ПЕРЕЛІК ПОСИЛАНЬ

1. Hyndman, R.J., Athanasopoulos, G. (2018). Chapter 8 ARIMA models. In R.J. Hyndman (Ed.), Forecasting: Principles and Practice (2nd edition). Melbourne, Australia: OTexts. Retrieved from https://otexts.com/fpp2/arima.html.
2. Robert Nau (n.d.). Introduction to ARIMA: nonseasonal models. Fuqua School of Business Duke University. Retrieved February 3, 2021, from https://people.duke.edu/~rnau/411arim.htm
3. Robert Nau (n.d.). Seasonal differencing in ARIMA models. Fuqua School of Business Duke University. Retrieved February 3, 2021, from https://people.duke.edu/~rnau/411sdif.htm.
4. Dag O., Yozgatligil C. GMDH: An R Package for Short Term Forecasting via GMDH-Type Neural Network Algorithms. The R Journal, 8(1), 379-386.
5. Kondo, T., Ueno, J. (2006). Revised GMDH-Type Neural Network Algorithm With A Feedback Loop Identifying Sigmoid Function Neural Network. International Journal of Innovative Computing, Information and Control, 2(5), 985-996.

УДК 004.832;681.324

А.Л. Ширін[1], Н.С. Дрешпак[1], А.Т. Харь[1], М.С. Міщенко[1]
[1]Національний технічний університет «Дніпровська політехніка», Дніпро, Україна

ВИКОРИСТАННЯ ДЕКОРАТОРІВ У МОВІ PYTHON 3

Анотація. Описано можливості використання декораторів у мові Python 3. Розглянуто приклади створення, виклику та синтаксису декораторів.

Ключові слова: *декоратор, Python, патерн, структурний шаблон проектування.*

Вступ. У сучасному програмуванні присутня потреба в чітко організованому коді. Такі потреби існують не тільки для того, щоб написаний код легше сприймався іншими програмістами, що працюють з ним, але і для відсутності потреби переписувати програму в майбутньому. Дійсно, якщо дотримуватися всіх рекомендацій з проектування коду, програмне забезпечення не потребуватиме серйозного редагування при потребі оновити його. Необхідний функціонал досить буде просто дописати і увімкнути в роботу. Один з кроків до створення продукту з такими можливостями – використання декораторів.

Основний зміст роботи. Розглянемо базові можливості мови Python. Функції в Python - це об'єкти [1]. Перевірити це можна за допомогою функції type() (рис. 1).

```
# визначення функції з назвою test
def test():
print("test") # вивід на екран тексту «test»
print(type(test))              # вивід на екран тип функції test
```

Рис. 1. Приклад використання функції type()

Програма виведе на екран <class 'function'>, що і підтверджує, що функції є об'єктами класу function. Це означає, що функції можна передавати як аргумент інших функцій і природно їх можна повертати як результат виконання (рис. 2).

Ще одна особливість мови Python 3 - визначення локальної функції всередині іншої функції [1]. Це означає, що код з оголошенням функції в функції буде успішно виконуватися (рис. 3).

Що таке декоратор і для чого він потрібен. Повертаючись до правильного проектування програмного коду, програму варто дописувати, а не переписувати, що вимагає в кілька разів менше зусиль. Так як продукт можна тільки дописувати, то змінювати якийсь об'єкт ми не маємо права. Отже, необхідний механізм, що дозволяє змінити поведінку об'єкта, не змінюючи цей самий об'єкт. І це дійсно можливо за допомогою декоратора [2].

```
# визначення функції з назвою test
def test():
print("test")          # вивід на екран тексту «test»
# визначення функції з назвою wrapper, яка приймає функцію як
# аргумент і так само повертає функцію як результат виконання
def wrapper(someFunc):
return someFunc   # повернення переданої функції як результат
newFunc = wrapper(test)# привласнення нової змінної результату
# функції
newFunc()  # непрямий виклик функції test шляхом
# виклику функції newFunc
```

Рис. 2. Приклад передачі функції в іншу і повернення як результат

```
def test():                    # визначення функції з назвою test
def testIntoTest():# визначення локальної функції всередині
# іншої функції з назвою testIntoTest
print("test")# вивід на екран тексту «test»
```

Рис. 3. Визначення функції всередині іншої функції

Виходить, декоратор - це викликаємий об'єкт, що дозволяє обернути інший об'єкт для розширення його функціональності без зміни коду самого

об'єкта. Патерн «Декоратор» реалізований за допомогою композиції класів такого компонента і самого декоратора [3]. Сам компонент успадковується від абстрактного. Те ж саме відбувається з класом декоратора. (рис. 4).

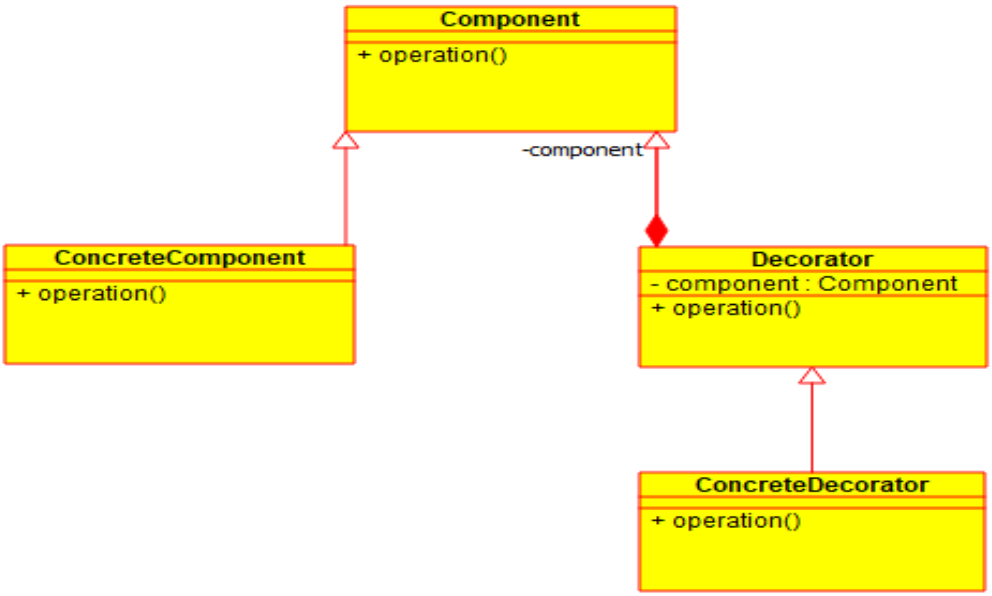

Рис. 4. UML-діаграма класів патерну «Декоратор»

Декоратор може виконувати безліч функцій, наприклад:

- виконання коду до виклику функції (зміна параметрів, типу параметрів);
- виконання коду після виклику функції (перевірка і обробка результатів);
- повторення виклику у разі помилки.

Знаючи що з себе представляє декоратор, розглянемо нестандартне його використання на прикладі функції (рис. 5).

```
# визначення функції-декоратора з функцією для декорування в параметрі
def decorator(someFunc):
# всередині декоратора визначається функція, яка отримує можливість
# виконувати код до і після виклику переданої функції
def wrapper():
print("before")      # код до виклику переданої функції
someFunc()           # виклик самої функції
print("after")       # код після виклику переданої функції
# повернення новоствореної функції всередині декоратора
return wrapper
```

Рис. 5. Приклад визначення декоратора-функції

Декоратор в вигляді функції створено. Виклик декоратора можна зробити трьома способами [4]:

- виклик декоратора безпосередньо в одному рядку (рис. 6);
- виклик декоратора через присвоювання результуючої функції в іншу функцію (рис. 7);
- виклик декоратора шляхом перевизначення початкової функції (рис. 8).

```
# виклик декоратора в одному рядку, викликаючи дві функції
decorator(someFunc)()
```

Рис. 6. Приклад виклику декоратора напряму

```
# привласнення результату виконання декоратора нової функції
someFuncDecorate = decorator(someFunc)
# виклик нової функції, в яку присвоєно результат виклику декоратора
someFuncDecorate()
```

Рис.7. Приклад виклику декоратора через привласнення

```
# привласнення результату виконання декоратора початкової функції
someFunc = decorator(someFunc)
# виклик початкової функції з новим функціоналом від декоратора
someFunc()
```

Рис. 8. Приклад виклику декоратора через перевизначення

Синтаксис декораторів в Python. В багатьох мовах програмування поняття декоратора існує тільки в області застосування патернів. Проектування і реалізація їх відбувається базовими можливостями мови, але в Python присутній синтаксис декораторів [5] для того, щоб спростити їх використання. Це є четвертий спосіб виклику декоратора (рис. 9).

```
    # те ж саме, що decorator(someFunc)
@decorator
    # визначення функції, яка згодом буде декорована
    def someFunc():
    # виклик початкової функції з новим функціоналом від декоратора
    print("something")        # зміст самої функції
    …
    someFunc()                # виклик декоратора
```

Рис. 9. Приклад виклику декоратора за допомогою синтаксису Python

Висновки. Декоратори використовуються для розширення можливостей функцій з сторонніх бібліотек, тобто функцій, код яких ми не можемо

змінювати. Так само вони створені для спрощення налагодження, коли зміна коду є небажаною. Також корисно використовувати декоратори для розширення різних функцій одним і тим же кодом, без повторного його переписування кожен раз.

ПЕРЕЛІК ПОСИЛАНЬ

1. Мова програмування Python 3, документація. Рівень доступу: https://docs.python.org/3/

2. Публікація T. Popovic: «Advanced Python Techniques: Decorators», 2015. Рівень доступу: https://www.researchgate.net/publication/272833859_Advanced_Python_Techniques_Decorators

3. Реалізація шаблону «Декоратор» у Python, Implementing the Decorator Pattern in Python, 2020. Рівень доступу: https://dev.to/erikwhiting88/implementing-the-decorator-pattern-in-python-1fdm

4. M. Harrison, Guide To: Learning Python Decorators. - CreateSpace 2013. - 59 с. - ISBN 9781492325611.

5. E. Matthes, Python Crash Course: A hands-on, project-based introduction to programming. - 2015. - 446-453с. - ISBN 978-1-59327-603-4.

УДК 004.415.3:681.6

С.М. Мацюк[1], Я.І. Журавльов[1]

[1]Національний технічний університет «Дніпровська політехніка», Дніпро, Україна

ГЕНЕРАЦІЯ МОДЕЛІ ЛАНДШАФТУ НА ОСНОВІ РЕГУЛЯРНОЇ СІТКИ ВИСОТ

Анотація. Проведено аналіз основних принципів представлення даних для зберігання інформації про ландшафти. Досліджено різні методи генерації тривимірних ландшафтів. Описано алгоритм для генерації моделі тривимірного ландшафту на основі регулярної сітки висот.

Ключові слова*: ландшафт, тріангуляція, карта висот, регулярна сітка висот, побудова поверхонь, тривимірна модель.*

Вступ. В даний час 3D-моделювання є найважливішою областю машинної графіки, оскільки побудова тривимірного зображення, близького до реального, є досить складним завданням. Але, завдяки великому колу споживачів і неймовірно швидкому зростанню продуктивності обчислювальних систем, ця область активно розвивається.

Модель 3D поверхні є цифровим відображенням просторових об'єктів, як реальних, так і гіпотетичних, в тривимірному просторі. Простими прикладами 3D поверхонь є ландшафти, міські вулиці, підземні газові сховища або мережа