Objektno orienitirano programiranje v Pythonu

Računalništvo
Univerza v Ljubljani, FMF

Koncept

- Program = zaporedje ukazov
- Problem – koda se začne ponavljati
- Rešitev: funkcije (izognemo se ponavljanju identične kode) – definicija na enem mestu, večkratna uporaba
- Objektno orientirano programiranje nam omogoča pakiranje logično sorodne kode in podatkov v objekte in s tem olajša upravljanje kode
- Objekt = entiteta, ki vsebuje podatke (atribute) in funkcije (metode)

Objekti

- Objekti so podatkovno-funkcionalne enote, ki združujejo podatke (**atribute**) in funkcije (**metode**)
- Vse podatkovne enote (kar lahko shranimo v spremenljivko) v Pythonu so objekti

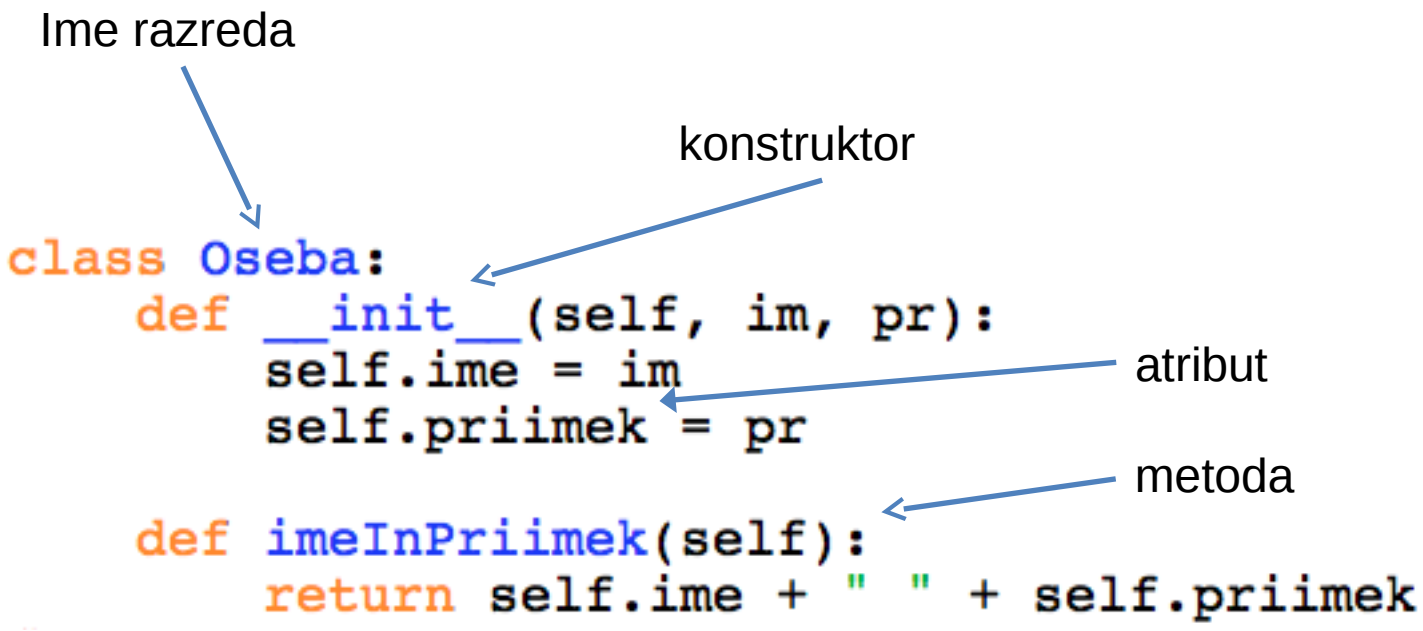
```
>>> a = 1.2
>>> a.real
1.2
>>> a.conjugate()
1.2
```

Razredi

- Vsak objekt ima **tip** (t.j. je nekega razreda)
- Tip je **razred** objekta, ki je določen z neko definicijo (oz. je vgrajen)

```
>>> a = 1
>>> type(a)
<class 'int'>
>>> a = 1.3
>>> type(a)
<class 'float'>
>>> a = "niz"
>>> type(a)
<class 'str'>
>>>
```

Definicija razreda



- **Konstruktor**: posebna metoda, ki se zažene ob generaciji objekta (instance) razreda
- V konstruktorju tipično inicializiramo atribute razreda

Uporaba objektov

```
>>> o = Oseba("Janez", "Novak")
>>> o.ime
'Janez'
>>> o.priimek
'Novak'
>>> o.imeInPriimek
<bound method Oseba.imeInPriimek of <__main__.Oseba object at 0x103e6f8d0>>
>>> o.imeInPriimek()
'Janez Novak'
```

generiranje instance razreda (klic konstruktorja)

dostop do atributov

Ime metode je tudi atribut – vrne objekt definicije metode

Objekt definicije metode je “klicljiv” (callable). Klic izvede metodo skladno z definicijo (namesto prvega parametra vstavi kar sam objekt (instanco).

Objekt definicije razreda

- Ob naloženi definiciji razreda se shrani pod ime razreda

```
>>> Oseba
<class '__main__.Oseba'>
```

- Objekt je *klicljiv* – klic izvede konstruktor in vrne novo instanco
- Do metod lahko dostopamo direktno preko atributov, obnašajo se kot navadne funkcije

```
>>> Oseba.imeInPriimek(o)
'Janez Novak'
```

Pomen prvega parametra

```
class Oseba:
    def __init__(self, im, pr):
        self.ime = im
        self.priimek = pr

    def imeInPriimek(x):
        return x.ime + " " + x.priimek
```

- Ime ni pomembno – pomembno je prvo mesto
- Dogovor: prvi parameter vedno imenujemo *self*

Dodatno o razredih

```
class Oseba:
    """Preprost razred, z imenom in priimkom osebe."""
    x = 3
    print(3)
    def __init__(self, im, pr):
        self.ime = im
        self.priimek = pr

    def imeInPriimek(self):
        return self.ime + " " + self.priimek
```

- Dokumentacijski niz, takoj na začetku
- Načeloma lahko poljubna koda (na začetku ali med definicijo metod), ki se izvede ob nalaganju definicije razreda
- Podobna logika kot v funkciji – različno le obravnavanje definiranih podfunkcij, ki avtomatično postanejo metode
- “Lokalne spremenljivke” postanejo atributi v objektu definicije razreda

Dedovanje

Nov razred prevzame vse lastnosti originalnega

```
class Student(Oseba):
    def __init__(self, im, pr, vpst):
        #klici konstruktorja 'očetovskega' razreda.
        Oseba.__init__(self, im, pr)
        #denimo da so vpisne številke oblike 27llnnnn,
        #kjer je ll letnik vpisa.
        self.vpisnaStevilka = vpst

    def letnik(self):
        return self.vpisnaStevilka // 10000 % 100
```

- Lahko redefiniramo konstruktor, metode
- Lahko dodamo nove metode in nove atribute
- Dedovanje nam prihrani ponavljanje kode

Dedovanje kot koncept abstrakcije

```
class Zival:
    def __init__(self, ime):
        self.ime = ime

    def oglasanje(self):
        return "No comment..."

class Pes(Zival):
    def __init__(self, ime, steviloNog = 4, dolzinaRepa = 0.2):
        Zival.__init__(self, ime)
        self.steviloNog = steviloNog
        self.dolzinaRepa = dolzinaRepa

    def oglasanje(self):
        return "Hov hov"

class Macka(Zival):
    def __init__(self, ime, dolzinaBrk=0.05):
        Zival.__init__(self, ime)
        self.dolzinaBrk = dolzinaBrk

    def oglasanje(self):
        return "Mjav mjav"

def testZivali():
    zivali = [Pes("Fifi"), Macka("Micka", dolzinaBrk=0.1), Macka("Leni")]
    for zival in zivali:
        print(zival.ime, ":", zival.oglasanje())
```

```
>>> testZivali()
Fifi : Hov hov
Micka : Mjav mjav
Leni : Mjav mjav
```

Razred Vektor2D

```
from math import *
class Vektor2D:
    "Dvodimezionalni vektor."
    def __init__(self, x=0, y=0):
        # 'skrita' atributa
        self._x = x
        self._y = y

    def norma(self):
        return sqrt(self._x**2 + self._y**2)
```

- Atributom, katerih imena se začnejo z podčrtajem, rečemo skriti atributi
- Razvojna orodja (IDLE) jih tipično ne prikazujejo
- Dejansko so enako dostopni kot ostali atributi, če poznamo njihovo ime
- Skrivanje torej preko dogovora bolj označuje namen, kot pa da predstavlja orodje zaščite

Lastnosti

```
@property
def norma2(self):
    return sqrt(self._x**2 + self._y**2)
```

```
>>> v = Vektor2D(1,2)
>>> v.norma2
2.23606797749979
```

- Obnašanje kot atributi, a dejansko so to funkcije
- Uporabili smo dekorator @property, ki spremeni funkcijo norma2, preveže (na isto ime) in dopolni, da doseže ciljno delovanje
- Primer uporabe: želimo imeti možnost aktivnosti tako pri vračanju kot pri pisanju v “navidezni atribut”

Lastnost x

```
def vrniX(self):
    "Vrne x."
    print("Vračam x ...")
    return self._x

def nastaviX(self, novi):
    print("Nastavljam x ...")
    if type(novi) is int or type(novi) is float:
        self._x = novi
    else:
        raise Exception()

x = property(fget=vrniX, fset=nastaviX, doc="Komponenta x.")
```

```
>>> v.x
Vračam x ...
1
>>> v.x = 7
Nastavljam x ...
>>> v.x
Vračam x ...
7
```

- Branje in nastavljanje sta dejansko funkciji, kjer lahko kaj postorimo, preverimo
- Pomagamo si s pomočjo funkcije property

Lastnost y (z dekoratorji)

```
@property
def y(self):
    return self._y

@y.setter
def y(self, novi):
    self._y = novi
```

- Enak učinek kot klic funkcije property pri x (razen dokumentacijskega niza)
- Dekorator @property je dejansko na poseben način napisana funkcija oz. razred property. Dekoracija jo uporabi za dopolnjevanje originalne funkcije y(), hkrati pa se izvede prevezava in redefinicija lastnosti y, ki ima pod metodo .setter spet razred oz. funkcijo, s katero je mogoče dekorirati
- Dekoratorji so na poseben način napisane bodisi funkcije bodisi razredi
- Mehanizem definicije razreda ob nalaganju jih zna uporabiti

Metode s posebnim pomenom

- Če na objektu o kličemo funkcijo abs(o), se dejansko izvede metoda o.__abs__()

```
def __abs__(self):
    return self.norma()
```

```
>>> v = Vektor2D(1,2)
>>> abs(v)
2.23606797749979
```

Posebne metode: __repr__

```
def __repr__(self):
    komponente = [str(i) if type(i) is int else "{0:.2f}".format(i) \
                  for i in (self._x, self._y)]
    return "({0},{1})".format(*komponente)
```

```
>>> v = Vektor2D(1,2)
>>> v
(1,2)
```

- IDLE uporablja __repr__ za tekstovno prezentacijo
- Opomba: uporaba formatiranih nizov (metoda format na nizih)
- Opomba: operator * pretvori seznam v zaporedje argumentov

Posebne metode: __add__

```
def __add__(self, u):
    return Vektor2D(self._x + u._x, self._y + u._y)
```

```
>>> u = Vektor2D(1,2)
>>> v = Vektor2D(-3,4)
>>> u + v
(-2,6)
```

- Ko uporabimo operator seštevanje, npr. a + b, se dejansko izvede metoda a.__add__(b)
- Podobno velja za druge operatorje (__sub__, __mul__, …)

Posebne metode: __

```
def __getitem__(self, i):
    if i == 0:
        return self._x
    if i == 1:
        return self._y
    print("Napaka.")
```

```
>>> v = Vektor2D(1,2)
>>> v[0]
1
>>> v[1]
2
>>> v[23]
Napaka.
```

- Operator oglati oklepaj (funkcija, ki se zgodi ob branju)
- Obstaja tudi __setitem__

Razred matrika

```
class Matrika:
    """Matrika 2x2 definirana z dvema vektorjema - stolpcema."""
    def __init__(self, v1, v2):
        self.s1 = v1
        self.s2 = v2

    def __repr__(self):
        return "|{0:4}{1:4}|\n|{2:4}{3:4}|\n".format(
                                    self.s1[0], self.s2[0],
                                    self.s1[1], self.s2[1])
    def __call__(self, v):
        return Vektor2D(self.s1[0]*v[0] + self.s2[0]*v[1],
                        self.s1[1]*v[0] + self.s2[1]*v[1])
```

```
>>> v1 = Vektor2D(1,2)
>>> v2 = Vektor2D(3,4)
>>> m = Matrika(v1, v2)
>>> m
|   1   3|
|   2   4|

>>> m(Vektor2D(1,1))
(4,6)
```

- Tudi funkcionalnost funkcijskega klica lahko definiramo s pomočjo metode __call__

Izjeme

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
```

- Ko se zgodi kaj nenavadnega, npr. deljenje z 0, Python generira izjeme
- Izjeme so spet objekti
- Lahko jih ulovimo, lahko jih sami izvržemo
- Lahko definiramo tudi svoje razrede, ki jih uporabimo kot izjeme

Primer: Razred vektor 2D

- Pisanje v lastnost x kliče metodo nastaviX(...), ki preveri tip podatka za vpis in vrže izjemo Exception(), če vpis ni ustreznega tipa

```
def nastaviX(self, novi):
    print("Nastavljam x ...")
    if type(novi) is int or type(novi) is float:
        self._x = novi
    else:
        raise Exception()
```

```
>>> v = Vektor2D(1,2)
>>> v.x = "abc"
Nastavljam x ...
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    v.x = "abc"
  File "/Volumes/Kingston/delo/vaje/2013-14/prog1/razredi/razredi.py", line 124
, in nastaviX
    raise Exception()
Exception
```

Lastne izjeme

- Definicija lastne izjeme (dedič razreda Exception)

```
class MojaIzjema(Exception):
    def __init__(self, ind):
        self.index = ind
```

- Popravek metode __getitem__ v razredu vektor

```
def __getitem__(self, i):
    if i == 0:
        return self._x
    if i == 1:
        return self._y
    # sprožanje lastne izjeme
    raise MojaIzjema(i)
```

Lovljenje izjem

```
v = Vektor2D(1,2)
try:
    v[3]
except MojaIzjema as e:
    print("Vpisan indeks je {0}.".format(e.index))
```

- Uporabimo zanko try … except

Načini lovljenja izjem

Oglejmo si primer.