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 – Deskriptory

Úvod

Motivační příklad:

> Představte si, že vyrobíte program pro kreslení plošných objektů do canvasu. Každý grafický objekt je představován nějakým programovým objektem. Jeden z *datových* atributů tohoto programového objektu (zvaný třeba `color`) udržuje povědomí o barvě objektu grafického. Uvedený datový atribut je možno číst – dostanete aktuální barvu objektu. Ale též je do něj možné zapisovat – pak by se zřejmě měla zároveň změnit i samotná barva grafického objektu.

Z uvedeného příkladu je zřejmé, že občas (ba dokonce v mnoha aplikacích velmi často) po *datových* atributech chceme, aby se chovaly trochu také jako metody, které něco dělají, a ne jen udržovaly nějakou hodnotu. Toto chování v mnoha jazycích zajišťují tzv. *getters & setters*, v Python'u je však známější pod pojmenováním *properties* a nebo ještě spíše pod jménem protokolu, který slouží (nejen) k jejich implementaci – *deskriptory* (*descriptors*).

Příklad

Ukažme si nejdříve poněkud „ukecaný“, leč poměrně přímočarý způsob, jak zavést na objektu atributy, které se budou chovat požadovaným způsobem:

```
# zavedení objektu
class Plocha:

    def __init__(self, barva=None):
        self._barva = barva

    def _get_barvu(self):
        print("Vracím barvu.")
        return self._barva

    def _set_barvu(self, val):
        print("Nastavuji barvu na:", val)
        self._barva = val

    barva = property(_get_barvu, _set_barvu)

# zavedení instance objektu
p = Plocha()

# „hraní si“ s atributem barva
b = p.barva
print(b)

print()

p.barva = "zelená"

print()

b = p.barva
print(b)
```

→ Zde nepoužitá část pro odstranění atributu by vypadala následovně:

```
def _del_barvu(self):
    print("Odstraňuji barvu.")
    del self._barva

barva = property(_get_barvu, _set_barvu, _del_barvu
```

Spustíme-li uvedený kód, obdržíme následující výstup (vysvětlující komentáře přidány):

```
Vracím barvu.    # důsledek zavolání „b = p.barva“
None             # vrácená hodnota (po __init__() je zatím None)

Nastavuji barvu na: zelená   # důsledek zavolání „p.barva = "zelen

Vracím barvu.    # důsledek zavolání „b = p.barva“
zelená           # vrácená hodnota
```

property() a spol.

I. Vestavěná funkce `property()`, kterou jsme na předchozím slajdu použili pro konstrukci *property*-atributu *barva*, má v úplnosti následující záběr:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

V plné šíři tedy může vygenerovat atribut, který nabídne obslužnou funkci při čtení (*fget*), zápisu (*fset*) a odstranění atributu (*fdel*; nepříliš častá operace) plus zavedení dokumentačního řetězce (*doc*).

→ Není-li parametr *doc* přítomen, pokusí se Python na jeho místo dosadit dokumentační řetězec funkce *fget*.

II. Výše uvedené použití však není příliš pěkné, pokud obslužné funkce nepotřebujeme na nic jiného (což nejspíše bude nejčastější případ) – pokud je totiž nezavedeme jako dvojpodtržítkové (na začátku jména), tak nám budou zcela zbytečně „zaneřáďovat“ jmenný prostor objektu:

```
>>> dir(p)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__f
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '
 '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__red
 '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclassho
 '__weakref__', '_barva', '_get_barvu', '_set_barvu', 'barva']
```

Od Python'u 2.6 se (nejen) uvedený problém řeší pomocí dekorátorů:

```
class Plocha:

    def __init__(self, barva=None):
        self._barva = barva

    @property
    def barva(self):
        print("Vracím barvu.")
        return self._barva

    @barva.setter
    def barva(self, val):
        print("Nastavuji barvu na:", val)
        self._barva = val
```

→ Funkce `property` tedy slouží jako dekorátor pro *getter*. Zde nepoužitý dekorátor pro odstranění atributu by byl `@barva.deleter`.

→ Ve jmenném prostoru objektu zůstanou po této „operaci“ pouze atributy `_barva` a `barva`. Prvního z nich se snadno

zbavíme jeho zavedením jako dvojpodtržítkového:

```
def __init__(self, barva=None):
    self.__barva = barva
```

Je samozřejmě třeba upravit i zbývající volání.

Teorie

I. V kapitole o introspekci jsme viděli, že atributy objektu „žijí“ v jeho jmenném prostoru díky mapování v atributu `__dict__`. Dotaz na „běžný“ (ne-*property*) atribut se tak přeloží na dotaz (resp. přiřazení) do slovníku příslušné instance nebo některé její rodičovské třídy:

```
# a) daný atribut je definován přímo na instanci
INSTANCE.__dict__['ATRIBUT']

# b) daný atribut definuje základní třída této instance
type(INSTANCE).__dict__['ATRIBUT']

# c) daný atribut je definován na některé předchozí rodičovské tří
...
```

II. Pokud se však jedná o *property*-atribut (tj. v odpovídající třídě je atribut definován jako *deskriptor*), přístup zajistí „magická“ metoda `__getattribute__()` (resp. `__setattr__()` a `__delattr__()`), která provede překlad podle typu přístupu (čtení, zápis, odstranění) na volání odpovídající magické metody:

```
# A) čtení
INSTANCE.ATRIBUT
   <=>
type(INSTANCE).__dict__['ATRIBUT'].__get__(INSTANCE, type(INSTANCE

# B) zápis
INSTANCE.ATRIBUT = value
   <=>
type(INSTANCE).__dict__['ATRIBUT'].__set__(INSTANCE, type(INSTANCE
```

→ Pro příslušné dotazy na *třídě* to dopadne trošku jinak, např. `TŘÍDA.ATRIBUT <=> TŘÍDA.__dict__['ATRIBUT'].__get__(None, TŘÍDA)` apod.

PS: Existuje i magická metoda `__getattr__()`. Narozdíl od *__getattribute__()*, která se volá vždy, je-li k dispozici, dochází k volání *__getattr__()* pouze tehdy, nepodaří-li se daný atribut nalézt jiným způsobem.

III. *Datové* deskriptory definují metody `__get__()` i `__set__()`, ***ne**datové*

deskriptory (tedy typicky funkce) na druhou stranu pouze metodu `__get__()`. V praxi největší rozdíl mezi nimi je však především ten, že metody (jakožto nedatové deskriptory) můžeme na instancích „přepsat“, datové deskriptory nikoli.

Poznámky

Deskriptory v Pyhton'u slouží k implementaci celé řady základních věcí:

- *properties* (tj. třídní atributy s kontrolovaným přístupem)
- metody tříd
- statické metody tříd
- třídní metody tříd
- funkce *super()*
- ...