Advanced Classes

Ітератор, генератор, корутина, метаклас

Ітератор

- Інтератор представляє собою об'єкт перечислювач, який для даного об'єкта видає наступний елемент, або кидає ексепшин StopIteration, якщо елементів більше немає

- Основне місце використання ітераторів - це цикл for. Якщо ви перебираєте елементи в якомусь списку або символи в строках за допомогою цикла for, то, фактично, це означає, що при кожній інтерації цикла відбувається звернення до ітератора, що міститься у строках / списках, з вимогою видати наступний елемент, якщо елементів в об'єкті більше немає, то ітератор генерує виключення, що опрацьовується в рамках цикла for незамітно для користувача.

Цикл for використовує ітератор

```
num_list = [1, 2, 3, 4, 5]
for i in num_list:
    print(i)
```

Цикл for "під капотом"

- В num_list береться ітератор і на кожній ітерації робимо next(iterator)

```
iterator = iter(num_list)
a = next(iterator)
```

Реалізація ітератора

- З точки зору реалізації в Python, ітератор - це екземпляр класу який має 2 магічних методи - __iter__ (який повертає об'єктітератор), __next__ (який повертає наступний елемент ітератора)

```
class SimpleIterator:
    def __iter__(self):
        return self

    def __init__(self, limit):
        self.limit = limit
        self.counter = 0

    def __next__(self):
        if self.counter < self.limit:
            self.counter += 1
            return 1
        else:
            raise StopIteration
```

Генератори

- Генератори - це функції, які можна припиняти і відновлювати під час їх виконання, при цьому вони повертають об'єкт, який можна ітерувати. На відміну від списків, вони ліниві і тому працюють з поточним елементом тільки за запитом. Таким чином, вони набагато ефективніше використовують пам'ять при роботі з великими наборами даних.

Функція генератор

- Щоб створити генератор, необхідно визначити функцію, як зазвичай, але використовувати yield замість return, вказуючи інтерпретатору, що цю функцію слід розглядати як генератор:

```
def countdown(num):
    print('Starting')
    while num > 0:
        yield num
        num -= 1
```

Як працює?

- Виклик функції не виконує її. Функція повертає об'єкт-генератор, який використовується для управління виконанням.
- Об'єкти генератора виконуються при виклику next (): val = countdown(5) next(val)
- Кожен раз при виклику next() функція відпрацьовує до yield і "замерзає". При наступному виклику next() вона "розмерзається" і продовжує роботу до наступного yield
- Функцію next() можна визвати стільки ж разів скільки є yield в функції генераторі. При наступних викликах буде кидатись ексепшин StopIteration

Вираз-генератор

- Як і списки, генератори також можуть бути написані таким же чином, за винятком того, що вони повертають об'єкт генератора, а не список:

```
my_list = ['a', 'b', 'c', 'd']
gen_obj = (x for x in my_list)
for val in gen_obj:
    print(val)
```

Yield from

- Дана конструкція поєднує в собі yield і цикл for. Наприклад

```
def numbers_range(n):
    for i in range(n):
        yield i

def numbers_range(n):
    yield from range(n)
```

Також yield from використовується для субгенераторів

```
def subgenerator():
    yield 'World'
def generator():
    yield 'Hello'
    yield from subgenerator()
    yield '!'
for i in generator():
    print(i, end = ' ')
```

Де застосовуються

- Генератори ідеально підходять для читання великої кількості великих файлів, оскільки вони видають дані по одному фрагменту за раз, незалежно від розміру вхідного потоку. Вони також можуть привести до більш чистого коду шляхом поділу процесу ітерації на більш дрібні компоненти.

Корутини

- Корутини - це генератори, які не тільки віддають, а й приймають дані. Будь-яка корутина це генератор, але не кожен генератор це корутина!

```
def courutine():
    while True:
        a = yield
        print(a)
```

Щоб передати значення в корутину потрібно
cour = courutine()

next(cour) - дана команда нічого не зробить, проте є обов'язковою. Часто її виносять в декоратор для корутин

cour.send(12) - таким чином в корутину передали число 12

Для чого використовується?

- На генераторах і корутинах побудований асинхронний підхід в Python

Також в корутину можна прокинути ексепшин

- cour.throw(StopIteration)

Метакласи

- Метаклас - це клас об'єктами якого є інші класи
- В Python будь-що є екземпляром певного класу, це стосується навіть вбудованих структур чи типів. Наприклад

```
>>> type(1)
<class 'int'>
```

- Тобто 1 - екземпляром класу int

- Ми можем створити свій тип, оголосивши певний клас.
- Перевірим тип даних екземпляру власного класу

```
>>> obj = MyClass()
>>> type(obj)
<class '__main__.MyClass'>
```

Функція також екземпляр класу?

```
>>> def func():
        pass
>>>type(func)
<class 'function'>
```

А клас?

```
>>> class MyClass:
        pass
>>> type(MyClass)
<class 'type'>
```

Що ж таке 'type'?

- type окрім того що повертає тип даних певного об'єкта, може виступати ще конструктором класів. Тобто метакласом
- Тобто за допомогою type ми можемо створити клас

```
>>> User = type("User", (), {"name": "Misha"})
>>> User.name
'Misha'
```

- Першим аргументом передається ім'я типу, другим клас від якого наслідуємося, третім словник атрибутів і їх значення.

- Якщо клас - це шаблон для об'єктів, то метаклас це шаблон для класів.
- Клас створює об'єкти, метаклас створює класи

Метакласом не обов'язково повинен бути клас

```
def upper_attr(future_class_name, future_class_parents, future_class_attr):
  uppercase_attr = {}
  for name, value in future_class_attr.items():
    if name.startswith('__'):
      continue
    uppercase_attr[name.upper()] = value
  return type(future_class_name, future_class_parents, uppercase_attr)


class Foo(metaclass=upper_attr):
  bar = 'bip'


obj = Foo()
```

```
class UpperAttrMetaclass(type):
  def __new__(upperattr_metaclass, future_class_name,
              future_class_parents, future_class_attr):
    uppercase_attr = {}
    for name, value in future_class_attr.items():
      if name.startswith('__'):
        continue
      uppercase_attr[name.upper()] = value
    return type(future_class_name, future_class_parents, uppercase_attr)

class Foo(metaclass=UpperAttrMetaclass):
  bar = 'bip'
```