

Enkapsulacija, polimorfizam i specijalne metode

Kako objekat čuva svoja pravila, kako isti kod radi sa različitim objektima i kako klasa nauči da sabira

**Danas pravimo: klasu Tacka koja ume da sabira,
kao `pg.Vector2`**

```
polozaj = Tacka(100, 50)
brzina = Tacka(30, 40)
print(polozaj + brzina)
print(brzina * 2)
print(Tacka(1, 2) == Tacka(1, 2))
```

ISPIS

```
(130, 90)
(60, 80)
True
```

Operator `+` radi sa našim objektima, jer klasa ima metodu `__add__`

Tri pitanja za zagrevanje

1 • prošla tema: nasleđivanje

```
class Lik:
    zivoti = 3

class Igrac(Lik):
    pass

i = Igrac()
print(i.zivoti)
```

3

2 • tema 01: reference

```
class Pahuljica:
    def __init__(self, v):
        self.v = v

a = Pahuljica(5)
b = a
b.v = 9
print(a.v)
```

9

3 • mešano: rečnik

```
boje = {"nebo": 90}
boje["sneg"] = 250
print(len(boje))
print(boje["sneg"])
```

2

250

Šest novih reči za danas

enkapsulacija

podaci objekta se menjaju samo kroz metode koje proveravaju

getter i setter

metoda koja vraća vrednost i metoda koja je proverava i upisuje

svojstvo

@property: getter i setter koji se koriste kao običan atribut

polimorfizam

isti poziv radi sa objektima različitih klasa

specijalna metoda

metoda sa dve crte, npr. `__str__`; Python je poziva sam

preopterećenje operatora

naša klasa sama određuje šta znače `+`, `*` i `==`

Igrač sa -2 života: niko nije proverio

```
class Igrac:  
    def __init__(self, ime):  
        self.ime = ime  
        self.zivoti = 3
```

```
igrac = Igrac("Čikica")  
igrac.zivoti -= 5 # pogodilo ga je pet grudvi  
print(igrac.zivoti)
```

ISPIS

-2

Program se ne ruši, a igra je pokvarena. Atribut zivoti može da promeni bilo koji red programa, na bilo koju vrednost. Danas klasa dobija pravo da kaže „ne“.

Enkapsulacija

Podatke menjamo samo kroz metode koje proveravaju

Vozač koristi volan, a motor ne dira

SPOLJA: vozač

- volan, gas, kočnica
- u kodu: metode `predji(km)`, `postavi_tip(tip)`

Ono što svako sme da koristi.

UNUTRA: mehaničar

- motor, kočioni sistem
- u kodu: atributi `_kilometraza`, `_tip`

Ono što zna i menja samo autor klase.

Enkapsulacija: atributi su „ispod haube“, a spolja se objekat menja samo kroz metode koje paze da vrednost ima smisla.

Auto iz starih beleški proverava pređeni put

```
class Auto:
    def __init__(self, kilometraza):
        self.kilometraza = kilometraza

    def predji(self, km):
        if km < 0:
            print("Put ne može biti negativan!")
        else:
            self.kilometraza += km

auto = Auto(2)
auto.predji(100)
auto.predji(-9999)
print(auto.kilometraza)
```

ISPIS

Put ne može biti negativan!
102

- Atribut menja metoda predji, a ne red izvan klase.
- Negativan put se odbija sa porukom.
- Ceo primer (tip, boja, ispis) je u 05_02.

Donje crte su dogovor, a ne katanac

```
class Lik:  
    def __init__(self):  
        self.ime = "Čikica"  
        self._brzina = 220  
        self.__zivoti = 3
```

```
lik = Lik()  
print(lik.ime, lik._brzina)  
print(lik._Lik__zivoti)  
print(lik.__dict__)
```

ime

javno: koristi ga ko god hoće

_brzina

„interno, ne diraj spolja“; Python ništa ne brani

__zivotiPython ga preimenuje u `_Lik__zivoti`

ISPIS

```
Čikica 220  
3  
{'ime': 'Čikica', '_brzina': 220, '_Lik__zivoti': 3}
```

Čudovište: AttributeError zbog dve crte

```
class Igrac:  
    def __init__(self):  
        self.__zivoti = 3  
  
igrac = Igrac()  
print(igrac.__zivoti)
```

PORUKA

```
AttributeError:  
'Igrac' object has no attribute '__zivoti'
```

Kako izgleda

AttributeError: objekat „nema“ atribut koji smo upravo napravili u `__init__`.

Zašto se pojavljuje

Ime sa dve crte Python u klasi preimenuje u `_Igrac__zivoti`, pa spolja `__zivoti` ne postoji.

Kako se pobeđuje

Spolja koristite metodu ili svojstvo, a za „interne“ attribute pišite jednu crtu: `_zivoti`.

Setter proverava, a getter samo vraća

```
class Igrac:
    def __init__(self):
        self._zivoti = 3

    def vrati_zivote(self):          # getter
        return self._zivoti

    def postavi_zivote(self, n):    # setter
        if n >= 0:
            self._zivoti = n

igrac = Igrac()
igrac.postavi_zivote(-2)
print(igrac.vrati_zivote())
igrac.postavi_zivote(5)
print(igrac.vrati_zivote())
```

ISPIS

3
5

- -2 je odbijeno: životi su i dalje 3.
- Jedna crta u _zivoti kaže: spolja koristite metode.
- Ispravka starog slajda: obe metode imaju self.

Šta ispisuje ovaj program?

```
class Zvucnik:  
    def __init__(self):  
        self._jacina = 50  
  
    def postavi(self, j):  
        if 0 <= j <= 100:  
            self._jacina = j  
  
z = Zvucnik()  
z.postavi(120)  
z.postavi(80)  
z.postavi(-5)  
print(z._jacina)
```

Seter prihvata samo brojeve od 0 do 100.

A

50

B

80

tačno

C

-5

D

120

Mana getera i setera: menja se ceo program

Pre: običan atribut

```
c = Celzijus()  
c.temperatura = 37  
print(c.temperatura)
```

Posle: geter i seter

```
c = Celzijus()  
c.postavi_temperaturu(37)  
print(c.vrati_temperaturu())
```

Klasa Celzijus iz starih beleški dobila je proveru: ispod $-273,15$ °C ne može.

Ali sada svako mesto u programu koje je pisalo `c.temperatura` mora da se prepravi. U igri sa hiljadu redova to je hiljadu prilika za grešku. Svojstvo rešava baš ovo.

@property: provera ostaje, a zapis je stari

```
class Celzijus:
    def __init__(self, temperatura=0):
        self.temperatura = temperatura    # seter

    @property
    def temperatura(self):                # getter
        return self._temperatura

    @temperatura.setter
    def temperatura(self, t):             # seter
        if t < -273.15:
            raise ValueError("Prehladno!")
        self._temperatura = t
```

```
c = Celzijus(100)
print(c.temperatura)
c.temperatura = -300
```

ISPIS

```
100
ValueError: Prehladno!
```

c.temperatura = -300 izgleda kao obična dodela, a zapravo poziva seter.

Spolja se piše isto kao pre, a provera radi.

Svojstvo zivoti nikad ne pada ispod nule

```
class Igrac:  
    def __init__(self, ime):  
        self.ime = ime  
        self.zivoti = 3
```

```
@property  
def zivoti(self):  
    return self._zivoti
```

```
@zivoti.setter  
def zivoti(self, n):  
    self._zivoti = max(0, n)    # nikad ispod nule
```

```
igrac = Igrac("Čikica")  
igrac.zivoti -= 5  
print(igrac.zivoti)
```

ISPIS

0

- Isti red igrac.zivoti -= 5 kao na početku časa.
- Sada je rezultat 0, a ne -2.
- max(0, n) vraća veći od dva broja.

Šta ispisuje ovaj program?

```
class Svetlo:
    def __init__(self):
        self.boja = "bela"
    @property
    def boja(self):
        print("čitam")
        return self._boja
    @boja.setter
    def boja(self, b):
        print("pišem")
        self._boja = b

s = Svetlo()
print(s.boja)
```

A

bela

Bčitam
bela**C**

tačno

pišem
čitam
bela**D**pišem
bela

Polimorfizam

Isti poziv, različiti objekti

Jedna petlja radi sa svim objektima igre

```
class Pahuljica:
    def opis(self):
        return "pada i njiše se"

class Prepreka:
    def opis(self):
        return "oduzima život"

class Jelka:
    def opis(self):
        return "stoji i krasí scenu"

objekti = [Pahuljica(), Prepreka(), Jelka()]
for o in objekti:
    print(o.opis())
```

ISPIS

```
pada i njiše se
oduzima život
stoji i krasí scenu
```

- Tri klase, jedna metoda istog imena.
- Petlja ne pita kojoj klasi o pripada.
- U temi 09 isto radi grupa: svakom sprajtu pozove update.

Funkcija ne pita kojoj klasi objekat pripada

```
class Golub:
    def leti(self):
        print("Golub leti iznad trga")

class Pingvin:
    def leti(self):
        print("Pingvin ne leti, ali pliva")
```

```
def probaj_let(ptica):
    ptica.leti()          # ne pita koja je klasa
```

```
probaj_let(Golub())
probaj_let(Pingvin())
```

„Ako hoda kao patka i kvače
kao patka...”

Pythonu je dovoljno da objekat ima metodu leti. Klase Golub i Pingvin nemaju zajedničkog roditelja.

ISPIS

```
Golub leti iznad trga
Pingvin ne leti, ali pliva
```

Šta ispisuje ovaj program?

```
class ObjekatIgre:
    def opis(self):
        return "nešto u igri"

class Pahuljica(ObjekatIgre):
    def opis(self):
        return "pahuljica: hvataj je!"

class Kamen(ObjekatIgre):
    pass

for o in [Pahuljica(), Kamen()]:
    print(o.opis())
```

A

nešto u igri
nešto u igri

B

tačno

pahuljica:
hvataj je!
nešto u igri

C

pahuljica:
hvataj je!
AttributeError

D

pahuljica:
hvataj je!

Python nema preopterećenje metoda

```
class Poruka:  
    def prikazi(self):  
        print("Obična poruka")  
  
    def prikazi(self, tekst):  
        print("Hitno:", tekst)  
  
p = Poruka()  
p.prikazi("sneg pada")  
p.prikazi()
```

Druga definicija briše prvu

Kao kad promenljivoj dodelimo novu vrednost: ostaje samo poslednja metoda prikazi, ona sa parametrom tekst.

ISPIS

```
Hitno: снег pada  
TypeError: Poruka.prikazi() missing 1 required positional argument: 'tekst'
```

Jedna metoda, dva načina poziva

```
class Poruka:  
    def prikazi(self, tekst=None):  
        if tekst is None:  
            print("Obična poruka")  
        else:  
            print("Hitno:", tekst)  
  
p = Poruka()  
p.prikazi()  
p.prikazi("vreme je za Python")
```

ISPIS

Obična poruka

Hitno: vreme je za Python

- tekst=None je podrazumevana vrednost.
- Poziv bez argumenta: tekst je None.
- Zadatak iz starih beleški: klasa Poruka.

Specijalne metode

Metode sa dve crte koje Python poziva sam

Mi pišemo izraz, a Python poziva metodu

<code>print(x), str(x)</code>	→ <code>__str__</code>	kako se objekat ispisuje
<code>len(x)</code>	→ <code>__len__</code>	koliko „stvari“ objekat ima
<code>a + b</code>	→ <code>__add__</code>	šta znači sabiranje
<code>a * 2</code>	→ <code>__mul__</code>	šta znači množenje
<code>a == b</code>	→ <code>__eq__</code>	kada su dva objekta jednaka

`__init__` već znamo: i njega Python poziva sam, kad napravimo objekat.

__str__ određuje šta ispisuje print

Bez __str__

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

```
i = Igrac("Čikica", 3)  
print(i)
```

ISPIS

```
<__main__.Igrac object at 0x7f9186736ba0>
```

Sa __str__

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

```
    def __str__(self):  
        return f"{self.ime}, života: {self.zivoti}"
```

```
i = Igrac("Čikica", 3)  
print(i)
```

ISPIS

```
Čikica, života: 3
```

Inventar zna koliko ima predmeta

```
class Inventar:
    def __init__(self, predmeti):
        self.predmeti = list(predmeti)

    def __len__(self):
        return len(self.predmeti)

    def __str__(self):
        return ", ".join(self.predmeti)

ranac = Inventar(["šal", "kapa", "sanke"])
print(len(ranac))
print(ranac)
```

ISPIS

```
3
šal, kapa, sanke
```

- len(ranac) poziva __len__.
- print(ranac) poziva __str__.
- Zadatak iz starih beleški: Narudzbina sa korpom.

Kucamo zajedno: Tacka koja ume da sabira

Cilj: za 20 minuta klasa Tacka radi sa +, * i ==, kao pg.Vector2 iz teme 07.

- 1 `__init__`: x i y
- 2 `__str__`: ispis (x, y)
- 3 `__add__`: zbir dve tačke je nova tačka
- 4 `__mul__`: tačka puta broj
- 5 `__eq__`: iste koordinate znače jednake

KAKO IZGLEDA NA KRAJU

```
polozaj = Tacka(100, 50)
brzina = Tacka(30, 40)
print(polozaj + brzina)
print(brzina * 2)
print(Tacka(1, 2) == Tacka(1, 2))
```

(130, 90)

(60, 80)

True

Ceo program: 05_23_tacka_i_vector2.py

Klasa Tacka: pet metoda, nijednu ne pozivamo sami

```
class Tacka:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})"

    def __add__(self, druga):
        return Tacka(self.x + druga.x, self.y + druga.y)

    def __mul__(self, broj):
        return Tacka(self.x * broj, self.y * broj)

    def __eq__(self, druga):
        return self.x == druga.x and self.y == druga.y
```

`__add__`

vraća NOVU tačku; stare ostaju iste

`druga`

tačka sa desne strane znaka +

`__eq__`

vraća True ili False

Operatori pozivaju naše metode

```
polozaj = Tacka(100, 50)
brzina = Tacka(30, 40)
print(polozaj + brzina)
print(brzina * 2)
print(Tacka(1, 2) == Tacka(1, 2))
```

ISPIS

```
(130, 90)
(60, 80)
True
```

PYTHON ZAPRAVO IZVRŠAVA

polozaj + brzina	→	polozaj.__add__(brzina)
brzina * 2	→	brzina.__mul__(2)
a == b	→	a.__eq__(b)

pg.Vector2 je ista ideja, samo gotova

```
import pygame as pg

polozaj = pg.Vector2(100, 50)
brzina = pg.Vector2(30, 40)
print(polozaj + brzina)
print(brzina * 2)
print(pg.Vector2(1, 2) == pg.Vector2(1, 2))
print(brzina.length())
```

ISPIS

```
[130, 90]
[60, 80]
True
50.0
```

Bez prozora, samo računanje. Vector2 je klasa sa `__add__`, `__mul__` i `__eq__`, kao naša Tacka. Ima još mnogo metoda (dužina, okretanje). U temi 07 njime pomeramo pahuljice.

Šta ispisuje ovaj program?

```
a = Tacka(2, 3)
b = a + Tacka(1, 1)
print(a, b)
```

Klasa Tacka je ista kao na prethodnim slajdovima.

A

(3, 4) (3, 4)

B

tačno

(2, 3) (3, 4)

C

(2, 3) (1, 1)

D

TypeError

Pronađi grešku: print prijavljuje grešku

```
class Pahuljica:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        print(f"Pahuljica na ({self.x}, {self.y})")

p = Pahuljica(10, 20)
print(p)
```

Rešenje

`__str__` mora da VRATI string. Ovaj ga ispiše i vrati `None`. Ispravka: `return f"Pahuljica na (...)"`.

ISPIS

```
Pahuljica na (10, 20)
TypeError: __str__ returned non-string (type NoneType)
```

Izaberite nivo i počnite

LEVEL 1

Zvučnik

Svojstvo jčina je uvek između 0 i 100, šta god da upišemo. `__str__` ispisuje „Jačina: 70“.

Pomoć: `min(100, max(0, j))`

LEVEL 2

Ranac

U ranac staje najviše pet predmeta. Šesti se odbija porukom. Dodajte `__len__` i `__str__`.

Pomoć: metoda dodaj proverava `len`.

LEVEL 3

Boje se sabiraju

`Boja(255, 0, 0) + Boja(0, 0, 255)` daje `#FF00FF`, kao svetla na sceni.

Pomoć: `min(255, ...)` i `f"#{r:02X}"`.

Sledeći put otvaramo prozor

```
import pygame as pg

pg.init()
ekran = pg.display.set_mode((640, 360))
okvir = ekran.get_rect()
print(okvir.center)
print(okvir)
```

ISPIS

```
(320, 180)
Rect(0, 0, 640, 360)
```

U temi 06 crtamo zimsku scenu igre „Zimska šetnja“. I tamo je sve objekat: prozor je površina (Surface), a njen okvir je pravougaonik (Rect) sa atributima center, midbottom i drugim.

Rezime

enkapsulacija

```
self._zivoti # dogovor  
self.__x    # → _Klasa__x
```

Spolja se menja kroz metode.

svojstvo

```
@property  
@zivoti.setter
```

Izgleda kao atribut, a proverava.

polimorfizam

```
for o in objekti:  
    o.opis()
```

Isti poziv, različite klase.

specijalne metode

```
__str__  __len__  
__add__  __eq__
```

Python ih poziva sam.

Pitanje pre odlaska

```
# klasa Tacka sa slajda „Kucamo zajedno“  
t = Tacka(3, 4) * 2 + Tacka(1, 1)  
print(t)
```

Šta ispisuje program? Koje dve specijalne metode Python poziva u drugom redu?

Pošaljite ispis i imena metoda pre nego što izađete.



Literatura

- Lutz, M. Learning Python, 5. izdanje, O'Reilly, 2013 (poglavljja o klasama i preopterećenju operatora)
- Wentworth, P., Elkner, J., Downey, A. B., Meyers, C. How to Think Like a Computer Scientist: Learning with Python 3 (besplatna knjiga na internetu)
- Zvanična dokumentacija za Python: tutorijal, poglavlje 9 „Classes“; „Data model“, odeljak 3.3 „Special method names“
- Dokumentacija za pygame-ce, modul math (Vector2): pyga.me/docs

Dodatna literatura i zanimljivi sajtovi

Python tutorijal: Classes

docs.python.org/3/tutorial/classes.html

Poglavlje o klasama; odeljak 9.6 objašnjava imena sa dve donje crte.

Python: specijalne metode

[docs.python.org · Special method names](https://docs.python.org/3/library/specialmethodnames.html)

Spisak svih specijalnih metoda: `__str__`, `__len__`, `__add__`, `__eq__` i mnogi drugi.

Python: property

[docs.python.org · property](https://docs.python.org/3/library/property.html)

Zvanični opis svojstva, sa primerom za `@property` i seter.

Python Tutor

pythontutor.com

Crta korak po korak šta program radi i šta je u kojoj promenljivoj.

pygame-ce: Vector2

[pyga.me/docs/ref/math.html](https://pygame-ce.github.io/en/latest/math.html)

Sve što vektor ume: sabiranje, dužina, normalizacija, rotacija.

Hvala!

Pitanja su uvek dobrodošla: na času, u četvrtu ili međim.

Slajdovi, primeri i kartica za podsećanje su na sajtu predmeta.

PREDAVANJA

dr Staša Vujičić Stanković

stasa.vujicic.stankovic@math.rs

www.matf.bg.ac.rs/~stasa.vujicic.stankovic

VEŽBE

Milan Kocić

milan.kocic@matf.bg.ac.rs

poincare.matf.bg.ac.rs/~milan.kocic