České vysoké učení technické v Praze FIT

Programování v Pythonu

Jiří Znamenáček

Příprava studijního programu Informatika je podporována projektem financovaným z Evropského sociálního fondu a rozpočtu hlavního města Prahy.

Praha & EU: Investujeme do vaší budoucnosti

Python – Dekorátory

Úvod

Dekorátory obecně jsou funkce, které berou jako parametr funkci (a případně nějaké další doplňující parametry) a vrací tuto funkci zadaným způsobem upravenou, „odekorovanou“.

> → Čistě teoreticky může dekorátor funkci redukovat na libovolnou návratovou hodnotu, ale to by bylo velmi specifické použití.

Většinou se pro jejich zápis používá konstrukce `@DEKORÁTOR`. Jinými slovy konstrukce..

```
@dekorátor
def moje_funkce():
    BLOK
```

..je vlastně přehlednější náhrada za..

```
def moje_funkce():
    BLOK

moje_funkce = dekorátor(moje_funkce)
```

Poznámka: Stejně jako funkce můžeme zanořovat do sebe, `@dekorátory` můžeme přidávat před definici funkce – aplikují se v uvedeném pořadí.

Konstrukce dekorátoru

Zavedeme-li dekorátor..

```
def dekorátor(funkce):
    def _dekorátor():
        print('START')
        ret = funkce()
        return ret
    return _dekorátor
```

..pak volání následujícím způsobem odekorované funkce..

```
@dekorátor
def moje_funkce():
    print( 'moje funkce' )
```

..vrátí:

```
>>> moje_funkce()
START
moje funkce
```

→ Vypsání textu ještě po zavolání funkce vyžaduje následující úpravu:

```
def _dekorátor():
    print('START')
    ret = funkce()
    print('END')
    return ret
```

Celé toto funguje právě proto, že funkce uvnitř funkcí si s sebou nesou celý aktuální (dynamický) kontext, jak jsme viděli už dříve na případě *uzávěrů* (*closures*).

Příklad

Ukažme si rozumnější příklad – dekorátor, který při volání funkce vypíše její jméno:

```
>>> def decorator(target):
...
...     def wrapper():
...         print( 'Calling function "%s"' % target.__name__ )
...         return target()
...
...     return wrapper

>>> @decorator
... def target():
...     print('I am the target function')

>>> target()
Calling function "target"
I am the target function
```

→ Podle http://www.siafoo.net/article/68

Předávání parametrů funkci

Bude-li dekorovaná funkce očekávat nějaké vstupní parametry, syntaxe se nepatrně změní o vnitřní parametr:

```
>>> def dekorátor(funkce):
...     def _dekorátor(x):
...         print('START')
...         ret = funkce(x)
...         return ret
...     return _dekorátor
...
... @dekorátor
... def moje_funkce(s):
...     print( 'Ahoj, ', s, '!', sep='' )

>>> moje_funkce('Láďo')
START
Ahoj, Láďo!
```

Příklad

Ukažme si rozumnější příklad – dekorátor, který při volání funkce vypíše všechny její parametry a navíc jeden (*debug*) nastaví, čímž i změní její chování:

```
>>> def decorator(target):
...
...     def wrapper(*args, **kwargs):
...         kwargs.update({'debug': True})  # Edit the keyword arg
...         print( 'Calling function "%s" with arguments %s and ke
...         return target(*args, **kwargs)
...
...     wrapper.attribute = 1
...     return wrapper

>>> @decorator
... def target(a, b, debug=False):
...     if debug: print('[Debug] I am the target function')
...     return a + b

>>> target(1,2)
Calling function "target" with arguments (1, 2) and keyword argume
[Debug] I am the target function
3
```

→ Podle http://www.siafoo.net/article/68

Předávání parametrů dekorátoru

Potřebujeme-li předat parametry nikoli dekorované funkci, ale dekorátoru samotnému, konstrukce se zesložití o další mezifunkci, která tyto parametry bude přebírat a posílat dál.

Příklad: Při zavedení následujícího dekorátoru..

```
def omez_hodnoty(maximum):          # funkce přebírající parametry d
    def _omez_hodnoty(funkce):      # funkce přebírající dekorovanou
        def __omez_hodnoty(xs):     # konečná návratová funkce
            ret = funkce( [x for x in xs if x <= maximum] )
            return ret
        return __omez_hodnoty
    return _omez_hodnoty
```

..můžeme provést odekorování např. takto:

```
@omez_hodnoty(9)
def zpracuj_hodnoty(xs):
    return [x**2 for x in xs]
```

Volání funkce *zpracuj_hodnoty()* pak vrátí:

```
>>> test = zpracuj_hodnoty( [1, 3, 11, 4, 7, 14, 5] )
>>> print(test)
[1, 9, 16, 49, 25]
```

Použití

I. Python sám nabízí několik připravených dekorátorů. Již jsme se seznámili s dekorátory `@classmethod` a `@staticmethod` u tříd, které zajišťují, že příslušné metody třídy se nechovají jako instanční, ale jako třídní nebo statické.

II. Z předchozích příkladů je vidět, že častého použití by se mohly dekorátory dočkat např. u logovacích nebo debugovacích funkcí:

- dekorátor pro výpis informací o volaných funkcích
- dekorátor pro ukládání informací o volaných funkcích
- ...

III. Zcela typicky se s dekorátory setkáte v různých webových frameworcích, kde slouží k zajištění vybraných podmínek pro vstupní nebo návratové hodnoty funkcí. Typické příklady:

- funkci je možné vykonat pouze tehdy, je-li daný uživatel přihlášen do systému
- funkce musí v daném kontextu vracet řetězce bez formátovacích prvků
- …

Poznámky

Při použití dekorátorů je třeba pamatovat na to, že „obalovací“ mezifunkce *zcela nahradí* dekorovanou funkci. Což mimo jiné znamená, že pokud budeme dekorované funkci pomocí introspekce klást nějaké otázky o její identitě, dostaneme úplně jiné odpovědi – budou se totiž týkat identity dekorátoru!

Jinými slovy – pokud se nebudete hodně snažit, použití dekorátoru zcela změní tzv. *otisk funkce* (*call signature*; tj. přibližně jméno funkce a seznam jejích argumentů).