**Практична робота 5**

**ІТЕРАТОРИ ТА ГЕНЕРАТОРИ В PYTHON**

Генератори і ітератори представляють собою інструменти, які, як правило, використовуються для потокової обробки даних.

Ітеріруемий об'єкт (iterable) - це об'єкт, який для даного об'єкта видає наступний елемент, або видає виняток, якщо елементів більше немає. Це об'єкт, який повертає свої елементи по одному за раз. Крім того, ітератор запам'ятовує, на якому об'єкті він зупинився в останню ітерацію.

Основне місце використання ітераторів - це цикл for. Якщо перебирати елементи в деякому списку або символи в рядку за допомогою циклу for, то, фактично, це означає, що при кожній ітерації циклу відбувається звернення до ітератору, що міститься в рядку/списку, з вимогою видати наступний елемент. Якщо елементів в об'єкті більше немає, то ітератор генерує виняток, яке обробляється в рамках циклу for непомітно для користувача.

>>> num\_list = [ 1 , 2 , 3 , 4 , 5 ]

>>> for i in num\_list:

print (i)

1

2

3

4

5

В Python за отримання ітератора відповідає функція *iter()*. У кожного ітератора присутній метод *\_\_iter\_\_* - тобто, будь-який ітератор є ітеруємим об'єктом. Цей метод просто повертає сам итератор. Можна використовувати функцію *next(),* який викликає метод *\_\_next\_\_,* щоб взяти наступний елемент:

>>> itr = iter (num\_list)

>>> print (next(itr))

1

>>> print (next(itr))

2

>>> print (next(itr))

3

>>> print (next(itr))

4

>>> print (next(itr))

5

>>> print (next(itr))

Traceback (most recent call last):

File "<pyshell#10>", line 1, in <module>

print (next(itr))

StopIteration

Після того, як елементи закінчилися, повертається виняток StopIteration. Для того, щоб ітератор знову почав повертати елементи, його треба створити заново.

Приклади ітеріруемих об'єктів:

* всі послідовності: list, str, tuple;
* словники;
* файли.

Один із найпоширеніших прикладів ітератора - файл.

Файл config\_sw.txt

!

version 15.0

no service pad

service timestamps debug datetime msec

service timestamps log datetime msec

service password-encryption

!

hostname S3

!

boot-start-marker

boot-end-marker

!

enable secret 4 06YFDUHH61wAE/kLkDq9BGho1QM5EnRtoyr8cHAUg.2

!

no aaa new-model

system mtu routing 1500

Якщо відкрити файл звичайною функцією *open()*, отримаємо об'єкт, який представляє файл:

>>> f = open ('config\_sw.txt')

Цей об'єкт є ітератором, що можна перевірити, викликавши метод *\_\_next\_\_*:

>>> f.\_\_next\_\_()

'!\n'

>>> f.\_\_next\_\_()

'version 15.0\n'

Генератори - це спеціальний клас функцій, який дозволяє легко створювати свої ітератори. На відміну від звичайних функцій, генератор не просто повертає значення і завершує роботу, а повертає ітератор, який віддає елементи по одному.

Звичайна функція завершує роботу, якщо:

* ­­зустрівся вираз return;
* закінчився код функції (це спрацьовує як вираз return None);
* виникло виключення.

Після виконання функції управління повертається і програма виконується далі. Всі аргументи, які передавалися в функцію, локальні, тому їх значення втрачаються. Залишається тільки результат, який повернула функція.

Функція може повертати список елементів, кілька об'єктів або повертати різні результати в залежності від аргументів, але вона завжди повертає якийсь один результат.

Генератор ж генерує значення. При цьому значення повертаються за запитом і після повернення одного значення виконання функції-генератора припиняється до запиту наступного значення. Між запитами генератор зберігає свій стан. В генераторі замість ключового слова *return* використовується *yield*.

Python дозволяє створювати генератори двома способами:

* генераторное вираз;
* функція-генератор.

**Регулярні вирази**

Регулярні вирази - це послідовність з звичайних та спеціальних символів. Ця послідовність задає шаблон, який пізніше використовується для пошуку підрядків.

При роботі з мережним обладнанням регулярні вирази можуть використовуватися, наприклад, для:

* отримання інформації з виводів команд *show*;
* відбору частини рядків з виводу команди *show*, які збігаються з шаблоном;
* перевірки, чи є певні налаштування в конфігурації.

Наприклад:

* обробивши вивід команди *show version*, можна збирати інформацію про версію ОС та uptime обладнання;
* отримати з лог-файлу ті рядки, які відповідають шаблону;
* отримати з конфігурації ті інтерфейси, на яких немає опису (description).

Крім того, в самому мережному обладнанні регулярні вирази можна використовувати для фільтрації виводу будь-яких команд *show*.

В цілому, використання регулярних виразів буде пов'язане з отриманням частини тексту з великого виводу. Але це не єдине, в чому вони можуть пригодитися.

Наприклад, за допомогою регулярних виразів можна виконати заміни в рядках або розділення рядків на частини.

Ці області застосування перетинаються з методами, які застосовуються до рядків.

І, якщо завдання зрозуміло і просто вирішується за допомогою методів строк, краще їх використовувати. Такий код буде простіше зрозуміти і, крім цього, методи строк працюють швидше.

Але методи строк можуть справитися не з усіма завданнями і можуть сильно ускладнити вирішення завдання. У цьому випадку можуть допомогти регулярні вирази.

Але методи строк можуть справитися не з усіма завданнями і можуть сильно ускладнити вирішення завдання. У цьому випадку можуть допомогти регулярні вирази.

**Синтаксис регулярних виразів**

У Python для роботи з регулярними виразами використовується модуль *re*. Відповідно, для початку роботи з регулярними виразами треба його імпортувати.

В першій половині цього розділу для всіх прикладів буде використовуватися функція *search*. А в наступних підрозділах будуть розглянуті інші функції модуля *re*.

Синтаксис функції *search* такий:

match = re.search(regex, string)

де

regex - регулярний вираз

string - рядок, в якій шукається збіг

Якщо збіг було знайдено, функція поверне спеціальний об'єкт Match. Якщо ж збігу не було, функція поверне None.

При цьому особливість функції пошуку полягає в тому, що вона шукає тільки першйи збіг.

Якщо в рядку є кілька підрядків, які відповідають регулярному виразу, пошук поверне тільки перше знайдений збіг.

Наприклад, чи є підрядок 'MTU' в рядку int\_line.

>>> import re

>>> int\_line = ' MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,'

>>> zbig = re.search('MTU', int\_line)

>>> print(zbig)

<\_sre.SRE\_Match object; span=(1, 4), match='MTU'>

У об'єкта Match є кілька методів, які дозволяють отримувати різну інформацію про отриманий збіг. Наприклад, метод group() показує, що в рядку збіглося з описаним виразом.

В даному випадку це просто підрядок 'MTU':

>>> zbig.group()

'MTU'

Можливості регулярних виразів проявляються при використанні спеціальних символів. Наприклад, символ \d означає цифру, а + означає повторення попереднього символу один або більше разів. Якщо їх поєднати \d+, вийде вираз, що означає одну або більше цифр.

Використовуючи цей вислів, можна отримати частину рядка, в якій описана пропускна здатність:

>>> match\_bw = re.search('BW \d+', int\_line)

>>> match\_bw.group()

'BW 10000'

Особливо корисні регулярні вирази в отриманні певних підрядків з рядка. Наприклад, необхідно отримати VLAN, MAC і порти з виводу такого лог-повідомлення:

>>> log2 = 'Oct 3 12:49:15.941: %SW\_MATM-4-MACFLAP\_NOTIF: Host f04d.a206.7fd6 in vlan 1 is flapping between port Gi0/5 and port Gi0/16'

>>> re.search('Host (\S+) in vlan (\d+) is flapping between port (\S+) and port (\S+)', log2).groups()

('f04d.a206.7fd6', '1', 'Gi0/5', 'Gi0/16')

Метод groups() повертає тільки ті частини початкового рядка, які потрапили в круглі дужки. Таким чином, уклавши частину виразу в дужки, можна вибрати фрагменти рядки, які треба запам'ятати.

Вираз \d+ вже використовувалося раніше - воно описує одну або більше цифр. А вираз \S+ описує всі символи, крім whitespace (пробіл, таб і інші).

Інші спеціальні символи представлені в таблицях.

|  |  |
| --- | --- |
| \d | будь-яка цифра |
| \D | будь-яке нечислове значення |
| \s | whitespace (\t\n\r\f\v) |
| \S | все, окрім whitespace |
| \w | будь-яка буква, цифра або нижнє підкреслення |
| \W | все, крім букв, цифр або нижнього підкреслення |

Символи повторення

|  |  |
| --- | --- |
| + | одне або більше повторення попереднього елемента |
| \* | нуль або більше повторення попереднього елемента |
| ? | нуль або одне повторення попереднього елемента |
| {n} | рівно n повторення попереднього елемента |
| {n, m} | від n до m повторення попереднього елемента |
| {n,} | n або більше повторення попереднього елемента |

Спеціальні символи

|  |  |
| --- | --- |
| . | будь-який символ, крім символу нового рядка |
| ^ | початок рядка |
| $ | кінець рядка |
| [abc] | будь-який символ в дужках |
| [^abc] | будь-який символ, крім тих, що в дужках |
| a|b | елемент a або b |
| (regex) | вираз розглядається як один елемент ті подрядок, який збігся з виразом, запам'ятовується |

**Завдання**

1. За допомогою регулярних виразів отримати з рядку

Jun 3 14:39:05.941: %SW\_MATM-4-MACFLAP\_NOTIF: Host f03a.baaa.7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15

­­− час;

− МАС-адрес;

− в МАС-адресі вивести частину, в якій є символ b, а потім нуль або більше повторень a;

2. З файлу с записами email зберігти тільки ті, які:

− в лівій частині адреси мають три слова, розділені крапкою, а в правій частині адрес nmu.one. Наприклад: [test.user.v@nmu.one](mailto:test.user.v@nmu.one).

− в лівій частині адреси мають нуль або одну крапку.

4. В файлі mac\_adr\_table.txt збережено вивід на комутаторі команди show mac-address-table. Отримати:

− значення всіх MAC-адрес з файлу;

for line in mac\_table.split('\n'):

match = re.search('\w{4}\.\w{4}\.\w{4}', line)

if match:

print('MAC: ', match.group())

− значення всіх VLAN з файлу;

for line in mac\_table.split('\n'):

match = re.search('\d{1,4}', line)

if match:

print('VLAN: ', match.group())

− всі порти FastEthernet;

− MAC-адреси, які знаходяться в діапазоні