PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ

ОСНОВЫ PYTHON

СИНТАКСИС PYTHON

Отступы имеют значение. Они определяют:

- какие выражения попадают в блок кода
- когда блок кода заканчивается

Tab или пробел:

- лучше использовать пробелы (настроить редактор)
- количество пробелов должно быть одинаковым в одном блоке:
  - лучше во всем коде
  - обычно используются 2-4 пробела (в курсе используются 4 пробела)

СИНТАКСИС 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")
```

КОММЕНТАРИИ

Однострочный комментарий:

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

Многострочный комментарий:

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

ИНТЕРПРЕТАТОР IPYTHON

ИНТЕРПРЕТАТОР IPYTHON

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

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

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

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

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

ИНТЕРПРЕТАТОР IPYTHON

Функция print()

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

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

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

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

IPYTHON MAGIC

История 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
```

IPYTHON HELP

```
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.
```

IPYTHON HELP

```
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
```

ПЕРЕМЕННЫЕ

ПЕРЕМЕННЫЕ

Переменные в Python:

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

Имя переменной:

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

ПЕРЕМЕННЫЕ

```
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
```

ПЕРЕМЕННЫЕ

Переменные являются ссылками на область памяти:

```
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
```

ПЕРЕМЕННЫЕ

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

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

ТИПЫ ДАННЫХ В PYTHON

- Упорядоченные:
  - Списки
  - Кортежи
  - Строки
- Неупорядоченные:
  - Словари
  - Множества

ЧИСЛА

ЧИСЛА

Пример различных типов числовых значений:

- int (40, -80)
- float (1.5, -30.7)

```
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
```

ЧИСЛА

Деление int и float:

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

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

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

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

ЧИСЛА

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
```

ЧИСЛА

Конвертация в тип int:

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

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

Во втором аргументе можно указывать систему исчисления:

```
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
```

ЧИСЛА

Функция bin():

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

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

Функция hex():

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

СТРОКИ

СТРОКИ

Строка в Python:

- последовательность символов, заключенная в кавычки
- неизменяемый, упорядоченный тип данных

```
In [1]: 'Hello'
Out[1]: 'Hello'

In [2]: "Hello"
Out[2]: 'Hello'

In [3]: 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 - УПОРЯДОЧЕННЫЙ ТИП ДАННЫХ

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

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

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

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

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

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

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

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

СТРОКИ - УПОРЯДОЧЕННЫЙ ТИП ДАННЫХ

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

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

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

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

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

МЕТОДЫ РАБОТЫ СО СТРОКАМИ

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 **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
```

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

```
In [36]: string1 = 'interface FastEthernet0/1'

In [37]: string1.find('Fast')
Out[37]: 10

In [38]: string1[string1.find('Fast')::]
Out[38]: 'FastEthernet0/1'
```

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'
```

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 **split()**:

```
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']

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']
```

ФОРМАТИРОВАНИЕ СТРОК

ФОРМАТИРОВАНИЕ СТРОК

Существует два варианта форматирования строк:

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

Пример использования метода format:

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

Аналогичный пример с оператором %:

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

ФОРМАТИРОВАНИЕ СТРОК

Выравнивание 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
```

ФОРМАТИРОВАНИЕ СТРОК

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'
```

СПИСКИ

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']
```

Список - упорядоченный тип данных:

```
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']
```

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'
```

МЕТОДЫ ДЛЯ РАБОТЫ СО СПИСКАМИ

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'
```

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']
```

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']
```

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

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']
```

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']
```

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
```

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]
```

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(3)]

In [5]: list2
Out[6]:
['FastEthernet0/0',
 'FastEthernet0/1',
 'FastEthernet0/2']
```

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
- 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': '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'
}
```

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'
    },
    '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'
```

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]: 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')])
```

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

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

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

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

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

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

```
In [1]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco', 'model': '4451', 'IOS': '1

In [2]: london.clear()

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

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'}
```

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'
```

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 указывать другое значение, вместо None:

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

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'}
```

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'}
```

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, None)

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

Генераторы словарей:

```
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}
```

КОРТЕЖ

КОРТЕЖ (TUPLE)

Кортеж это неизменяемый упорядоченный тип данных.

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

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')
```

МНОЖЕСТВО

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}
```

МЕТОДЫ РАБОТЫ С МНОЖЕСТВАМИ

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}
```

Метод **clear()** очищает множество:

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

In [9]: set1.clear()

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

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

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}
```

ОПЕРАЦИИ С МНОЖЕСТВАМИ

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

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
```

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

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

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

Аналогичная функция есть и для преобразования в шестнадцатиричный формат:

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

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

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]
```

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.

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 пригодится в том случае, если нужно получить неизменяемый объект.

Функция str() преобразует аргумент в строку:

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

list comprehensions:

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

In [18]: ','.join([ str(vlan) for vlan in vlans ])
Out[18]: '10,20,30,40'
```

ПРОВЕРКА ТИПОВ

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]
```

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
```

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
```