Основы Python

Типы данных в Python

Типы данных в Python

В Python есть несколько стандартных типов данных:

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

Типы данных в Python

Изменяемые:
- Списки
- Словари
- Множества

Неизменяемые
- Числа
- Строки
- Кортежи

Упорядоченные:
- Списки
- Кортежи
- Строки

Неупорядоченные:
- Словари
- Множества

Числа

Пример различных типов числовых значений:

int (40, -80, 0x0800)

float (1.5, -30.7)

long (52934861L)

```
In [1]: 3 * 5
Out[1]: 15

In [2]: 3**5
Out[2]: 243

In [3]: 100%3
Out[3]: 1

In [4]: 3**50
Out[4]: 717897987691852588770249L

In [5]: 10.0/3
Out[5]: 3.3333333333333335
```

Строки (Strings)

Строки это неизменяемый, упорядоченный тип данных. Строка в Python это последовательность символов, заключенная в кавычки.

```
In [6]: 'Hello'
Out[6]: 'Hello'

In [7]: "Hello"
Out[7]: 'Hello'

In [8]: """
   ...: 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
   ...: """

Out[8]: '\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'
```

Список (List)

Список это изменяемый упорядоченный тип данных.
Список в Python это последовательность элементов, разделенных между собой запятой и заключенных в квадратные скобки.

```
In [9]: [1,2,3,4,5]
Out[9]: [1, 2, 3, 4, 5]

In [10]: ['switchport', 'mode', 'access']
Out[10]: ['switchport', 'mode', 'access']

In [11]: [[1, 2, 3], 4, 5]
Out[11]: [[1, 2, 3], 4, 5]
```

Словарь (Dictionary)

Словари – это изменяемый, неупорядоченный тип данных.

```
In [12]: london = {'name': 'London1', 'location': 'London Str',
'vendor': 'Cisco', 'model': '4451', 'IOS': '15.4'}

In [13]: london = {
    ....:           'id': 1,
    ....:           'name':'London',
    ....:           'IT_VLAN':320,
    ....:           'User_VLAN':1010,
    ....:           'Mngmt_VLAN':99,
    ....:           'to_name': None,
    ....:           'to_id': None,
    ....:           'port':'G1/0/11'
    ....: }
```

Кортеж (Tuple)

Кортеж это **неизменяемый** упорядоченный тип данных.

Кортеж в Python это последовательность элементов, разделенных между собой запятой и заключенных в скобки.

```
In [14]: (1,2,4,5,6,7)
Out[14]: (1, 2, 4, 5, 6, 7)

In [15]: 6,7,8,8,9
Out[15]: (6, 7, 8, 8, 9)

In [16]: (2,)
Out[16]: (2,)
```

Множество (Set)

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

Множество в Python это последовательность элементов, разделенные между собой запятой и заключенные в фигурные скобки.

```
In [17]: vlans = [10, 20, 30, 40, 100, 10]

In [18]: set(vlans)
Out[18]: set([40, 100, 10, 20, 30])

In [19]: {1,2,2,3,5,4,6,3}
Out[19]: set([1, 2, 3, 4, 5, 6])
```

Упорядоченные типы данных

Упорядоченные типы данных

В Python такие типа данных являются упорядоченными:

- Списки
- Кортежи
- Строки

Дла всех упорядоченных типов данных есть общие операции.

Операции работающие для всех упорядоченных типов данных

`x in s`	True если элемент в s равен *x*, иначе False
`x not in s`	False если элемент в s равен *x*, иначе True
`s + t`	объединение *s* и *t*
`s * n, n * s`	эквивалентно добавлению *s* к себе *n* раз
`s[i]`	i-ый элемент *s*, начиная с 0
`s[i:j]`	срез *s* с *i* до *j*
`s[i:j:k]`	срез *s* с *i* до *j* с шагом *k*
`len(s)`	длинна *s*
`min(s)`	наименьший элемент *s*
`max(s)`	наибольший элемент *s*
`s.index(x)`	индекс первого местонахождения *x* в *s*
`s.count(x)`	количество элемента *x* в *s*

В таблице, s и t это последовательности одного типа; n, i и j числа

Операции работающие для всех упорядоченных типов данных на примере списка

```
In [26]: s = [0,1,2,3,4,5]

In [27]: 1 in s
Out[27]: True

In [28]: 1 not in s
Out[28]: False

In [29]: s + [6,7,8]
Out[29]: [0, 1, 2, 3, 4, 5, 6, 7, 8]

In [30]: s * 2
Out[30]: [0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5]

In [31]: 2 * s
Out[31]: [0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5]

In [32]: s[3]
Out[32]: 3
```

```
In [34]: s[1:4]
Out[34]: [1, 2, 3]

In [35]: s[1:5:2]
Out[35]: [1, 3]

In [36]: len(s)
Out[36]: 6

In [37]: min(s)
Out[37]: 0

In [38]: max(s)
Out[38]: 5

In [39]: s.index(4)
Out[39]: 4

In [40]: s.count(4)
Out[40]: 1
```

Операции работающие для всех упорядоченных типов данных на примере строк

```
In [43]: s = "String is a sequence"

In [44]: 'is' in s
Out[44]: True

In [45]: 'is' not in s
Out[45]: False

In [46]: s + " of letters"
Out[46]: 'String is a sequence of letters'

In [47]: s * 2
Out[47]: 'String is a sequenceString is a
sequence'

In [48]: 2 * s
Out[48]: 'String is a sequenceString is a
sequence'

In [49]: s[3]
Out[49]: 'i'
```

```
In [50]: s[7:10]
Out[50]: 'is '

In [51]: s[1:14:2]
Out[51]: 'tigi  e'

In [52]: len(s)
Out[52]: 20

In [53]: min(s)
Out[53]: ' '

In [54]: max(s)
Out[54]: 'u'

In [55]:
s.index('is')
Out[55]: 7

In [56]: s.count('i')
Out[56]: 2
```

Строки

Строки (Strings)

Строки это неизменяемый, упорядоченный тип данных. Строка в Python это последовательность символов, заключенная в кавычки.

```
In [6]: 'Hello'
Out[6]: 'Hello'

In [7]: "Hello"
Out[7]: 'Hello'

In [8]: """
   ...: 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
   ...: """

Out[8]: '\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'
```

Строки (Strings)

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

```
In [14]: string1 = 'interface FastEthernet1/0'

In [15]: string1[0]
Out[15]: 'i'

In [16]: string1[1]
Out[16]: 'n'
```

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 [17]: string1[-1]
Out[17]: '0'
```

Строки (Strings)

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

```
In [14]: string1 = 'interface FastEthernet1/0'

In [18]: string1[0:9]
Out[18]: 'interface'

In [19]: string1[10:22]
Out[19]: 'FastEthernet'
```

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

```
In [20]:  string1[10:]
Out[20]: 'FastEthernet1/0'
```

Срезать три последних символа строки:

```
In [21]: string1[-3:]
Out[21]: '1/0'
```

Строка в обратном порядке:

```
In [23]: a[::]
Out[23]: '0123456789'

In [24]: a[::-1]
Out[24]: '9876543210'
```

Полезные методы для работы со строками

По сути, конфигурационный файл это просто текстовый файл и поэтому, при автоматизации, очень часто надо будет работать со строками. Знание различных методов (то есть, действий), которые можно применять к строкам, очень сильно облегчает жизнь.

Конечно, чаще всего, эти методы будут применяться в циклах, при обработке файла. Но уже тут должно быть понятно, что многие вещи можно сделать довольно просто.

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).

Это связано с тем, что строка в Python это неизменяемый тип данных.

Полезные методы для работы со строками

Преобразование регистра строки (методы upper(), lower(), swapcase(), capitalize()):

```
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'
```

Метод .lower() особенно полезен при получении данных от пользователя. Так как он позволяет преобразовать данные в нижний регистр и, соответственно не учитывать регистр далее, при обработке введенных данных.

Полезные методы для работы со строками

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
```

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'
```

Полезные методы для работы со строками

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
```

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'
```

Полезные методы для работы со строками

Часто, при обработки файла, файл открывается построчно. Но в конце каждый строки, как правило есть какие-то спецсимволы (а могут быть и вначале). 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'
```

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

Полезные методы для работы со строками

Метод split() разбивает строку на части, используя как разделитель какой-то символ (или символы). По умолчанию, в качестве разделителя используется пробел. Но в скобках можно указать любой разделитель, который нужен.

В итоге строка будет разбита на части и представлена в виде частей, которые содержатся в списке.

```
In [51]: string1 = 'switchport trunk allowed vlan 10,20,30,100-200'

In [52]: string1.split()
Out[52]: ['switchport', 'trunk', 'allowed', 'vlan', '10,20,30,100-200']
```

Еще один пример:

```
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']
```

Списки

Список (List)

Список это изменяемый упорядоченный тип данных.
Список в Python это последовательность элементов, разделенных между собой запятой и заключенных в квадратные скобки.

```
In [9]: [1,2,3,4,5]
Out[9]: [1, 2, 3, 4, 5]

In [10]: ['switchport', 'mode', 'access']
Out[10]: ['switchport', 'mode', 'access']

In [11]: [[1, 2, 3], 4, 5]
Out[11]: [[1, 2, 3], 4, 5]
```

Список (List)

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]
```

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']
```

Полезные методы для работы со списками

При рассмотрении полезных методов для строк, мы остановились на том, что получили итоговый список vlans:

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

Теперь, допустим, что после необходимых операций с списком VLAN(отбросим последний диапазон), нужно опять записать их файл. То есть надо сделать так, чтобы теперь все значений списка были собраны в одну строку, но записаны через запятую (сделать операцию обратную split()).

Метод join() позволяет собрать 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'
```

Полезные методы для работы со списками

Метод append() позволяет добавить в конец списка указанные 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']
```

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']
```

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

Полезные методы для работы со списками

Метод pop() позволяет удалить элемент, который соответствует указанному номеру. По умолчанию берется последний элемент списка. При этом метод выводит 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']
```

В методе remove() надо указывать сам элемент, который надо удалить, а не его номер в списке. Кроме того, remove() не отображает удаленный элемент:

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

In [32]: 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

In [33]: vlans.remove('20')

In [34]: vlans
Out[34]: ['10', '30', '100-200']
```

Полезные методы для работы со списками

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

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

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

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]
```

Создание списка с помощью функции list():

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

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

Генераторы списков (list comprehention):

```
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']
```

Словари

Словарь (Dictionary)

Словари – это изменяемый, неупорядоченный тип данных.

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 неупорядоченны, 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

Словарь (Dictionary)

Пример словаря:

```
london = {'name': 'London1', 'location': 'London Str',
'vendor': 'Cisco', 'model': '4451', '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'
}
```

Словарь (Dictionary)

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'}
```

Словарь (Dictionary)

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'
        },
    '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

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]: {}
```

Метод copy() позволяет создать полную копию словаря.

```
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'
```

Полезные методы для работы со словарями

Если при обращении к словарю указывается ключ, которого нет в словаре, мы получаем ошибку:

```
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() позволяет запросить значение, но если его нет, вместо ошибки возвращается указанное значение (по умолчанию возвращается None):

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

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

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

Полезные методы для работы со словарями

Методы keys(), values(), items():

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

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

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

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

Удалить ключ и значение:

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

In [29]: del(london['name'])

In [30]: london
Out[30]: {'location': 'London Str', 'vendor': 'Cisco'}
```

Варианты создания словаря

Словарь можно создать с помощью литерала:

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

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'}
```

Варианты создания словаря

Если необходимо 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, None)

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

Генераторы словарей. Сгенерируем словарь с нулевыми значениями, как в предыдущем примере:

```
In [16]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']

In [17]: d = {x: None for x in d_keys}

In [18]: d
Out[18]:
{'IOS': None,
 'IP': None,
 'hostname': None,
 'location': None,
 'model': None,
 'vendor': None}
```

Словарь из двух списков

```
In [4]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']
In [5]: d_values = ['london_r1', '21 New Globe Walk', 'Cisco', '4451', '15.4',
'10.255.0.1']

In [6]: zip(d_keys,d_values)
Out[6]:
[('hostname', 'london_r1'),
 ('location', '21 New Globe Walk'),
 ('vendor', 'Cisco'),
 ('model', '4451'),
 ('IOS', '15.4'),
 ('IP', '10.255.0.1')]

In [7]: dict(zip(d_keys,d_values))
Out[7]:
{'IOS': '15.4',
 'IP': '10.255.0.1',
 'hostname': 'london_r1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
In [8]: r1 = dict(zip(d_keys,d_values))

In [9]: r1
Out[9]:
{'IOS': '15.4',
 'IP': '10.255.0.1',
 'hostname': 'london_r1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
```

Словарь из двух списков

```
In [10]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']

In [11]: data = {
   ....: 'r1': ['london_r1', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.255.0.1'],
   ....: 'r2': ['london_r2', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.255.0.2'],
   ....: 'sw1': ['london_sw1', '21 New Globe Walk', 'Cisco', '3850', '3.6.XE', '10.255.0.101']
   ....: }

In [12]: london_co = {}

In [13]: for k in data.keys():
   ....:     london_co[k] = dict(zip(d_keys,data[k]))
   ....:

In [14]: london_co
Out[14]:
{'r1': {'IOS': '15.4',
  'IP': '10.255.0.1',
  'hostname': 'london_r1',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'r2': {'IOS': '15.4',
  'IP': '10.255.0.2',
  'hostname': 'london_r2',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'sw1': {'IOS': '3.6.XE',
  'IP': '10.255.0.101',
  'hostname': 'london_sw1',
  'location': '21 New Globe Walk',
  'model': '3850',
  'vendor': 'Cisco'}}
```

Кортеж

Кортеж (Tuple)

Кортеж это неизменяемый упорядоченный тип данных.

Кортеж в Python это последовательность элементов, разлделенные между собой запятой и заключенные в скобки.

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

То что кортеж неизменяем, может быть очень полезно в некоторых случаях.

Кортеж (Tuple)

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')
```

Кортеж (Tuple)

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

Множество (Set)

Множество это изменяемый неупорядоченный тип данных. В множестве всегда содержатся только уникальные элементы. Множество в Python это последовательность элементов, разделенные между собой запятой и заключенные 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
set([40, 100, 10, 20, 30])
```

Полезные методы для работы с множествами

Метод add() добавляет элемент в множество:

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

In [2]: set1.add(50)

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

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}
```

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}
```

Операции с множествами

Уникальные элементы в множестве:

```
In [11]: vlans1 = {10,20,30,100,101}

In [12]: vlans2 = {100,101,102,102,200}

In [13]: vlans1.difference(vlans2)
Out[13]: {10, 20, 30}

In [14]: vlans2.difference(vlans1)
Out[14]: {102, 200}

In [15]: vlans1.symmetric_difference(vlans2)
Out[15]: {10, 20, 30, 102, 200}
```

Добавить в множество vlans1 элементы из множества vlans2:

```
In [16]: vlans1.update(vlans2)

In [17]: vlans1
Out[17]: {10, 20, 30, 100, 101, 102, 200}
```

Python для сетевых инженеров

Автор курса: Наташа Самойленко
nataliya.samoylenko@gmail.com