PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ

ПОЛЕЗНЫЕ ВСТРОЕННЫЕ ФУНКЦИИ

ФУНКЦИЯ PRINT

ФУНКЦИЯ PRINT

Функция print выводит все элементы, разделяя их значением sep, и завершает вывод значением end.

```
print(*items, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Все аргументы, которые управляют поведением функции print, надо передавать как ключевые, а не позиционные.

ФУНКЦИЯ PRINT

Все элементы, которые передаются как аргументы, конвертируются в строки:

```
In [4]: def f(a):
   ...:     return a
   ...:

In [5]: print(1, 2, f, range(10))
1 2 <function f at 0xb4de926c> range(0, 10)
```

ФУНКЦИЯ PRINT

Для функций f и range результат равнозначен применению str():

```
In [6]: str(f)
Out[6]: '<function f at 0xb4de926c>'

In [7]: str(range(10))
Out[7]: 'range(0, 10)'
```

SEP

Параметр sep контролирует то, какой разделитель будет использоваться между элементами.

По умолчанию используется пробел:

```
In [8]: print(1, 2, 3)
1 2 3
```

SEP

Можно изменить значение sep на любую другую строку:

```
In [9]: print(1, 2, 3, sep='|')
1|2|3

In [10]: print(1, 2, 3, sep='\n')
1
2
3

In [11]: print(1, 2, 3, sep='\n'+'-'*10+'\n')
1
----------
2
----------
3
```

SEP

В некоторых ситуациях функция print может заменить метод join:

```
In [12]: items = [1,2,3,4,5]

In [13]: print(*items, sep=', ')
1, 2, 3, 4, 5
```

END

Параметр end контролирует то, какое значение выведется после вывода всех элементов.

По умолчанию используется перевод строки:

```
In [19]: print(1,2,3)
1 2 3
```

Можно изменить значение end на любую другую строку:

```
In [20]: print(1,2,3, end='\n'+'-'*10)
1 2 3
----------
```

FILE

Параметр file контролирует то, куда выводятся значения функции print. По умолчанию все выводится на стандартный поток вывода - sys.stdout.

Но Python позволяет передавать file как аргумент любой объект с методом write(string). За счет этого с помощью print можно записывать строки в файл:

```
In [1]: f = open('result.txt', 'w')

In [2]: for num in range(10):
   ...:     print('Item {}'.format(num), file=f)
   ...:

In [3]: f.close()

In [4]: cat result.txt
Item 0
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
```

FLUSH

По умолчанию при записи в файл или выводе на стандартный поток вывода вывод буферизируется. Функция print позволяет отключать буферизацию.

Пример скрипта, который выводит число от 0 до 10 каждую секунду (файл print_nums.py):

```
import time

for num in range(10):
    print(num)
    time.sleep(1)
```

FLUSH

Теперь, аналогичный скрипт, но числа будут выводиться в одной строке (файл print_nums_oneline.py):

```
import time

for num in range(10):
    print(num, end=' ')
    time.sleep(1)
```

Числа не выводятся по одному в секунду, а выводятся все через 10 секунд.

FLUSH

Чтобы скрипт отрабатывал как нужно, необходимо установить flush равным True (файл print_nums_oneline_fixed.py):

```
import time

for num in range(10):
    print(num, end=' ', flush=True)
    time.sleep(1)
```

ФУНКЦИЯ RANGE

ФУНКЦИЯ RANGE

Функция range возвращает неизменяемую последовательность чисел в виде объекта range.

Синтаксис функции:

```
range(stop)
range(start, stop)
range(start, stop, step)
```

Параметры функции:

- **start** - с какого числа начинается последовательность. По умолчанию - 0
- **stop** - до какого числа продолжается последовательность чисел. Указанное число не включается в диапазон
- **step** - с каким шагом растут числа. По умолчанию 1

ФУНКЦИЯ RANGE

Функция range хранит только информацию о значениях start, stop и step и вычисляет значения по мере необходимости. Это значит, что, независимо от размера диапазона, который описывает функция range, она всегда будет занимать фиксированный объем памяти.

ФУНКЦИЯ RANGE

Самый простой вариант range - передать только значение stop:

```
In [1]: range(5)
Out[1]: range(0, 5)

In [2]: list(range(5))
Out[2]: [0, 1, 2, 3, 4]
```

ФУНКЦИЯ RANGE

Если передаются два аргумента, то первый используется как start, а второй - как stop:

```
In [3]: list(range(1, 5))
Out[3]: [1, 2, 3, 4]
```

И, чтобы указать шаг последовательности, надо передать три аргумента:

```
In [4]: list(range(0, 10, 2))
Out[4]: [0, 2, 4, 6, 8]

In [5]: list(range(0, 10, 3))
Out[5]: [0, 3, 6, 9]
```

ФУНКЦИЯ RANGE

С помощью range можно генерировать и убывающие последовательности чисел:

```
In [6]: list(range(10, 0, -1))
Out[6]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

In [7]: list(range(5, -1, -1))
Out[7]: [5, 4, 3, 2, 1, 0]
```

Для получения убывающей последовательности надо использовать отрицательный шаг и соответственно указать start - большим числом, а stop - меньшим.

В убывающей последовательности шаг тоже может быть разным:

```
In [8]: list(range(10, 0, -2))
Out[8]: [10, 8, 6, 4, 2]
```

ФУНКЦИЯ RANGE

Функция поддерживает отрицательные значения start и stop:

```
In [9]: list(range(-10, 0, 1))
Out[9]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]

In [10]: list(range(0, -10, -1))
Out[10]: [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

ФУНКЦИЯ RANGE

Объект range поддерживает все операции, которые поддерживают последовательности в Python, кроме сложения и умножения.

Проверка, входит ли число в диапазон, который описывает range:

```
In [11]: nums = range(5)

In [12]: nums
Out[12]: range(0, 5)

In [13]: 3 in nums
Out[13]: True

In [14]: 7 in nums
Out[14]: False
```

Начиная с версии Python 3.2, эта проверка выполняется за постоянное время (O(1)).

ФУНКЦИЯ RANGE

Можно получить конкретный элемент диапазона:

```
In [15]: nums = range(5)

In [16]: nums[0]
Out[16]: 0

In [17]: nums[-1]
Out[17]: 4
```

ФУНКЦИЯ RANGE

Range поддерживает срезы:

```
In [18]: nums = range(5)

In [19]: nums[1:]
Out[19]: range(1, 5)

In [20]: nums[:3]
Out[20]: range(0, 3)
```

ФУНКЦИЯ RANGE

Можно получить длину диапазона:

```
In [21]: nums = range(5)

In [22]: len(nums)
Out[22]: 5
```

ФУНКЦИЯ RANGE

А также минимальный и максимальный элемент:

```
In [23]: nums = range(5)

In [24]: min(nums)
Out[24]: 0

In [25]: max(nums)
Out[25]: 4
```

ФУНКЦИЯ RANGE

Кроме того, объект range поддерживает метод index:

```
In [26]: nums = range(1, 7)

In [27]: nums.index(3)
Out[27]: 2
```

ФУНКЦИЯ SORTED

ФУНКЦИЯ SORTED

Функция `sorted()` возвращает новый отсортированный список, который получен из итерируемого объекта, который был передан как аргумент. Функция также поддерживает дополнительные параметры, которые позволяют управлять сортировкой.

SORTED ВСЕГДА ВОЗВРАЩАЕТ СПИСОК

```
In [1]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [2]: sorted(list_of_words)
Out[2]: ['', 'dict', 'list', 'one', 'two']

In [3]: tuple_of_words = ('one', 'two', 'list', '', 'dict')

In [4]: sorted(tuple_of_words)
Out[4]: ['', 'dict', 'list', 'one', 'two']

In [5]: set_of_words = {'one', 'two', 'list', '', 'dict'}

In [6]: sorted(set_of_words)
Out[6]: ['', 'dict', 'list', 'one', 'two']
```

SORTED ВСЕГДА ВОЗВРАЩАЕТ СПИСОК

```
In [7]: string_to_sort = 'long string'

In [8]: sorted(string_to_sort)
Out[8]: [' ', 'g', 'g', 'i', 'l', 'n', 'n', 'o', 'r', 's', 't']

In [9]: dict_for_sort = {
   ...:         'id': 1,
   ...:         'name':'London',
   ...:         'to_name': None,
   ...:         'to_id': None,
   ...:         'port':'G1/0/11'
   ...: }

In [10]: sorted(dict_for_sort)
Out[10]:
['id',
 'name',
 'port',
 'to_id',
 'to_name']
```

REVERSE

Флаг reverse позволяет управлять порядком сортировки. По умолчанию сортировка будет по возрастанию элементов.

```
In [11]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [12]: sorted(list_of_words)
Out[12]: ['', 'dict', 'list', 'one', 'two']

In [13]: sorted(list_of_words, reverse=True)
Out[13]: ['two', 'one', 'list', 'dict', '']
```

KEY

С помощью параметра key можно указывать, как именно выполнять сортировку. Параметр key ожидает функцию, с помощью которой должно быть выполнено сравнение.

Например, таким образом можно отсортировать список строк по длине строки:

```
In [14]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [15]: sorted(list_of_words, key=len)
Out[15]: ['', 'one', 'two', 'list', 'dict']
```

KEY

Если нужно отсортировать ключи словаря, но при этом игнорировать регистр строк:

```
In [16]: dict_for_sort = {
    ...:          'id': 1,
    ...:          'name':'London',
    ...:          'IT_VLAN':320,
    ...:          'User_VLAN':1010,
    ...:          'Mngmt_VLAN':99,
    ...:          'to_name': None,
    ...:          'to_id': None,
    ...:          'port':'G1/0/11'
    ...: }

In [17]: sorted(dict_for_sort, key=str.lower)
Out[17]:
['id',
 'IT_VLAN',
 'Mngmt_VLAN',
 'name',
 'port',
 'to_id',
 'to_name',
 'User_VLAN']
```

KEY

Параметру key можно передавать любые функции, не только встроенные. Также тут удобно использовать анонимную функцию lambda.

С помощью параметра key можно сортировать объекты не по первому элементу, а по любому другому. Но для этого надо использовать или функцию lambda, или специальные функции из модуля operator.

KEY

Например, чтобы отсортировать список кортежей из двух элементов по второму элементу, надо использовать такой прием:

```
In [18]: from operator import itemgetter

In [19]: list_of_tuples = [('IT_VLAN', 320),
    ...:  ('Mngmt_VLAN', 99),
    ...:  ('User_VLAN', 1010),
    ...:  ('DB_VLAN', 11)]

In [20]: sorted(list_of_tuples, key=itemgetter(1))
Out[20]: [('DB_VLAN', 11), ('Mngmt_VLAN', 99), ('IT_VLAN', 320), ('User_VLAN', 1010)]
```

ENUMERATE

ENUMERATE

Иногда, при переборе объектов в цикле for, нужно не только получить сам объект, но и его порядковый номер. Это можно сделать, создав дополнительную переменную, которая будет расти на единицу с каждым прохождением цикла. Однако, гораздо удобнее это делать с помощью итератора **`enumerate()`**.

```
In [15]: list1 = ['str1', 'str2', 'str3']

In [16]: for position, string in enumerate(list1):
    ...:     print(position, string)
    ...:
0 str1
1 str2
2 str3
```

ENUMERATE

`enumerate()` умеет считать не только с нуля, но и с любого значение, которое ему указали после объекта:

```
In [17]: list1 = ['str1', 'str2', 'str3']

In [18]: for position, string in enumerate(list1, 100):
    ...:     print(position, string)
    ...:
100 str1
101 str2
102 str3
```

ENUMERATE

Иногда нужно проверить, что сгенерировал итератор, как правило, на стадии написания скрипта. Если необходимо увидеть содержимое, которое сгенерирует итератор, полностью, можно воспользоваться функцией list:

```
In [19]: list1 = ['str1', 'str2', 'str3']

In [20]: list(enumerate(list1, 100))
Out[20]: [(100, 'str1'), (101, 'str2'), (102, 'str3')]
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ ENUMERATE ДЛЯ EEM

В этом примере используется Cisco EEM. Если в двух словах, то EEM позволяет выполнять какие-то действия (action) в ответ на событие (event).

Выглядит applet EEM так:

```
event manager applet Fa0/1_no_shut
 event syslog pattern "Line protocol on Interface FastEthernet0/0, changed state to down"
 action 1 cli command "enable"
 action 2 cli command "conf t"
 action 3 cli command "interface fa0/1"
 action 4 cli command "no sh"
```

В EEM, в ситуации, когда действий выполнить нужно много, неудобно каждый раз набирать `action x cli command`. Плюс, чаще всего, уже есть готовый кусок конфигурации, который должен выполнить EEM.

ПРИМЕР ИСПОЛЬЗОВАНИЯ ENUMERATE ДЛЯ EEM

С помощью простого скрипта Python можно сгенерировать команды EEM на основании существующего списка команд (файл enumerate_eem.py):

```
import sys

config = sys.argv[1]

with open(config, 'r') as f:
    for i, command in enumerate(f, 1):
        print('action {:04} cli command "{}"'.format(i, command.rstrip()))
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ ENUMERATE ДЛЯ EEM

Файл с командами выглядит так (r1_config.txt):

```
en
conf t
no int Gi0/0/0.300
no int Gi0/0/0.301
no int Gi0/0/0.302
int range gi0/0/0-2
 channel-group 1 mode active
interface Port-channel1.300
 encapsulation dot1Q 300
 vrf forwarding Management
 ip address 10.16.19.35 255.255.255.248
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ ENUMERATE ДЛЯ EEM

Вывод будет таким:

```
$ python enumerate_eem.py r1_config.txt
action 0001 cli command "en"
action 0002 cli command "conf t"
action 0003 cli command "no int Gi0/0/0.300"
action 0004 cli command "no int Gi0/0/0.301"
action 0005 cli command "no int Gi0/0/0.302"
action 0006 cli command "int range gi0/0/0-2"
action 0007 cli command " channel-group 1 mode active"
action 0008 cli command "interface Port-channel1.300"
action 0009 cli command " encapsulation dot1Q 300"
action 0010 cli command " vrf forwarding Management"
action 0011 cli command " ip address 10.16.19.35 255.255.255.248"
```

ФУНКЦИЯ ZIP

ФУНКЦИЯ ZIP

- на вход функции передаются последовательности
- zip() возвращает итератор с кортежами, в котором n-ый кортеж состоит из n-ых элементов последовательностей, которые были переданы как аргументы
  - например, десятый кортеж будет содержать десятый элемент каждой из переданных последовательностей
- если на вход были переданы последовательности разной длины, то все они будут отрезаны по самой короткой последовательности
- порядок элементов соблюдается

ФУНКЦИЯ ZIP

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

In [2]: b = [100,200,300]

In [3]: list(zip(a,b))
Out[3]: [(1, 100), (2, 200), (3, 300)]
```

ФУНКЦИЯ ZIP

Использование zip() со списками разной длины:

```
In [4]: a = [1,2,3,4,5]

In [5]: b = [10,20,30,40,50]

In [6]: c = [100,200,300]

In [7]: list(zip(a,b,c))
Out[7]: [(1, 10, 100), (2, 20, 200), (3, 30, 300)]
```

ИСПОЛЬЗОВАНИЕ ZIP ДЛЯ СОЗДАНИЯ СЛОВАРЯ

```
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]: list(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'}
```

ИСПОЛЬЗОВАНИЕ ZIP ДЛЯ СОЗДАНИЯ СЛОВАРЯ

Соберем их в словарь с ключами из списка и информацией из словаря data:

```
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',
```

ФУНКЦИЯ ALL

ФУНКЦИЯ ALL

Функция all() возвращает True, если все элементы истина (или объект пустой).

```
In [1]: all([False, True, True])
Out[1]: False

In [2]: all([True, True, True])
Out[2]: True

In [3]: all([])
Out[3]: True
```

ФУНКЦИЯ ALL

Например, с помощью all можно проверить, все ли октеты в IP-адресе являются числами:

```
In [4]: IP = '10.0.1.1'

In [5]: all( i.isdigit() for i in IP.split('.'))
Out[5]: True

In [6]: all( i.isdigit() for i in '10.1.1.a'.split('.'))
Out[6]: False
```

ФУНКЦИЯ ANY

ФУНКЦИЯ ANY

Функция any() возвращает True, если хотя бы один элемент истина.

```
In [7]: any([False, True, True])
Out[7]: True

In [8]: any([False, False, False])
Out[8]: False

In [9]: any([])
Out[9]: False

In [10]: any( i.isdigit() for i in '10.1.1.a'.split('.'))
Out[10]: True
```

ФУНКЦИЯ ANY

Например, с помощью any, можно заменить функцию ignore_command:

```
def ignore_command(command, ignore):
    ignore = ['duplex', 'alias', 'Current configuration']

    ignore_command = False

    for word in ignore:
        if word in command:
            return True
    return ignore_command
```

На такой вариант:

```
def ignore_command(command, ignore):
    ignore = ['duplex', 'alias', 'Current configuration']

    return any(word in command for word in ignore)
```