Типы данных в Python

В Python есть несколько стандартных типов данных:

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

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

- изменяемые (списки, словари и множества)
- неизменяемые (числа, строки и кортежи)
- упорядоченные (списки, кортежи, строки и словари)
- неупорядоченные (множества) Содержание раздела:

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
- 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 profile 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 строк, 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'
```

Также в срезе можно указывать шаг. Так можно получить нечетные числа:

```
In [28]: a = '0123456789'
In [29]: a[1::2]
Out[29]: '13579'
```

А таким образом можно получить все четные числа строки a:

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

Срезы также можно использовать для получения строки в обратном порядке:

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

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

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

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

Функция `len` позволяет получить количество символов в строке:

```
In [1]: line = 'interface Gi0/1'

In [2]: len(line)
Out[2]: 15
```

Примечание: Функция и метод отличаются тем, что метод привязан к объекту конкретного типа, а функция, как правило, более универсальная и может применяться к объектам разного типа. Например, функция `len` может применяться к строкам, спискам, словарям и так далее, а метод `startswith` относится только к строкам.

Полезные методы для работы со строками

При автоматизации очень часто надо будет работать со строками, так как конфигурационный файл, вывод команд и отправляемые команды - это строки.

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

Строки неизменяемый тип данных, поэтому все методы, которые преобразуют строку возвращают новую строку, а исходная строка остается неизменной.

Методы 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
```

Метод 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
```

Метод 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'
```

Если совпадение не найдено, метод find возвращает `-1`.

Методы 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
```

Метод 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'
```

Метод 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 пробельные символы. В этот набор символов входят: `\t\n\r\f\v`

Методу strip 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()`.

Метод split

Метод `split()` разбивает строку на части, используя как разделитель какой-то символ (или символы) и возвращает список строк:

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

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

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

В примере выше `string1.split()` разбивает строку по пробельным символам и возвращает список строк. Список записан в переменную commands.

По умолчанию в качестве разделителя используются пробельные символы (пробелы, табы, перевод строки), но в скобках можно указать любой разделитель:

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

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

В списке commands последний элемент это строка с вланами, поэтому используется индекс -1. Затем строка разбивается на части с помощью split `commands[-1].split(',')`. Так как, как разделитель указана запятая, получен такой список `['10', '20', '30', '100-200']`.

Полезная особенность метода split с разделителем по умолчанию — строка не только разделяется в список строк по пробельным символам, но пробельные символы также удаляются в начале и в конце строки:

```
In [58]: string1 = ' switchport trunk allowed vlan 10,20,30,100-200\n\n'

In [59]: string1.split()
Out[59]: ['switchport', 'trunk', 'allowed', 'vlan', '10,20,30,100-200']
```

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

```
In [60]: sh_ip_int_br = "FastEthernet0/0    15.0.15.1    YES manual up      up"

In [61]: sh_ip_int_br.split()
Out[61]: ['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 [62]: sh_ip_int_br.split(' ') Out[62]: ['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 несколько вариантов форматирования строк:

- с оператором `%` — более старый вариант
- метод `format()` — относительно новый вариант
- f-строки — новый вариант, который появился в Python 3.6.

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

Форматирование строк с методом format

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.

Значения, которые подставляются в фигурные скобки, могут быть разного типа. Например, это может быть строка, число или список:

```
In [3]: print('{}'.format('10.1.1.1'))
10.1.1.1

In [4]: print('{}'.format(100))
100

In [5]: print('{}'.format([10, 1, 1,1]))
[10, 1, 1, 1]
```

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

Выравнивание по левой стороне:

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

Шаблон для вывода может быть и многострочным:

```
In [6]: ip_template = '''
   ...: IP address:
   ...: {}
   ...: '''

In [7]: print(ip_template.format('10.1.1.1'))

IP address:
10.1.1.1
```

С помощью форматирования строк можно также влиять на отображение чисел.

Например, можно указать, сколько цифр после запятой выводить:

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

С помощью форматирования строк можно конвертировать числа в двоичный формат:

```
In [11]: '{:b} {:b} {:b} {:b}'.format(192, 100, 1, 1)
Out[11]: '11000000 1100100 1 1'
```

При этом по-прежнему можно указывать дополнительные параметры, например, ширину столбца:

```
In [12]: '{:8b} {:8b} {:8b} {:8b}'.format(192, 100, 1, 1)
Out[12]: '11000000 1100100        1        1'
```

А также можно указать, что надо дополнить числа нулями, вместо пробелов:

```
In [13]: '{:08b} {:08b} {:08b} {:08b}'.format(192, 100, 1, 1)
Out[13]: '11000000 01100100 00000001 00000001'
```

В фигурных скобках можно указывать имена. Это позволяет передавать аргументы в любом порядке, а также делает шаблон более понятным:

```
In [15]: '{ip}/{mask}'.format(mask=24, ip='10.1.1.1')
Out[15]: '10.1.1.1/24'
```

Еще одна полезная возможность форматирования строк - указание номера аргумента:

```
In [16]: '{1}/{0}'.format(24, '10.1.1.1')
Out[16]: '10.1.1.1/24'
```

За счет этого, например, можно избавиться от повторной передачи одних и тех же значений:

```
In [19]: ip_template = '''
    ...: IP address:
    ...: {:<8} {:<8} {:<8} {:<8}
    ...: {:08b} {:08b} {:08b} {:08b}
    ...: '''

In [20]: print(ip_template.format(192, 100, 1, 1, 192, 100, 1, 1))

IP address:
192      100      1        1
11000000 01100100 00000001 00000001
```

В примере выше октеты адреса приходится передавать два раза - один для отображения в десятичном формате, а второй - для двоичного.

Указав индексы значений, которые передаются методу format, можно избавиться от дублирования:

```
In [21]: ip_template = '''
    ...: IP address:
    ...: {0:<8} {1:<8} {2:<8} {3:<8}
    ...: {0:08b} {1:08b} {2:08b} {3:08b}
    ...: '''

In [22]: print(ip_template.format(192, 100, 1, 1))

IP address:
192      100      1        1
11000000 01100100 00000001 00000001
```

Форматирование строк с помощью f-строк

В Python 3.6 добавился новый вариант форматирования строк - f-строки или интерполяция строк. F-строки позволяют не только подставлять какие-то значения в шаблон, но и позволяют выполнять вызовы функций, методов и т.п.

Во многих ситуациях f-строки удобней и проще использовать, чем format, кроме того, f-строки работают быстрее, чем format и другие методы форматирования строк.

Синтаксис

F-строки — это литерал строки с буквой `f` перед ним. Внутри f-строки в паре фигурных скобок указываются имена переменных, которые надо подставить:

```
In [1]: ip = '10.1.1.1'

In [2]: mask = 24

In [3]: f"IP: {ip}, mask: {mask}"
Out[3]: 'IP: 10.1.1.1, mask: 24'

Аналогичный результат с format можно получить так:
``"IP: {ip}, mask: {mask}".format(ip=ip, mask=mask)``.
```

Очень важное отличие f-строк от format: f-строки — это выражение, которое выполняется, а не просто строка. То есть, в случае с ipython, как только мы написали выражение и нажали Enter, оно выполнилось и вместо выражений `{ip}` и `{mask}` подставились значения переменных.

Поэтому, например, нельзя сначала написать шаблон, а затем определить переменные, которые используются в шаблоне:

```
In [1]: f"IP: {ip}, mask: {mask}"
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-e6f8e01ac9c4> in <module>() ---
-> 1 f"IP: {ip}, mask: {mask}"

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

Кроме подстановки значений переменных, в фигурных скобках можно писать выражения:

```
In [5]: first_name = 'William'

In [6]: second_name = 'Shakespeare'

In [7]: f"{first_name.upper()} {second_name.upper()}"
Out[7]: 'WILLIAM SHAKESPEARE'
```

После двоеточия в f-строках можно указывать те же значения, что и при использовании format:

```
In [9]: oct1, oct2, oct3, oct4 = [10, 1, 1, 1]
In [10]: print(f'''
    ...: IP address:
    ...: {oct1:<8} {oct2:<8} {oct3:<8} {oct4:<8}
    ...: {oct1:08b} {oct2:08b} {oct3:08b} {oct4:08b}''')

IP address:
10       1        1        1
00001010 00000001 00000001 00000001
```

Предупреждение: Так как для полноценного объяснения f-строк, надо показывать примеры с циклами и работой с объектами, которые еще не рассматривались, это тема также есть в разделе *Форматирование строк с помощью f-строк* с дополнительными примерами и пояснениями.

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
↪команду из списка? [y/n]'
```

Список (List)

Список в Python это:

- последовательность элементов, разделенных между собой запятой и заключенных в квадратные скобки
- изменяемый упорядоченный тип данных Примеры списков:

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

Создание списка с помощью литерала:

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

Примечание: 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']
```

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'
```

Функция len возвращает количество элементов в списке:

```
In [1]: items = [1, 2, 3]

In [2]: len(items)
Out[2]: 3
```

А функция sorted сортирует элементы списка по возрастанию и возвращает новый список с отсортированными элементами:

```
In [1]: names = ['John', 'Michael', 'Antony']

In [2]: sorted(names)
Out[2]: ['Antony', 'John', 'Michael']
```

Полезные методы для работы со списками

Список - это изменяемый тип данных, поэтому очень важно обращать внимание на то, что большинство методов для работы со списками меняют список на месте, при этом ничего не возвращая.

join()

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

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

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

Примечание: Метод join на самом деле относится к строкам, но так как значение ему надо передавать как список, он рассматривается тут.

---

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']
```

Метод append меняет список на месте и ничего не возвращает.

extend()

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

У этих способов есть важное отличие - extend меняет список, к которому применен метод, а суммирование возвращает новый список, который состоит из двух.

Метод extend:

```
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']
```

Суммирование списков:

```
In [27]: vlans = ['10', '20', '30', '100-200'] In [28]: vlans2 = ['300', '400',

'500']

In [29]: vlans + vlans2
Out[29]: ['10', '20', '30', '100-200', '300', '400', '500']
```

Обратите внимание на то, что при суммировании списков в ipython появилась строка Out. Это означает, что результат суммирования можно присвоить в переменную:

```
In [30]: result = vlans + vlans2

In [31]: result
Out[31]: ['10', '20', '30', '100-200', '300', '400', '500']
```

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']
```

sort()

Метод sort сортирует список на месте:

```
In [40]: vlans = [1, 50, 10, 15]

In [41]: vlans.sort()

In [42]: vlans
Out[42]: [1, 10, 15, 50]
```

Словарь (Dictionary)

Словари – это изменяемый упорядоченный тип данных:

- данные в словаре – это пары `ключ: значение`
- доступ к значениям осуществляется по ключу, а не по номеру, как в списках
- данные в словаре упорядочены по порядку добавления элементов
- 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

Примечание: В других языках программирования тип данных подобный словарю может называться ассоциативный массив, хеш или хеш-таблица.

Пример словаря:

```
london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}
```

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'
```

Функция sorted сортирует ключи словаря по возрастанию и возвращает новый список с отсортированными ключами:

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

In [2]: sorted(london)
Out[2]: ['location', 'name', 'vendor']
```

Полезные методы для работы со словарями

clear()

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

```
In [1]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco',
,→'model': '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]: {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco',
 'ios': None}
```

Если ключ есть, setdefault возвращает значение, которое ему соответствует:

```
In [25]: london.setdefault('name')
Out[25]: 'London1'
```

Второй аргумент позволяет указать, какое значение должно соответствовать ключу:

```
In [26]: model = london.setdefault('model', 'Cisco3580')

In [27]: print(model)
Cisco3580

In [28]: london
Out[28]:
{'name': 'London1',
 'location': 'London Str',
 'vendor': 'Cisco',
 'ios': None,
 'model': 'Cisco3580'}
```

Метод setdefault заменяет такую конструкцию:

```
In [30]: if key in london: ...:
      value = london[key] ...:
    else:
    ...:     london[key] = 'somevalue'
    ...:     value = london[key]
    ...:
```

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',
↪'Cisco')])
```

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, location 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]: {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco', 'ios':
↪'15.2'}
```

Аналогичным образом можно обновить значения:

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

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

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]: {'model': '4451', 'ios': '15.4'}
```

Второй вариант создания словаря с помощью dict:

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

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

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]:
{'hostname': None,
'location': None,
'vendor': None,
'model': None,
'ios': None,
'ip': 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]: {'ISR2811': 0, 'ISR2911': 0, 'ISR2921': 0, 'ASR9002': 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 [10]: router_models = ['ISR2811', 'ISR2911', 'ISR2921', 'ASR9002']

In [11]: routers = dict.fromkeys(router_models, [])
    ...:

In [12]: routers
Out[12]: {'ISR2811': [], 'ISR2911': [], 'ISR2921': [], 'ASR9002': []}

In [13]: routers['ASR9002'].append('london_r1')

In [14]: routers
Out[14]:
{'ISR2811': ['london_r1'],
 'ISR2911': ['london_r1'],
 'ISR2921': ['london_r1'],
 'ASR9002': ['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.

Примечание: Для такой задачи лучше подходит генератор словаря. Смотри раздел *List, dict, set comprehensions*

Кортеж (Tuple)

Кортеж в Python это:

- последовательность элементов, которые разделены между собой запятой и заключены в скобки
- неизменяемый упорядоченный тип данных

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
```

Функция sorted сортирует элементы кортежа по возрастанию и возвращает новый список с отсортированными элементами:

```
In [2]: tuple_keys = ('hostname', 'location', 'vendor', 'model', 'ios', 'ip')

In [3]: sorted(tuple_keys)
Out[3]: ['hostname', 'ios', 'ip', 'location', 'model', 'vendor']
```

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}
```

Булевы значения

Булевы значения в Python это две константы `True` и `False`.
В Python истинными и ложными значениями считаются не только True и False.

- истинное значение:
  - любое ненулевое число
  - любая непустая строка – любой непустой объект
- ложное значение:
  - 0
  - None
  - пустая строка
  - пустой объект

Остальные истинные и ложные значения, как правило, логически следуют из условия.
Для проверки булевого значения объекта, можно воспользоваться `bool`:

```
In [2]: items = [1, 2, 3]
In [3]: empty_list = []
In [4]: bool(empty_list)
Out[4]: False

In [5]: bool(items)
Out[5]: True

In [6]: bool(0)
Out[6]: False

In [7]: bool(1)
Out[7]: False
```

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'
```

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
```

isalpha()

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

```
In [8]: "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 или цифр:

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

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

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