратные скобки.

Примеры списков:

```
In [1]: list1 = [10,20,30,77]
In [2]: list2 = ['one', 'dog', 'seven']
In [3]: list3 = [1, 20, 4.0, 'word']
```

Так как список - это упорядоченный тип данных, то, как и в строках, в списках можно обращаться к элементу по номеру, делать срезы:

```
In [4]: list3 = [1, 20, 4.0, 'word']

In [5]: list3[1]
Out[5]: 20

In [6]: list3[1::]
Out[6]: [20, 4.0, 'word']

In [7]: list3[-1]
Out[7]: 'word'

In [8]: list3[::-1]
Out[8]: ['word', 4.0, 20, 1]
```

Перевернуть список наоборот можно и с помощью метода reverse():

```
In [10]: vlans = ['10', '15', '20', '30', '100-200']

In [11]: vlans.reverse()

In [12]: vlans
Out[12]: ['100-200', '30', '20', '15', '10']
```

Так как списки изменяемые, элементы списка можно менять:

```
In [13]: list3
Out[13]: [1, 20, 4.0, 'word']

In [14]: list3[0] = 'test'

In [15]: list3
Out[15]: ['test', 20, 4.0, 'word']
```

Можно создавать и список списков. И, как и в обычном списке, можно обращаться к элементам во вложенных списках:

```
In [16]: interfaces = [['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up'],
   ....: ['FastEthernet0/1', '10.0.1.1', 'YES', 'manual', 'up', 'up'],
   ....: ['FastEthernet0/2', '10.0.2.1', 'YES', 'manual', 'up', 'down']]

In [17]: interfaces[0][0]
Out[17]: 'FastEthernet0/0'

In [18]: interfaces[2][0]
Out[18]: 'FastEthernet0/2'

In [19]: interfaces[2][1]
Out[19]: '10.0.2.1'
```

Полезные методы для работы со списками

`join()`

Метод **join()** собирает список строк в одну строку с разделителем, который указан в join():

```
In [16]: vlans = ['10', '20', '30', '100-200']

In [17]: ','.join(vlans[:-1])
Out[17]: '10,20,30'
```

`append()`

Метод **append()** добавляет в конец списка указанный элемент:

```
In [18]: vlans = ['10', '20', '30', '100-200']

In [19]: vlans.append('300')

In [20]: vlans
Out[20]: ['10', '20', '30', '100-200', '300']
```

`extend()`

Если нужно объединить два списка, то можно использовать два способа: метод **extend()** и операцию сложения:

```
In [21]: vlans = ['10', '20', '30', '100-200']

In [22]: vlans2 = ['300', '400', '500']

In [23]: vlans.extend(vlans2)

In [24]: vlans
Out[24]: ['10', '20', '30', '100-200', '300', '400', '500']

In [25]: vlans + vlans2
Out[25]: ['10', '20', '30', '100-200', '300', '400', '500', '300', '400', '500']

In [26]: vlans
Out[26]: ['10', '20', '30', '100-200', '300', '400', '500']
```

Но при этом метод extend() расширяет список "на месте", а при операции сложения выводится итоговый суммарный список, но исходные списки не меняются.

pop()

Метод **pop()** удаляет элемент, который соответствует указанному номеру. Но, что важно, при этом метод возвращает этот элемент:

```
In [28]: vlans = ['10', '20', '30', '100-200']

In [29]: vlans.pop(-1)
Out[29]: '100-200'

In [30]: vlans
Out[30]: ['10', '20', '30']
```

Без указания номера удаляется последний элемент списка.

remove()

Метод **remove()** удаляет указанный элемент.

remove() не возвращает удаленный элемент:

```
In [31]: vlans = ['10', '20', '30', '100-200']

In [32]: vlans.remove('20')

In [33]: vlans
Out[33]: ['10', '30', '100-200']
```

В методе remove надо указывать сам элемент, который надо удалить, а не его номер в списке. Если указать номер элемента, возникнет ошибка:

```
In [34]: vlans.remove(-1)
-------------------------------------------------
ValueError      Traceback (most recent call last)
<ipython-input-32-f4ee38810cb7> in <module>()
----> 1 vlans.remove(-1)

ValueError: list.remove(x): x not in list
```

index()

Метод **index()** используется для того, чтобы проверить, под каким номером в списке хранится элемент:

```
In [35]: vlans = ['10', '20', '30', '100-200']

In [36]: vlans.index('30')
Out[36]: 2
```

`insert()`

Метод **insert()** позволяет вставить элемент на определенное место в списке:

```
In [37]: vlans = ['10', '20', '30', '100-200']

In [38]: vlans.insert(1,'15')

In [39]: vlans
Out[39]: ['10', '15', '20', '30', '100-200']
```

Варианты создания списка

Создание списка с помощью литерала:

```
In [1]: vlans = [10, 20, 30, 50]
```

> Литерал - это выражение, которое создает объект.

Создание списка с помощью функции **list()**:

```
In [2]: list1 = list('router')

In [3]: print(list1)
['r', 'o', 'u', 't', 'e', 'r']
```

Генераторы списков:

```
In [4]: list2 = ['FastEthernet0/'+ str(i) for i in range(10)]

In [5]: list2
Out[6]:
['FastEthernet0/0',
 'FastEthernet0/1',
 'FastEthernet0/2',
 'FastEthernet0/3',
 'FastEthernet0/4',
 'FastEthernet0/5',
 'FastEthernet0/6',
 'FastEthernet0/7',
 'FastEthernet0/8',
 'FastEthernet0/9']
```

Словарь (Dictionary)

Словари - это изменяемый неупорядоченный тип данных

> В модуле collections доступны упорядоченные объекты, внешне идентичные словарям OrderedDict.

Словарь (ассоциативный массив, хеш-таблица):

- данные в словаре - это пары `ключ: значение`
- доступ к значениям осуществляется по ключу, а не по номеру, как в списках
- словари не упорядочены, поэтому не стоит полагаться на порядок элементов словаря
- так как словари изменяемы, то элементы словаря можно менять, добавлять, удалять
- ключ должен быть объектом неизменяемого типа:
  - число
  - строка
  - кортеж
- значение может быть данными любого типа

Пример словаря:

```
london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco', 'model': '44
51', 'ios': '15.4'}
```

Можно записывать и так:

```
london = {
        'id': 1,
        'name':'London',
        'it_vlan':320,
        'user_vlan':1010,
        'mngmt_vlan':99,
        'to_name': None,
        'to_id': None,
        'port':'G1/0/11'
}
```

Для того, чтобы получить значение из словаря, надо обратиться по ключу, таким же образом, как это было в списках, только вместо номера будет использоваться ключ:

```
In [1]: london = {'name': 'London1', 'location': 'London Str'}

In [2]: london['name']
Out[2]: 'London1'

In [3]: london['location']
Out[3]: 'London Str'
```

Аналогичным образом можно добавить новую пару ключ:значение:

```
In [4]: london['vendor'] = 'Cisco'

In [5]: print(london)
{'vendor': 'Cisco', 'name': 'London1', 'location': 'London Str'}
```

В словаре в качестве значения можно использовать словарь:

```
london_co = {
    'r1' : {
    'hostname': 'london_r1',
    'location': '21 New Globe Walk',
    'vendor': 'Cisco',
    'model': '4451',
    'ios': '15.4',
    'ip': '10.255.0.1'
    },
    'r2' : {
    'hostname': 'london_r2',
    'location': '21 New Globe Walk',
    'vendor': 'Cisco',
    'model': '4451',
    'ios': '15.4',
    'ip': '10.255.0.2'
    },
    'sw1' : {
    'hostname': 'london_sw1',
    'location': '21 New Globe Walk',
    'vendor': 'Cisco',
    'model': '3850',
    'ios': '3.6.XE',
    'ip': '10.255.0.101'
    }
}
```

Получить значения из вложенного словаря можно так:

```
In [7]: london_co['r1']['ios']
Out[7]: '15.4'

In [8]: london_co['r1']['model']
Out[8]: '4451'

In [9]: london_co['sw1']['ip']
Out[9]: '10.255.0.101'
```

Полезные методы для работы со словарями

`clear()`

Метод **clear()** позволяет очистить словарь:

```
In [1]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco', 'mod
el': '4451', 'ios': '15.4'}

In [2]: london.clear()

In [3]: london
Out[3]: {}
```

`copy()`

Метод **copy()** позволяет создать полную копию словаря.

Если указать, что один словарь равен другому:

```
In [4]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [5]: london2 = london

In [6]: id(london)
Out[6]: 25489072

In [7]: id(london2)
Out[7]: 25489072

In [8]: london['vendor'] = 'Juniper'

In [9]: london2['vendor']
Out[9]: 'Juniper'
```

В этом случае london2 это еще одно имя, которое ссылается на словарь. И при изменениях словаря london меняется и словарь london2, так как это ссылки на один и тот же объект.

Поэтому, если нужно сделать копию словаря, надо использовать метод copy():

```
In [10]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [11]: london2 = london.copy()

In [12]: id(london)
Out[12]: 25524512

In [13]: id(london2)
Out[13]: 25563296

In [14]: london['vendor'] = 'Juniper'

In [15]: london2['vendor']
Out[15]: 'Cisco'
```

get()

Если при обращении к словарю указывается ключ, которого нет в словаре, возникает ошибка:

```
In [16]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [17]: london['ios']
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-17-b4fae8480b21> in <module>()
----> 1 london['ios']

KeyError: 'ios'
```

Метод **get()** запрашивает ключ и, если его нет, вместо ошибки возвращает `None`.

```
In [18]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [19]: print(london.get('ios'))
None
```

Метод get() позволяет также указывать другое значение вместо `None`:

```
In [20]: print(london.get('ios', 'Ooops'))
Ooops
```

setdefault()

Метод **setdefault()** ищет ключ и, если его нет, вместо ошибки создает ключ со значением `None`.

```
In [21]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [22]: ios = london.setdefault('ios')

In [23]: print(ios)
None

In [24]: london
Out[24]: {'ios': None, 'location': 'London Str', 'name': 'London1', 'vendor': 'Cisco'}
```

Второй аргумент позволяет указать, какое значение должно соответствовать ключу:

```
In [25]: model = london.setdefault('model', 'Cisco3580')

In [26]: print(model)
Cisco3580

In [27]: london
Out[27]:
{'ios': None,
 'model': 'Cisco3580',
 'location': 'London Str',
 'name': 'London1',
 'vendor': 'Cisco'}
```

`keys(), values(), items()`

Методы **keys()**, **values()**, **items()**:

```
In [24]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [25]: london.keys()
Out[25]: dict_keys(['name', 'location', 'vendor'])

In [26]: london.values()
Out[26]: dict_values(['London1', 'London Str', 'Cisco'])

In [27]: london.items()
Out[27]: dict_items([('name', 'London1'), ('location', 'London Str'), ('vendor', 'Cisc
o')])
```

Все три метода возвращают специальные объекты view, которые отображают ключи, значения и пары ключ-значение словаря соответственно.

Очень важная особенность view заключается в том, что они меняются вместе с изменением словаря. И фактически они лишь дают способ посмотреть на соответствующие объекты, но не создают их копию.

На примере метода keys():

```
In [28]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [29]: keys = london.keys()

In [30]: print(keys)
dict_keys(['name', 'location', 'vendor'])
```

Сейчас переменной keys соответствует view dict_keys, в котором три ключа: name, locationa и vendor.

Но, если мы добавим в словарь еще одну пару ключ-значение, объект keys тоже поменяется:

```
In [31]: london['ip'] = '10.1.1.1'

In [32]: keys
Out[32]: dict_keys(['name', 'location', 'vendor', 'ip'])
```

Если нужно получить обычный список ключей, который не будет меняться с изменениями словаря, достаточно конвертировать view в список:

```
In [33]: list_keys = list(london.keys())

In [34]: list_keys
Out[34]: ['name', 'location', 'vendor', 'ip']
```

`del`

Удалить ключ и значение:

```
In [35]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [36]: del(london['name'])

In [37]: london
Out[37]: {'location': 'London Str', 'vendor': 'Cisco'}
```

`update`

Метод update позволяет добавлять в словарь содержимое другого словаря:

```
In [38]: r1 = {'name': 'London1', 'location': 'London Str'}

In [39]: r1.update({'vendor': 'Cisco', 'ios':'15.2'})

In [40]: r1
Out[40]: {'ios': '15.2', 'location': 'London Str', 'name': 'London1', 'vendor': 'Cisco'
}
```

Аналогичным образом можно обновить значения:

```
In [41]: r1.update({'name': 'london-r1', 'ios':'15.4'})

In [42]: r1
Out[42]:
{'ios': '15.4',
 'location': 'London Str',
 'name': 'london-r1',
 'vendor': 'Cisco'}
```

Варианты создания словаря

Литерал

Словарь можно создать с помощью литерала:

```
In [1]: r1 = {'model': '4451', 'ios': '15.4'}
```

dict

Конструктор **dict** позволяет создавать словарь несколькими способами.

Если в роли ключей используются строки, можно использовать такой вариант создания словаря:

```
In [2]: r1 = dict(model='4451', ios='15.4')

In [3]: r1
Out[3]: {'ios': '15.4', 'model': '4451'}
```

Второй вариант создания словаря с помощью dict:

```
In [4]: r1 = dict([('model','4451'), ('ios','15.4')])

In [5]: r1
Out[5]: {'ios': '15.4', 'model': '4451'}
```

dict.fromkeys

В ситуации, когда надо создать словарь с известными ключами, но, пока что, пустыми значениями (или одинаковыми значениями), очень удобен метод **fromkeys()**:

```
In [5]: d_keys = ['hostname', 'location', 'vendor', 'model', 'ios', 'ip']

In [6]: r1 = dict.fromkeys(d_keys)

In [7]: r1
Out[7]:
{'ios': None,
 'ip': None,
 'hostname': None,
 'location': None,
 'model': None,
 'vendor': None}
```

По умолчанию, метод fromkeys подставляет значение None. Но можно указывать и свой вариант значения:

```
In [8]: router_models = ['ISR2811', 'ISR2911', 'ISR2921', 'ASR9002']

In [9]: models_count = dict.fromkeys(router_models, 0)

In [10]: models_count
Out[10]: {'ASR9002': 0, 'ISR2811': 0, 'ISR2911': 0, 'ISR2921': 0}
```

Этот вариант создания словаря подходит не для всех случаев. Например, при использовании изменяемого типа данных в значении, будет создана ссылка на один и тот же объект:

```
In [11]: router_models = ['ISR2811', 'ISR2911', 'ISR2921', 'ASR9002']

In [12]: routers = dict.fromkeys(router_models, [])

In [13]: routers
Out[13]: {'ASR9002': [], 'ISR2811': [], 'ISR2911': [], 'ISR2921': []}

In [14]: routers['ASR9002'].append('london_r1')

In [15]: routers
Out[15]:
{'ASR9002': ['london_r1'],
 'ISR2811': ['london_r1'],
 'ISR2911': ['london_r1'],
 'ISR2921': ['london_r1']}
```

В данном случае каждый ключ ссылается на один и тот же список. Поэтому, при добавлении значения в один из списков обновляются и остальные.

Генератор словаря (dict comprehensions)

И последний метод создания словаря - **генераторы словарей**.

Сгенерируем словарь со списками в значении, как в предыдущем примере:

```
In [16]: router_models = ['ISR2811', 'ISR2911', 'ISR2921', 'ASR9002']

In [17]: routers = {key: [] for key in router_models}

In [18]: routers
Out[18]: {'ASR9002': [], 'ISR2811': [], 'ISR2911': [], 'ISR2921': []}

In [19]: routers['ASR9002'].append('london_r1')

In [20]: routers
Out[20]: {'ASR9002': ['london_r1'], 'ISR2811': [], 'ISR2911': [], 'ISR2921': []}
```

Кортеж (Tuple)

Кортеж - это неизменяемый упорядоченный тип данных.

Кортеж в Python - это последовательность элементов, которые разделены между собой запятой и заключены в скобки.

Грубо говоря, кортеж - это список, который нельзя изменить. То есть, в кортеже есть только права на чтение. Это может быть защитой от случайных изменений.

Создать пустой кортеж:

```
In [1]: tuple1 = tuple()

In [2]: print(tuple1)
()
```

Кортеж из одного элемента (обратите внимание на запятую):

```
In [3]: tuple2 = ('password',)
```

Кортеж из списка:

```
In [4]: list_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']

In [5]: tuple_keys = tuple(list_keys)

In [6]: tuple_keys
Out[6]: ('hostname', 'location', 'vendor', 'model', 'IOS', 'IP')
```

К объектам в кортеже можно обращаться, как и к объектам списка, по порядковому номеру:

```
In [7]: tuple_keys[0]
Out[7]: 'hostname'
```

Но так как кортеж неизменяем, присвоить новое значение нельзя:

```
In [8]: tuple_keys[1] = 'test'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-1c7162cdefa3> in <module>()
----> 1 tuple_keys[1] = 'test'

TypeError: 'tuple' object does not support item assignment
```

Множество (Set)

Множество - это изменяемый неупорядоченный тип данных. В множестве всегда содержатся только уникальные элементы.

Множество в Python - это последовательность элементов, которые разделены между собой запятой и заключены в фигурные скобки.

С помощью множества можно легко убрать повторяющиеся элементы:

```
In [1]: vlans = [10, 20, 30, 40, 100, 10]

In [2]: set(vlans)
Out[2]: {10, 20, 30, 40, 100}

In [3]: set1 = set(vlans)

In [4]: print(set1)
{40, 100, 10, 20, 30}
```

Полезные методы для работы с множествами

`add()`

Метод `add()` добавляет элемент во множество:

```
In [1]: set1 = {10,20,30,40}

In [2]: set1.add(50)

In [3]: set1
Out[3]: {10, 20, 30, 40, 50}
```

`discard()`

Метод `discard()` позволяет удалять элементы, не выдавая ошибку, если элемента в множестве нет:

```
In [3]: set1
Out[3]: {10, 20, 30, 40, 50}

In [4]: set1.discard(55)

In [5]: set1
Out[5]: {10, 20, 30, 40, 50}

In [6]: set1.discard(50)

In [7]: set1
Out[7]: {10, 20, 30, 40}
```

`clear()`

Метод `clear()` очищает множество:

```
In [8]: set1 = {10,20,30,40}

In [9]: set1.clear()

In [10]: set1
Out[10]: set()
```

Операции с множествами

Множества полезны тем, что с ними можно делать различные операции и находить объединение множеств, пересечение и так далее.

Объединение множеств можно получить с помощью метода `union()` или оператора `|`:

```
In [1]: vlans1 = {10,20,30,50,100}
In [2]: vlans2 = {100,101,102,102,200}

In [3]: vlans1.union(vlans2)
Out[3]: {10, 20, 30, 50, 100, 101, 102, 200}

In [4]: vlans1 | vlans2
Out[4]: {10, 20, 30, 50, 100, 101, 102, 200}
```

Пересечение множеств можно получить с помощью метода `intersection()` или оператора `&`:

```
In [5]: vlans1 = {10,20,30,50,100}
In [6]: vlans2 = {100,101,102,102,200}

In [7]: vlans1.intersection(vlans2)
Out[7]: {100}

In [8]: vlans1 & vlans2
Out[8]: {100}
```

Варианты создания множества

Нельзя создать пустое множество с помощью литерала (так как в таком случае это будет не множество, а словарь):

```
In [1]: set1 = {}

In [2]: type(set1)
Out[2]: dict
```

Но пустое множество можно создать таким образом:

```
In [3]: set2 = set()

In [4]: type(set2)
Out[4]: set
```

Множество из строки:

```
In [5]: set('long long long long string')
Out[5]: {' ', 'g', 'i', 'l', 'n', 'o', 'r', 's', 't'}
```

Множество из списка:

```
In [6]: set([10,20,30,10,10,30])
Out[6]: {10, 20, 30}
```

Генератор множеств:

```
In [7]: set2 = {i + 100 for i in range(10)}

In [8]: set2
Out[8]: {100, 101, 102, 103, 104, 105, 106, 107, 108, 109}

In [9]: print(set2)
{100, 101, 102, 103, 104, 105, 106, 107, 108, 109}
```

Преобразование типов

В Python есть несколько полезных встроенных функций, которые позволяют преобразовать данные из одного типа в другой.

`int()`

`int()` - преобразует строку в int:

```
In [1]: int("10")
Out[1]: 10
```

С помощью функции int можно преобразовать и число в двоичной записи в десятичную (двоичная запись должна быть в виде строки)

```
In [2]: int("11111111", 2)
Out[2]: 255
```

`bin()`

Преобразовать десятичное число в двоичный формат можно с помощью `bin()` :

```
In [3]: bin(10)
Out[3]: '0b1010'

In [4]: bin(255)
Out[4]: '0b11111111'
```

`hex()`

Аналогичная функция есть и для преобразования в шестнадцатеричный формат:

```
In [5]: hex(10)
Out[5]: '0xa'

In [6]: hex(255)
Out[6]: '0xff'
```

`list()`

Функция `list()` преобразует аргумент в список:

```
In [7]: list("string")
Out[7]: ['s', 't', 'r', 'i', 'n', 'g']

In [8]: list({1,2,3})
Out[8]: [1, 2, 3]

In [9]: list((1,2,3,4))
Out[9]: [1, 2, 3, 4]
```

set()

Функция `set()` преобразует аргумент в множество:

```
In [10]: set([1,2,3,3,4,4,4,4])
Out[10]: {1, 2, 3, 4}

In [11]: set((1,2,3,3,4,4,4,4))
Out[11]: {1, 2, 3, 4}

In [12]: set("string string")
Out[12]: {' ', 'g', 'i', 'n', 'r', 's', 't'}
```

Эта функция очень полезна, когда нужно получить уникальные элементы в последовательности.

tuple()

Функция `tuple()` преобразует аргумент в кортеж:

```
In [13]: tuple([1,2,3,4])
Out[13]: (1, 2, 3, 4)

In [14]: tuple({1,2,3,4})
Out[14]: (1, 2, 3, 4)

In [15]: tuple("string")
Out[15]: ('s', 't', 'r', 'i', 'n', 'g')
```

Это может пригодиться в том случае, если нужно получить неизменяемый объект.

str()

Функция `str()` преобразует аргумент в строку:

```
In [16]: str(10)
Out[16]: '10'
```

Например, она пригодится в ситуации, когда есть список VLANов, который надо преобразовать в одну строку, где номера перечислены через запятую.

Если сделать `join` для списка чисел, возникнет ошибка:

```
In [17]: vlans = [10, 20, 30, 40]

In [18]: ','.join(vlans)
---------------------------------------------------------
TypeError           Traceback (most recent call last)
<ipython-input-39-d705aed3f1b3> in <module>()
----> 1 ','.join(vlans)

TypeError: sequence item 0: expected string, int found
```

Чтобы исправить это, нужно преобразовать числа в строки. Это удобно делать с помощью list comprehensions:

```
In [19]: ','.join([ str(vlan) for vlan in vlans ])
Out[19]: '10,20,30,40'
```

Проверка типов

При преобразовании типов данных могут возникнуть ошибки такого рода:

```
In [1]: int('a')
---------------------------------------------------
ValueError           Traceback (most recent call last)
<ipython-input-42-b3c3f4515dd4> in <module>()
----> 1 int('a')

ValueError: invalid literal for int() with base 10: 'a'
```

Ошибка абсолютно логичная. Мы пытаемся преобразовать в десятичный формат строку 'a'.

И если тут пример выглядит, возможно, глупым, тем не менее, когда нужно, например, пройтись по списку строк и преобразовать в числа те из них, которые содержат числа, можно получить такую ошибку.

Чтобы избежать её, было бы хорошо иметь возможность проверить, с чем мы работаем.

`isdigit()`

В Python такие методы есть. Например, чтобы проверить, состоит ли строка из одних цифр, можно использовать метод `isdigit()`:

```
In [2]: "a".isdigit()
Out[2]: False

In [3]: "a10".isdigit()
Out[3]: False

In [4]: "10".isdigit()
Out[4]: True
```

Пример использования метода:

```
In [5]: vlans = ['10', '20', '30', '40', '100-200']

In [6]: [ int(vlan) for vlan in vlans if vlan.isdigit() ]
Out[6]: [10, 20, 30, 40]
```

`isalpha()`

Метод `isalpha()` позволяет проверить, состоит ли строка из одних букв:

```
In [7]: "a".isalpha()
Out[7]: True

In [8]: "a100".isalpha()
Out[8]: False

In [9]: "a--  ".isalpha()
Out[9]: False

In [10]: "a ".isalpha()
Out[10]: False
```

`isalnum()`

Метод `isalnum()` позволяет проверить, состоит ли строка из букв и цифр:

```
In [11]: "a".isalnum()
Out[1]: True

In [12]: "a10".isalnum()
Out[12]: True
```

`type()`

Иногда, в зависимости от результата, библиотека или функция может выводить разные типы объектов. Например, если объект один, возвращается строка, если несколько, то возвращается кортеж.

Нам же надо построить ход программы по-разному, в зависимости от того, была ли возвращена строка или кортеж.

В этом может помочь функция `type()`:

```
In [13]: type("string")
Out[13]: str

In [14]: type("string") is str
Out[14]: True
```

Аналогично с кортежем (и другими типами данных):

```
In [15]: type((1,2,3))
Out[15]: tuple

In [16]: type((1,2,3)) is tuple
Out[16]: True

In [17]: type((1,2,3)) is list
Out[17]: False
```

Дополнительные материалы

Документация:

- Strings. String Methods
- Lists basics. More on lists
- Tuples. More on tuples
- Sets basics. More on sets
- Dict basics. More on dicts
- Common Sequence Operations

Форматирование строк:

- Примеры использования форматирования строк
- Документация по форматированию строк

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

> Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.
>
> Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.
>
> Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 3.1

Обработать строку NAT таким образом, чтобы в имени интерфейса вместо FastEthernet было GigabitEthernet.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
NAT = "ip nat inside source list ACL interface FastEthernet0/1 overload"
```

Задание 3.2

Преобразовать строку MAC из формата XXXX:XXXX:XXXX в формат XXXX.XXXX.XXXX

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
MAC = 'AAAA:BBBB:CCCC'
```

Задание 3.3

Получить из строки CONFIG список VLANов вида: ['1', '3', '10', '20', '30', '100']

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
CONFIG = 'switchport trunk allowed vlan 1,3,10,20,30,100'
```

Задание 3.4

Из строк command1 и command2 получить список VLANов, которые есть и в команде command1 и в команде command2.

Для данного примера, результатом должен быть список: [1, 3, 100] Этот список содержит подсказку по типу итоговых данных.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
command1 = 'switchport trunk allowed vlan 1,3,10,20,30,100'
command2 = 'switchport trunk allowed vlan 1,3,100,200,300'
```

Задание 3.5

Список VLANS это список VLANов, собранных со всех устройств сети, поэтому в списке есть повторяющиеся номера VLAN.

Из списка нужно получить уникальный список VLANов, отсортированный по возрастанию номеров.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
VLANS = [10, 20, 30, 1, 2, 100, 10, 30, 3, 4, 10]
```

Задание 3.6

Обработать строку ospf_route и вывести информацию в виде:

```
Protocol:               OSPF
Prefix:                 10.0.24.0/24
AD/Metric:              110/41
Next-Hop:               10.0.13.3
Last update:            3d18h
Outbound Interface:     FastEthernet0/0
```

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
ospf_route = 'O        10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h, FastEthernet0/0'
```

Задание 3.7

Преобразовать MAC-адрес в двоичную строку (без двоеточий).

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
MAC = 'AAAA:BBBB:CCCC'
```

Задание 3.8

Преобразовать IP-адрес (переменная IP) в двоичный формат и вывести вывод столбцами, таким образом:

- первой строкой должны идти десятичные значения байтов
- второй строкой двоичные значения

Вывод должен быть упорядочен также, как в примере:

- столбцами
- ширина столбца 10 символов

Пример вывода:

```
10        1         1         1
00001010  00000001  00000001  00000001
```

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
IP = '192.168.3.1'
```

Задание 3.9

Найти индекс последнего вхождения элемента.

Например, для списка num_list, число 10 последний раз встречается с индексом 4; в списке word_list, слово 'ruby' последний раз встречается с индексом 6.

Сделать решение общим (то есть, не привязываться к конкретному элементу в конкретном списке) и проверить на двух списках, которые указаны и на разных элементах.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
num_list = [10, 2, 30, 100, 10, 50, 11, 30, 15, 7]
word_list = ['python', 'ruby', 'perl', 'ruby', 'perl', 'python', 'ruby', 'perl']
```