ПРОГРАМУВАННЯ (PYTHON)

ДЕКОРАТОРИ

Декоратори

Функції як об’єкти можуть бути аргументами і результатами виконання інших функцій. Ця можливість покладена в основу функціонування декораторів. Декоратор дає змогу змінювати поведінку функції[85], змінюючи або не змінюючи саму функцію (іншими словами, декоратор підміняє одну функцією іншою).

У інших мовах програмування декорування – паттерн проектування, у Python – це елемент синтаксису.

---

85 Декоратор можна застосовувати не тільки до функції, а до будь-якого об’єкта, який можна викликати (методу, класу).

ДЕКОРУВАННЯ ФУНКЦІЙ

Інтерпретатор Python розпізнає декоратор за символом @. Синтаксис декорування функції виглядає так:

```
@ім'я_декоратора
def ім'я_функції():
    тіло функції
```

Цей запис аналогічний наступному:

```
def ім'я_функції():
    тіло функції

def ім'я_декоратора(функція):
    тіло декоратора

ім'я_функції = ім'я_декоратора(ім'я_функції)
```

Тобто, декоратор — це функція, аргументом якої є інша функція. Результатом виконання декоратора теж має бути функція.

Перший варіант запису, попри свою лаконічність, не завжди кращий. Оскільки функція декорується безпосередньо під час оголошення, то доступ до оригіналу (недекорованої функції) ускладнюється. Тому інколи доцільніше використовувати другий варіант декорування.

З попередніх фрагментів коду не зрозуміло, що ж міститься всередині функції-декоратора. Зазвичай там оголошується нова функція-обгортка, яка не модифікує невідворотньо оригінальну функцію, а тільки змінює її поведінку у випадку декорування. Продемонструємо це на конкретному прикладі

```
def money(deposit, years, pc):
    amount = deposit * (1 + pc/100)**years
    return round(amount)

print(money(1000, 2, 10))       # 1210
```

```python
# декоратор
def uah(func):
    def wrapper(*args, **kwargs):
        return f"{func(*args, **kwargs)} грн."
    return wrapper
```

В обгортці `wrapper` використано параметри `*args` і `**kwargs`, оскільки декоратору нічого невідомо про аргументи вхідної функції, а він зазвичай повинен декорувати різні функції з різною кількістю аргументів.

```python
# декорування (синтаксичний цукор)

@uah
def money(deposit, years, pc):
    amount = deposit * (1 + pc/100)**years
    return round(amount)

print(money(1000, 2, 10))       # 1210 грн.
```

```python
# декорування (розгорнута версія)

def money(deposit, years, pc):
    amount = deposit * (1 + pc/100)**years
    return round(amount)

money = uah(money)

print(money(1000, 2, 10))       # 1210 грн.
```

```python
# декорування (синтаксичний цукор)

@uah
def money(deposit, years, pc):
    amount = deposit * (1 + pc/100)**years
    return round(amount)

print(money(1000, 2, 10))       # 1210 грн.

# немає інструментів звернутися до початкової (недекорованої) функції:
print(money.__name__)   # wrapper
```

```python
# декорування (розгорнута версія)

def money(deposit, years, pc):
    amount = deposit * (1 + pc/100)**years
    return round(amount)

old_money = money   # створення іншого імені для недекорованої функції
money = uah(money)  # декорування

print(old_money(1000, 2, 10))   # 1210
print(money(1000, 2, 10))       # 1210 грн.
```

Щоб зберегти доступ до атрибутів недекорованої функції (але не до її виклику), можна всередині свого декоратора використовувати декоратор **`functools.wraps`** стандартної бібліотеки Python, який переносить метадані з недекорованого об'єкту (у нашому випадку — функції) в декорований. Саме такий підхід рекомендується використовувати.

```python
import functools

# рекомендований синтаксис декоратора
def uah(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        return f"{func(*args, **kwargs)} грн."
    return wrapper

@uah
def money(deposit, years, pc):
    amount = deposit * (1 + pc/100)**years
    return round(amount)

print(money(1000, 2, 10))       # 1210 грн.
print(money.__name__)           # money
```

До функцій можна застосовувати одночасно кілька декораторів, які викликатимуться у зворотному порядку:

```
def timer(func):
    from time import time
    def wrapper(*args, **kwargs):
        start = time()
        result = func(*args, **kwargs)
        print(f"Time of execution: {time()-start} seconds")
        return result
    return wrapper

@timer
@uah
def money(deposit, years, pc):
    amount = deposit * (1 + pc/100)**years
    return round(amount)

print(money(1000, 2, 10))

# Time of execution: 2.8133392333984375e-05 seconds
# 1210 грн.
```

```python
@timer
def power_of_2(x):
    return 2**x

import math
@timer
def len_factorial(n):
    return len(str(math.factorial(n)))

print(power_of_2(100))
# Time of execution: 5.7220458984375e-06 seconds
# 1267650600228229401496703205376

print(len_factorial(100_000))
# Time of execution: 5.039614200592041 seconds
# 456574

len_factorial(10**5) # Time of execution: 5.053248882293701 seconds
```

ДЕКОРУВАННЯ МЕТОДІВ КЛАСУ

Вбудовані декоратори

Python має декілька вбудованих декораторов, серед яких найвживанішими є:

`@classmethod`

`@staticmethod`

`@property`

Також доступні декоратори зі стандартної бібліотеки (наприклад, **functools.wraps**)

```python
class Class:

    # звичайний метод
    def doubler_1(self, x):
        print(x*2)

    # метод класу
    def doubler_2(class_, x):
        print(x*2)

    # статичний метод
    def doubler_3(x):
        print(x*2)

c = Class()


c.doubler_1(5)      # 10
Class.doubler_1(5)  # TypeError: doubler_1() missing 1 required positional argument: 'x'

c.doubler_2(5)      # 10
Class.doubler_2(5)  # TypeError: doubler_2() missing 1 required positional argument: 'x'

c.doubler_3(5)      # TypeError: doubler_3() takes 1 positional argument but 2 were given
Class.doubler_3(5)  # 10
```

```python
class Class:

    def doubler_1(self, x):
        print(x*2)

    @classmethod
    def doubler_2(class_, x):
        print(x*2)

    @staticmethod
    def doubler_3(x):
        print(x*2)

c = Class()


c.doubler_1(5)      # 10
Class.doubler_1(5)  # TypeError: doubler_1() missing 1 required positional argument: 'x'

c.doubler_2(5)      # 10
Class.doubler_2(5)  # 10

c.doubler_3(5)      # 10
Class.doubler_3(5)  # 10
```

@property

перетворення методу в захищений від змін атрибут (властивість)

```python
class Person:

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"


teacher = Person('Roman', 'Seliverstov')

print(teacher.full_name)                # Roman Seliverstov

teacher.full_name = 'R. Seliverstov'    # AttributeError: can't set attribute
```

```python
class Person:

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
        self.full_name = first_name + last_name


teacher = Person('Roman', 'Seliverstov')

print(teacher.full_name)                    # Roman Seliverstov

teacher.full_name = 'Peter Kravets'
print(teacher.full_name)                    # Peter Kravets
print(teacher.first_name)                   # Roman
```

Геттери та сеттери

```
class Class:

    def __init__(self):
        self._x = None

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

c = Class()

print(c.x)      # None

c.x = 100
print(c.x)      # 100
```

ДЕКОРУВАННЯ КЛАСІВ

```python
def decorator(cls):   # викликається на етапі декорування @

    class Wrapper:

        def __init__(self, *args): # викликається на етапі створення екземпляра
            self.wrapped = cls(*args)

        def __getattr__(self, name): # викликається при звертанні до атрибуту
            return getattr(self.wrapped, name)*3

    return Wrapper

@decorator
class Class: # Class = decorator(Class)
    def __init__(self, x, y): # викликається методом Wrapper.__init__
        self.attr = 'spam'

x = Class(6, 7) # насправді викличе Wrapper(6, 7)
print(x.attr)   # викличе Wrapper.__getattr__, виведе spamspamspam
```

КЛАС-ДЕКОРАТОР

```python
import time

class Timer:

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        start = time.time()
        result = self.func(*args, **kwargs)
        self.time_of_execution = time.time() - start
        print(f"Time of execution: {self.time_of_execution} seconds")
        return result

@Timer
def listcomp(n):
    return [x**2 for x in range(n)]

listcomp(100) # Time of execution: 0.00012874603271484375 seconds
```