Декоратори

Функції як процедури

- Процедура - це іменована послідовність обчислювальних кроків. Будь-яку процедуру можна викликати в будь-якому місці програми, в тому числі всередині іншої процедури або навіть самої себе.

Функції як об'єкти першого класу

- В Python все є об'єктом, а не тільки об'єкти, які ви створюєте з класів. У цьому сенсі він (Python) повністю відповідає ідеям об'єктно-орієнтованого програмування.
- Це означає, що в Python все це - об'єкти:
- числа;
- строки;
- класи;
- функції (те, що нас цікавить).

Що нам це дає?

- Якраз через те що функції це об'єкти ми можемо:

1. Присвоювати їх як змінні
2. Передавати в якості аргументів в інші функції
3. Повертати з інших функцій
4. Визначити одну функцію всередині іншої

Функції вищих порядків

- Функції вищих порядків - це такі функції, які можуть приймати в якості аргументів і повертати інші функції.

Декоратори

- Декоратор - це функція, яка дозволяє обернути іншу функцію для розширення її функціональності без безпосереднього зміни її коду.

Як працюють декоратори?

- Приклад декоратора

```
def decorator_function(func):
    def wrapper():
        print(Викликаємо обгорнуту функцію...')
        func()
        print('Виходимо з обгортки')
    return wrapper
```

- Дана функція є функцією вищого порядку
- Усередині decorator_function () ми визначили іншу функцію, яка обгортала функцію-аргумент і потім змінювала її поведінку. Декоратор повертає цю обгортку

Тепер подивимося на декоратор в дії

```
@decorator_function
def hello_world():
    print('Hello world!')
```

- Просто додавши @decorator_function перед визначенням функції hello_world (), ми модифікували її поведінку.
- Слід розуміти що, вираз з @ є всього лише синтаксичним цукром для hello_world = decorator_function (hello_world).

Декоратор з аргументами

- Щоб добавити можливість передавати аргументи в декоратор, потрібно зробити 3 "шар" обгортки який прийме аргументи
- І так матимемо замикання 3 вкладених функцій, де 1 шар - прийме аргументи декоратора і створить декоратор, другий шар прийме функцію яку декоруємо, і третій шар прийме аргументи функції що декорується

functools.wraps

- Розглянемо такий код

```
def decor(func):
    def wrap():
        func()
    return wrap

@decor
def func():
    pass

print(func.__name__)
```

- Прінт дасть нам wrap а не func, і всі звернення до func будуть насправді зверненням до wrap і це логічно. Проте це може зломати нам деяку логіку коду.
- Щоб виправити це можна скористатись "костилем":
- ```
  def decor(func):
      def wrap():
          func()

      wrap.__name__ = func.__name__
      return wrap
  ```
  ```

- Проте якщо ми хочемо перевизначити більше атрибутів, даний варіант нам не підходить

- Для перевизначення всіх атрибутів можна скористатись декоратором wraps який задекорує внутрішній метод

```
from functools import wraps

def decor(func):
    @wraps(func)
    def wrap():
        func()

    return wrap
```

Клас-декоратор

- Використовуючи магічний метод __call__ який робить так щоб екземпляр класу вів себе як функція, можна добитись того що декоратором стане екземпляр класу. А отже можна буде використати всі переваги класів, у створені новго декоратора.

Приклад

```
class Repeater:
    def __init__(self, n):
        self.n = n

    def __call__(self, f):
        def wrapper(*args, **kwargs):
            for _ in range(self.n):
                f(*args, **kwargs)
        return wrapper

@Repeater(3)
def foo():
    print('foo')
```

Декоратор класу

```
def decor_all_methods(cls):
    class NewCls:
        def __init__(self, *args, **kwargs):
            self._obj = cls(*args, **kwargs)

        def __getattribute__(self, s):
            try:
                x = super().__getattribute__(s)
            except AttributeError:
                pass
            else:
                return x
            attr = self._obj.__getattribute__(s)
            if isinstance(attr, type(self.__init__)):
                return decor(attr)
            else:
                return attr
    return NewCls
```