OOP - objektivno opširno predavanje

Stefan Nožinić stefan@lugons.org

Ugrađeni tipovi podataka u pythonu

- celi brojevi - a = 5, a = int(5)
- brojevi sa pokretnim zarezom
- boolean
- stringovi

definisanje novih tipova

- definisanje njegovih atributa (polja) i tipova tih atributa
- definisanje operacija nad tipom

Tipovi operacija

- konstruktori - operacija kreiranja nove instance
- metode klase

Enkapsulacija

- sakrivanje detalja implementacije
- korisnik klase vidi samo koje su akcije moguće nad tipom
- ne vidi atribute klase niti implementaciju operacija

Tipovi opoeracija

- konstruktor - stvaranje objekta
- destruktor - poziva se prilikom brisanja objekta
- accessor - računa vrednost na osnovu atributa i vraća je
  - specijalni tip accessora - getter - vraća vrednost određenog atributa
- transfoormer - menja vrednosti atributa
  - specijalan slučaj - setter

Primer

```
class Klasa:
    def __init__(self, a, b, c):
        self.atribut1 = a
        self.atribut2 = b
        self.atribut3 = c

    def __del__(self):
        # destruktor

    def vrati_a(self):
        # getter
        return self.atribut1

    def promeni_b(self, novo_b):
        self.atribut2 = novo_b
```

Primer: date koji čuva dan mesec i godinu i ima operaciju povecanja za n dana

```
class Date:
    def __init__(self, y, m, d):
        self.y = y
        self.m = m
        self.d = d
    def advance(self, n):
        # ...
```

Primer: date koji čuva u unix timestamp

```
class Date:
    def __init__(self, y, m, d):
        self.time = y*365 + 30*m + d
    def advance(self, n):
        self.time += n
```

Primer: upotreba ista, organizacija drugačija!

```
today = Date(2018, 6, 29)
today.advance(5)
# ne interesuje me kooja je interna struktura od ove dve
```

Tipovi transformera na osnovu ponašanja

- mutativni - menja vrednosti atributa unutar instance
- aplikativni - vraća novu instancu klase sa novim vrednostima atributa dok ne menja postojeću instancu

Mutabilna klasa

- je klasa sa bar jednim mutativnim transformerom
- imutabilna - nema mutatiivnog transformera

Hmmm?

- kako napraviti Date da bude imutabilna klasa?

Apstraktni tip podataka

- opisan vrednostima i operacijama
  '+ nije opisan atributima

Privatne metode

- metode koje se mogu pozvati samo iz klase, ne van nje!
- u Pythonu ne postoji distinkcija između privatnih i javnih metoda ali je konvencija da se privatne započnu sa "__" (dve donje crte)
- privatne metode se ne pozivaju van klase

Nasleđivanje

- definisanje nove klase koja ima sve atribute i operacije neke druge klase i još dodatne atribute/operacije
- klasa koja se nasleđuje naziva se nadklasa ili roditelj
- klasa koja nasleđuje se naziva podklasa ili dete klasa
- opšta svojstva i operacije se implementiraju u osnovnu klasu
- podklase koje je nasleđuju specijalizuju ponašanje

Primer nasleđivanja

```
class Building:
    def __init__(self, location_lat, location_long):
        self.location_lat = location_lat
        self.location_long = location_long

    def print(self):
        # ...

class House(Building):
    # dodatne operacije

class Office(Building):
    # dodatne operacije ...
```

Pozivanje konstruktora nadklase

```
class Building:
    def __init__(self, location):
        self.location = location


    def get_location(seelf):
        return self.location

class House(Building):
    def __init__(self, location, family):
        super(House, self).__init__(location)
        self.family = famil

    def get_family(self):
        return self.familyy
```

```
house = House((-22, 50), "Simic")
print(house.get_location())
print(house.get_family())
```

Polimorfizam - Virtuelne metode

- polimorfizam - osobina da operacija promeni svoje ponašanje u zavisnosti od tipa nad kojim je pozvana
- metode čija se implementacija može promeniti u deteetu klase
- npr klasa House da u svom ispisu ima dodatne informacije pored lokacije
- pozivanje metoda roditelja se obavlja sa super(Klasa, self).metoda(argumenti)

Pozivanje metoda nadklase

```
class Foo(Bar):
    def baz(self, arg):
        return super(Foo, self).baz(arg)
```

Apstraktne metode u Pythonu

```
class GameObject:
    def __init__(self, x,y):
        self.x = x
        self.y = y
    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def draw(self):
        raise NotImplementedError
    def step(self):
        raise NotImplementedError
```

```python
class Monster(GameObject):
    def draw(self):
        # ...

    def step(self):
        # logika za pomeranje objekta u svakom frejmu

class Player(GameObject):
    def __init__(self, x, y):
        super(Player, self).__init__(self, x, y)
        self.is_dead = False

    def draw(self):
        # crtanje igraca

    def step(self):
        # logika ...

    def isDead(self):
        return self.is_dead
```

```
m1 = Monster(5,5)
m2 = Monster(100, 100)
player = Player(0, 0)
objects = [player, m1, m2]
while not player.isDead():
    handle_input()
    for obj in objects:
        obj.step()
    for oobj in objects:
        obj.draw()
```

Kompozicija

- sadrži u sebi više instanci različitih klasa
- operacije mogu da definišu operacije nad svim objektima koji čine ovu klasu
- Primer: stack, klasa Game, red čekannja, ...

Invarijante klase

- osoobine koje objekat date klase ima od svog nastanka do svog uništenja
- npr dan u datumu je uvek manji od 31
- mesec je uvek između 1 i 12
- svaka operacija nad objektom treba da održi invarijantu
- Privatna metoda sme da pokvari invarijantu ali se to izbegava

Primer: GameObject klasa

- da li važi invarijanta 0 < x,y < 100?
- da li klase koje nasleđuju GameObject održavaju tu invarijantu?

Nasleđivanje

- Klasa koja nasleđuje klasu održava njenu invarijantu sve dok ne menja privatne atribute (atribute roditelja)
- ovo neki jezici zabranjuju ali Python dozvoljava!

Princip jednog zaduženja

- klasa treba da ima samo jedno zaduženje, da jedna promena u speccifikaciji softvera utiče na to da se klasa mora promeniti
- Primer kako NE TREBA: ReportGenerator kooji se može promeniti ako se menja specifikacija formata i specifikacija konkretnog sadržaja
- Razdvojiti klase da imaju samo jedno zaduženje
- Primer: Report, ReportGenerator, ReportPrinter

Princip ootvoorenosti / zatvorenosti

- kod je otvooren za dodavanje ali je zatvoren za modifikaciju
- ovo znači da treba da dizajniramo softver tako da se on menja dodavanjem novih klasa a da izbegavamo menjanje starih
- Na primer, napravimo interfejse i onda svaka klasa implementira pojedini interfejs

Princim otvorenosti / zatvorenosti

- Primer: interfejs kooji sadrži funkcije za čuvanje teksta u dokument a svaka klasa implementira čuvanje u specifičan format - dodavanje formata = dodavanje nove klase

Da li sledeći kod ispunjava princip otvorenosti-zatvorenosti?

```
class DocumentReader:
    def read(self, path):
        raise NotImplementedError

def ExcelReader(DocumentReader):
    def read(self, path):
        # ...
        return result

def DocReader(DocumentReader):
    # ...
```

```
path = get_input_file_path()
excel = ExcelReader()
content = None
doc = DocReader()
if path.endswith(".xlsx"):
    content = excel.read(path)
else:
    content = doc.read(path)
```

Modifikacija

```
class DocumentReader:
    def read(self, path):
        raise NotImplementedError

    def verify(self, path):
        raise NotImplementedError

def ExcelReader(DocumentReader):
    def read(self, path):
        # ...
        return result
    def verify(self, path):
        return path.endswith(".xlsx")

def DocReader(DocumentReader):
    # ...
```

```
path = get_input_file_path()
readers = [
    DocReader(),
    ExcelReader()
]
for reader in readers:
    if reader.verify(path):
        content = reader.read()
```

Princip Liskove zamene

- Ako klasa S nasleđuje klasu T onda svaki deo programa koji koristi klasu T može biti zamenjen objektima klase S bez promene korektnosti programa

Primer neispunjenog ovog principa

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_width(self):
        return self.width
    def get_height(self):
        return self.height
    def set_width(self, w):
        self.width = w
    def set_height(self, h):
        self.height = h
```

```
class Square(Rectangle):
    def __init__(self, height):
        super(Square, self).__init__(height, height)

a = Square(5)
a.set_width(7) # sta cemo sad?!
```

Princip Liskove zamene

- ako u stvarnom svetu nešto smatramo specijalnim slučajem, to ne znači da i u programu to smatramo specijalnim slučajem klase!
- u primeru klasa Square gubi invarijantu da je širina jednaka dužini
- u gornjem primeru bismo uvek morali da proveravamo da li je objekat tipa Rectangle ili tipa Square što gubi smisao nasleđivanja!

Princip segregacije interfejsa

- bolje je imati više interfejsa nego jedan veliki interfejs
- Na primer, interfejs za čitanje i pisanje za specifičan format
- možda je neki format read-only, pa mu ne treba implementacija za pisanje
- bolje je napraviti dva interfejsa Writable and Readable i onda svaka klasa da implementira ili oba ili samo jedan

Primer

```
class Writable:
    def write(self):
        raise NotImplementedError

class Readable:
    def read(self):
        raise NotImplementedError

class ExcelFormat(Readable, Writable):
    def write(self):
        # ...
    def read(self):
        # ....

class DocFormat(Readable):
    def read(self):
        # ...
```

```
def save(writable: Writable):
    # koristi klasu koja implementira Writable

def open(readable: Readable):
    # koristi klasu koja implementira Readable interfejs
```

Princip inverzije zavisnosti

- svaka klasa da zavisi od interfejsa
- metode koje primaju parametre da primaju interfejse

Zadaci

- Klasa PrimeGenerator koja pri pozivu funkcije get() vraća trenutni prost broj a pri pozivu next() prelazi na sledeći
- Generisanje HTML-a pomoću objekata kao npr h1, h2, ..., p, a, ...
- nadokraditi zadatak 2 da podržava i markdown