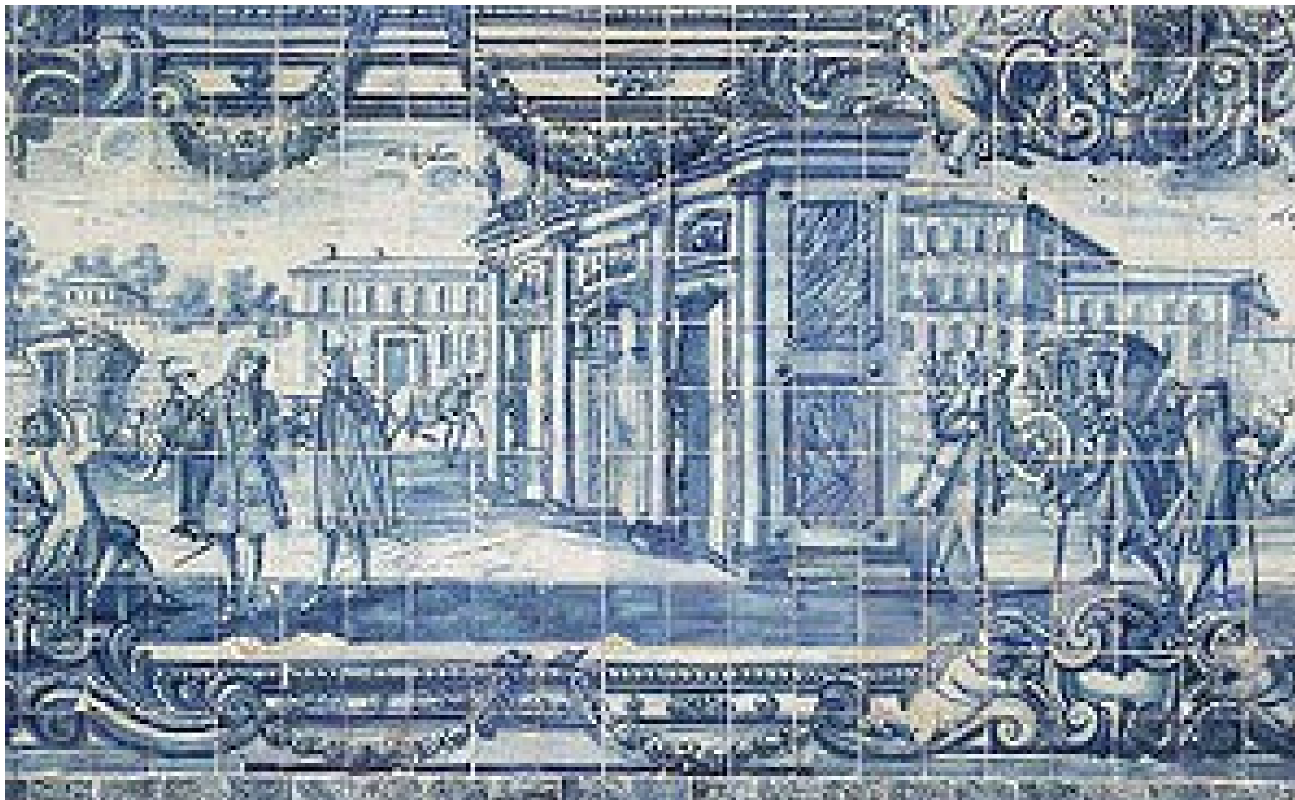
ІНФОРМАТИКА ТА ПРОГРАМУВАННЯ

Тема 17. Декоратори

Декоратори

- **Декоратори** – це функції, які модифікують поведінку інших функцій, не змінюючи їх.



Декоратори.2

- По суті, декоратор реалізує композицію функцій.
- Якщо g – це декоратор, а $f(x)$ – функція, то застосування декоратора g до функції f означає виклик $g(f(x))$.
- Сенс застосування декораторів полягає у тому, що вони можуть додавати однакову поведінку різним функціям.
- При цьому, декоратори та функції, які вони модифікують, можуть створюватись незалежно та у різний час.

Синтаксис та виконання декораторів

- У Python декоратори мають такий синтаксис:

@decorator

  - де decorator – ім’я декоратора.

- Для застосування декоратора до функції необхідно вказати його безпосередньо перед описом функції:

```
@decorator
def f(x):
    P
```

  - де decorator – ім’я декоратора, *f* – ім’я функції, *P* – інструкція.

- Правило виконання застосування декоратора.

Коли Python зустрічає позначення декоратора перед описом функції *f*, він модифікує програмний код так, що додає після опису функції виклик декоратора, а функція *f* передається йому як параметр. Результат виклику присвоюється змінній f.

Стандартні декоратори

- Python містить декілька стандартних декораторів.
- Серед них виділимо два: @staticmethod та @property. Декоратор @staticmethod використовують для вказання статичних методів класу.
- Застосування декоратора @staticmethod рівносильно такому опису:

```
class A:
    ...
    @staticmethod
    def f():
        P
```

≡

```
class A:
    ...
    def f():
        P

    f = staticmethod(f)
```

Стандартні декоратори.2

- Декоратор @property використовують для вказання властивостей класу.
- У Python поля класу відрізняються від властивостей класу тим, що властивості можна визначити як такі, що дозволяється тільки читати, але не змінювати.
- Або для властивості можна вказати певні дії, що треба виконати під час зміни значення властивості.
- Декоратор @property, як і інші декоратори, вказують перед описом функції.
- У даному випадку, - перед описом метода, ім’я якого і буде іменем властивості.
- Наприклад, якщо у класі A вказати перед описом метода h() декоратор @property, то h буде властивістю, до якої можна звертатись як до поля класу.

Стандартні декоратори.3

```
class A:
    ...
    @property
    def h():
        P
...
a = A()
x = a.h
```

Приклад

- Скласти модуль для зображення та переміщення точок та кіл по екрану з використанням декораторів @staticmethod та @property.
- Цей приклад було розглянуто у темі «Класи та об’єкти».
- Тож застосування декораторів трохи змінить текст.
- Зокрема, замість довгих імен функцій getx, gety визначимо властивості x, y, значення яких можна тільки читати.
- Також визначимо статичний метод printcount за допомогою відповідного декоратора.

Реалізація власних декораторів

- У Python, окрім використання стандартних декораторів, можна також описувати власні декоратори.
- Будь-який декоратор – це функція, яка в якості параметру має іншу функцію і повертає підфункцію, що виконує додаткову роботу, передбачену декоратором.
- Загальний шаблон декоратора має такий вигляд:

Реалізація власних декораторів.2

```
def mydecorator(function):
    def _mydecorator(*args, **kw):
        # виконати дії перед викликом реальної функції
        res = function(*args, **kw)
        # виконати дії після виклику функції
        return res
    # повренути підфункцію
    return _mydecorator
```

Реалізація власних декораторів.3

- Тоді вказання декоратора @mydecorator рівносильно такій послідовності команд:

```
@mydecorator     ≡     def f(x):
def f(x):                  P
    P
                       f = mydecorator(f)
```

- Тому, подальші виклики функції f по суті будуть означати виклики функції _mydecorator з тими ж параметрами, які були вказані для f.
- Зверніть увагу, що у шаблоні декоратора функція приймає довільну кількість позиційних та ключових параметрів, що дозволяє застосовувати цей декоратор для будь-якої функції.

Приклад. Реалізація декоратора @benchmark

- Реалізувати декоратор, що вимірює час виконання функції та застосувати його для перевірки часу обчислення числа Фібоначчі з використанням нерекурсивного, рекурсивного варіантів, а також генератора-функції.
- Реалізуємо генератор @benchmark, який використовує описаний вище загальний шаблон, у окремому модулі.
- Для вимірювання часу виконання функції використаємо стандартну функцію time.perf_counter() з модуля time, яка повертає поточний час.

Приклад. Реалізація декоратора @benchmark.2

```
def benchmark(f):
    '''Декоратор @benchmark для обчислення часу виконання функції
f.
     '''
    import time
    def _benchmark(*args, **kw): #функція _benchmark містить код, що
                #виконується перед та після виклику f
        t = time.perf_counter()   #вимірюємо час перед викликом функції
        rez = f(*args, **kw)      #викликаємо f
        t = time.perf_counter() - t       #вимірюємо різницю у часі
        print('{0} time elapsed {1:.8f}'.format(f.__name__, t))
        return rez
    return _benchmark
```

Приклад. Реалізація декоратора @benchmark.3

- Після виконання цього прикладу (версія 1) бачимо, що все працює правильно окрім визначення імені функції, оскільки декоратор замість імені декорованої функції підставляє своє ім'я.
- Зарадити цьому можна різними способами, які реалізовані у версіях 2 та 3 даного прикладу.

Реалізація власних декораторів з параметрами

- У деяких випадках виконання декоратора залежить від параметрів.
- Тоді шаблон декоратора дещо змінюється: з’являється ще одна зовнішня функція, яка повертає декоратор, який, в свою чергу, повертає підфункцію.

Реалізація власних декораторів з параметрами.2

```
def mydecorator(arg1, arg2):
    def _mydecorator(function):
        def __mydecorator(*args, **kw):
   # виконати дії перед викликом реальної функції
            res = function(*args, **kw)
            # виконати дії після виклику функції
            return res
        # повренути підфункцію
        return __mydecorator
    return _mydecorator
```

Реалізація власних декораторів з параметрами.3

- Тут arg1, arg2 – параметри декоратора, які можуть використовуватись у всіх вкладених функціях.
- Застосування декоратора з параметрами виглядає так:

```
@mydecorator(e1, e2)
def f(x):
    P
```

- Треба відмітити, що значення фактичних параметрів e1, e2 декоратора mydecorator повинні бути визначені до моменту його застосування до функції f, тобто до місця опису (а не виклику) функції.
- Потім у всіх викликах функції f використовують ті ж самі значення параметрів e1, e2.

Вкладені декоратори

- До однієї функції можуть бути застосовані декілька декораторів.
- Тоді всі ці декоратори вказують перед описом функції.
- Вказані декоратори застосовуються до функції послідовно, починаючи з найближчого до опису функції декоратора.

```
@mydecorator1      ≡     def f(x):
@mydecorator2                P
def f(x):
    P                    f = mydecorator1(mydecorator2(f))
```

Приклад. Реалізація декоратора @trace

- Реалізувати декоратор, який виводить повідомлення про вхід та вихід з кожної функції. Виведення повідомлень повинно здійснюватись тільки в режимі налагодження (debug).
- Виведення повідомлень про вхід та вихід з кожної функції ще називають трасуванням.
- Тому відповідний декоратор називається @trace.
- Цей декоратор реалізовано у окремому модулі.
- Програма, що перевіряє застосування @trace, - та ж сама, що і для декоратора @benchmark.
- У ній до функцій обчислення чисел Фібоначчі застосовуються обидва декоратори.
- Застосування @trace залежить від параметра debug.

Приклад. Реалізація декоратора @trace.2

```
def trace(debug = True):
  '''Декоратор @trace для відсліковування виконання функції f.
  Параметр debug вказує, чи здійснювати трасування    '''
  def _trace(f):
    @functools.wraps(f)   #декоратор оновлює значення атрибутів
                          #__trace відповідними атрибутами f
    def __trace(*args, **kw):
      if debug:
        print('вхід до',f.__name__)
      rez = f(*args, **kw)    #викликаємо f
      if debug:
        print('вихід з',f.__name__)
      return rez
    return __trace
  return _trace
```

Резюме

- Ми розглянули:
  1. Визначення декоратора
  2. Застосування стандартних декораторів @staticmethod, @property
  3. Написання власних декораторів
  4. Написання власних декораторів з параметрами
  5. Вкладені декоратори

Де прочитати

1. Обвінцев О.В. Об’єктно-орієнтоване програмування. Курс на основі Python. Матеріали лекцій. – К., Основа, 2017
2. Марк Лутц, Изучаем Python, 4-е издание, 2010, Символ-Плюс
3. Python 3.4.3 documentation
4. Марк Саммерфилд, Программирование на Python 3. Подробное руководство. - Символ-Плюс, 2009.
5. Tarek Ziadé. Expert Python Programming. - Packt Publishing, 2008.
6. David Beazley and Brian K. Jones, Python Cookbook. - O’Reilly Media, 2013.
7. https://wiki.python.org/moin/PythonDecoratorLibrary
8. http://habrahabr.ru/post/141501/
9. http://www.linuxtopia.org/online_books/programming_books/python_programming/python_ch26s05.html
10. http://thecodeship.com/patterns/guide-to-python-function-decorators/
11. http://www.ibm.com/developerworks/library/l-cpdecor/