1 Osnove objektno usmjerenog programiranja

U ovom poglavlju naučit ćeš:

- razlikovati osnovne pojmove vezane za objektno usmjereno programiranje
- oblikovati klasu s pripadnim elementima za zadani jednostavni problem
- apstrahirati zadani problem
- definirati operacije i funkcije nad klasama s pomoću specijalnih metoda
- primijeniti principe objektno usmjerenog programiranja: nasljeđivanje i enkapuslaciju pri kreiranju klasa.

1.1. Objekti i klase objekata

Pri rješavanju složenih problema u današnje se vrijeme ustalio tzv. objektno usmjereni pristup kojim se nastoji što vjernije opisati i modelirati stvarni svijet iz kojeg problem potječe. Primjerice, u trgovinama se na blagajnama izdavanje računa obavlja računalom, slika 1.1. Blagajna je povezana s računalnim sustavom u kojem se evidentira i broji svaki prodani proizvod. Tako se može automatski dojaviti skladištu kada treba obnoviti zalihe i poslati narudžbu dobavljaču tog proizvoda.

Podatci za proizvod mogli bi biti:

- cijena
- naziv
- vrsta pakiranja
- masa
- porez i sl.

U računu se pojavljuju i drugi elementi kao što su:

- broj računa
- datum računa
- prodavač
- lista proizvoda
- ukupni iznos i sl.

Slika 1.1.

U navedenom primjeru računalni programi oponašaju poslovne procese u trgovini. Na sličan se način mogu promatrati i drugi sustavi. Primjenu računala imamo u proizvodnji, prometu, znanosti, financijama, školama i sl.

U svim se tim primjenama pojavljuju određeni objekti s kojima treba obaviti neke željene aktivnosti. Objekti u stvarnom životu obično predstavljaju nešto konkretno i opipljivo. Od rane mladosti susrećemo se s objektima, primjerice igračkama, stvarima u okolini itd. Objekti su

prepoznatljivi po boji, obliku, zvuku, okusu i sličnom – imaju svoje atribute. S objektima se može nešto činiti (umetnuti trokutasti oblik u otvor u obliku trokuta, zazvoniti zvoncem i sl.), kažemo da postoje neke **metode** za manipuliranje objektima. Pri programiranju se takvi konkretni **objekti** opisuju modelima objekata. U modelima postoje **atributi** koji opisuju objekt određenim podatcima. Ponašanje modela objekata određuje se programskim **metodama** koje djeluju na te atribute. Modeliranje klase za konkretni objekt nazivamo **apstrakcija**.

Objekti koji se međusobno neznatno razlikuju mogu se svrstati u tzv. **klase objekata** ili kraće: **klase**.

Primjerice, ako se radi o trgovini pametnim telefonima, tada se svi objekti – telefoni – mogu svrstati u jednu **klasu**, nazovimo je: *Telefon*. Za sve **objekte** iz te klase možemo promatrati **atribute** kao što su: proizvođač, model, svojstva kamere, kapacitet baterije, veličina memorije, način spajanja na mrežu, operacijski sustav, boja itd. Kada želimo isprobati svojstva odabranog pametnog telefona, uključimo ga i ispitujemo ponašanje tog objekta i **metode** kojima utječemo na njegovo ponašanje: uspostava poziva, kvaliteta slike, prilagodba osobnih postavki, spajanje na internet i sl.

Pokazalo se da se na takav način može sistematizirati i znatno olakšati rješavanje mnogih složenih problema. Općenito gledano, razlikujemo tri koraka:

- objektno usmjerenu analizu problema (OOA od engl. *Object-Oriented Analysis*)
- objektno usmjereno zasnivanje (dizajn) rješenja (OOD od engl. *Object-Oriented Design*)
- objektno usmjereno programiranje (OOP od engl. *Object-Oriented Programming*).

Objektno usmjerena analiza problema prvi je korak u rješavanju problema. U tom prvom koraku mora se ustanoviti što se želi postići, mora se prepoznati koji objekti postoje i kako oni međusobno djeluju. Rezultat analize je opis mogućeg rješenja i grubi odabir objekata, njihovih atributa i metoda.

U drugom se koraku, objektno usmjerenom dizajnu rješenja, rezultati prvog koraka moraju dodatno doraditi. Mora se osmisliti izgled i struktura cijelog programskog rješenja te opisati sve objekte, svrstati ih u klase i za te klase odrediti atribute i metode. Rezultat tog drugog koraka je specifikacija koja može poslužiti kao osnova za programiranje u bilo kojem objektno usmjerenom jeziku.

Konačno, zadnji je korak objektno usmjereno programiranje. U tom se koraku razrađuju konkretna programska rješenja u odabranom, objektno usmjerenom programskom jeziku. Mi ćemo to činiti u *Pythonu*.

U praksi se prethodno opisana tri koraka međusobno isprepliću tako da ih se katkada teško prepoznaje. U nastavku ćemo se baviti uglavnom trećim korakom – objektno usmjerenim programiranjem. Pritom će težište biti na načinima priprema vlastitih klasa. U svakoj ćemo klasi morati odrediti atribute i metode. Atributi će u ostvarenju klase postati podatci te klase, a metode ćemo ostvariti kao funkcije koje pripadaju toj klasi.

Drugim riječima, klase možemo promatrati kao tip podatka za koji su definirane operacije ostvarene funkcijama koje zovemo **metodama**. Kao što nekim varijablama pridružujemo vrijednosti određenog tipa podataka (`int`, `str`, `list` i druge), tako možemo pridružiti i konkretne "vrijednosti" dane klase pri čemu govorimo da je to **objekt** te **klase**. Objekte ćemo nazivati **jedinkama** klase (engl. *instance*). Nazive **objekt** klase i **jedinka** klase smatrat ćemo sinonimima.

1.2. Svi su tipovi podataka u jeziku *Python* klase

Već pri početnom izučavanju programskog jezika *Python* ustanovili smo da za podatkovne zbirke (stringove, liste, rječnike, skupove) postoje neki operatori i ugrađene funkcije te posebni oblik funkcija koje nazivamo metodama. Tako primjerice, za stringove postoje operatori: `+`, `*`, `in`, `not in`, funkcije: `len()`, `min()`, `max()` te metode kao što su: `center(w)`, `ljust(w)`, `rjust(w)`, `capitalize()`, `lower()`, `strip()`, `index(s)`.

Možemo primijetiti da zbirka **string** ima metode, a atribut ove zbirke može biti sam tekst koji se nalazi u toj varijabli te se nad njim obavljaju navedene metode. Zbog toga se umjesto naziva tip podataka može rabiti naziv **klasa**, a umjesto naziva vrijednost naziv **jedinka klase** ili **objekt**.

Posebno je zanimljivo da se i izrazi napisani operatorima mogu napisati s pomoću metoda. Naime, u *Pythonu* postoje tzv. specijalne metode (engl. *special methods*) kojima se ostvaruju operatori. Tako se, primjerice, operator + ostvaruje metodom `__add__()`, a operator * metodom `__mul__()`. Provjerimo to u interaktivnom sučelju:

```
>>> s1 = 'abc'
>>> s2 = 'bcd'
>>> s1 + s2
'abcbcd'
>>> s1.__add__(s2)
'abcbcd'
>>> s1 * 3
'abcabcabc'
>>> s1.__mul__(3)
'abcabcabc'
```

Uočimo da specijalne metode počinju i završavaju dvjema donjim crticama čime je označeno da su to specijalne metode.

I osnovni brojčani tipovi podataka `int` i `float` također su klase. Za te dvije klase postoje specijalne metode kojima se ostvaruju operacije zadane pojedinim operatorima. Pogledajmo to na nekoliko primjera:

```
>>> a = 120
>>> b = 35
>>> a + b
155
>>> a.__add__(b)
155
>>> a / b
3.4285714285714284
>>> a.__truediv__(b)
3.4285714285714284
```

```
>>> a // b
3
>>> a.__floordiv__(b)
3
>>> a % b
15
>>> a.__mod__(b)
15
```

Dakle, svi su tipovi podataka u *Pythonu* klase (engl. *class*), a sve su pojedine vrijednosti jedinke (engl. *instances*), odnosno objekti (engl. *objects*) klasa.

1.3. Oblikovanje klasa u jeziku *Python*

Mnogi se programi mogu napisati i bez poznavanja i uporabe klasa. Međutim, pokazalo se tijekom vremena da se uporabom klasa mogu pripremiti programi koji su znatno pregledniji i koji se mogu bitno jednostavnije nadopunjavati i mijenjati.

Programiranje uporabom klasa omogućuje većina suvremenih programskih jezika kao što su *Java*, C++ i C#. Danas se skoro svi veliki programski sustavi u svijetu zasnivaju na programiranju uporabom klasa.

Svaka klasa ima svoj naziv. U skladu s dogovorom u *Pythonu* se naziv klase piše velikim početnim slovom (primjerice: `Točka`, `Pravac`, `Razlomak`). Ako se naziv sastoji od više riječi, one se po tom dogovoru pišu bez razmaka, ali velikim početnim slovom svake riječi (primjerice: `MojaPrvaKlasa`).

Klasu ćemo u *Pythonu* definirati s pomoću:

```
class ImeKlase:
    definicija klase
```

Definicija klase obuhvaća definicije njenih metoda i atributa. Jedna metoda ima osobito važno značenje. Ime te metode je predefinirano i oblika je `__init__(self, parametri)`. Ova metoda naziva se **konstruktor** i ona se izvodi prilikom kreiranja objekta iz klase. Osnovna namjena ove metode postavljanje je atributa (svojstava) klase na neku vrijednost. Atributi klase na razini klase su globalni. To znači da im se može pristupiti iz svih metoda. Svakom atributu pristupamo na sljedeći način: `self.ime_atributa`. Isto tako, u svim metodama (koje će biti definirane jednako kao i funkcije) mora se pojaviti parametar `self` kao prvi parametar (nekada će to biti i jedini parametar). Taj parametar predstavlja pojedinu jedinku ili objekt te klase nad kojim se metoda poziva. Prisjetimo se da se metode klase pozivaju tako da se napiše ime objekta, iza njega točka i zatim naziv metode. Kažemo i da metoda pripada tom objektu. Pri definiciji metode to će biti označeno tim prvim parametrom `self`.

Ilustrirajmo kreiranje i upotrebu klase na jednom jednostavnom primjeru:

Primjer 1.1. Odredimo neke osnovne atribute i metode za kvadrat te kreirajmo klasu **Kvadrat** s pripadnim atributima i metodama.

U ovom slučaju imat ćemo samo jedan atribut, stranicu kvadrata (`stranica`). Klasa će imati metode `opseg()` za računanje opsega kvadrata i `povrsina()` za računanje površine kvadrata.

Definicija klase imat će sljedeći oblik:

```
class Kvadrat:
    def __init__(self, a=0):
        self.stranica = a

    def opseg(self):
        return 4 * self.stranica

    def povrsina(self):
        return self.stranica**2
```

Prvo ilustrirajmo upotrebu ove klase na nekoliko primjera. Pojedinu jedinku klase (objekt) kreirat ćemo tako da uvedemo varijablu `naziv_objekta` i pridružimo joj `NazivKlase` s odabranom početnom vrijednošću parametra:

```
naziv_objekta = NazivKlase(parametri)
```

Vrijednosti objekta možemo dobaviti tako da napišemo:

```
naziv_objekta.naziv_elementa
```

U našem bi primjeru klase `Kvadrat` definiranje objekta imena `k` i stranice vrijednosti 3 izgledalo ovako:

```
>>> k = Kvadrat(3)
```

Atributu `stranica` klase `Kvadrat` za ovaj objekt (jedinku klase) pridružuje se vrijednost 3. Što možemo i provjeriti:

```
>>> k.stranica
3
```

Za kreirani objekt metode će dati sljedeće rezultate:

```
>>> k.opseg()
12
>>> k.povrsina()
9
```

Vrijednosti atributa moguće je i naknadno mijenjati:

```
>>> k.stranica = 5
>>> k.stranica
5
>>> k.opseg()
20
>>> k.povrsina()
25
```

Vratimo se sada na samu definiciju klase. Konstruktor klase, kao što smo rekli, metoda je čije je ime `__init__()`. Parametar a u ovoj je metodi definiran kao opcijski parametar s inicijalnom vrijednosti 0. Primijetimo da se opcijski parametri mogu izostaviti pri pozivu funkcije te da će tada njihove vrijednost biti jednake odabranim opcijskim vrijednostima. To vrijedi i za metode. Prema tome, atribut `stranica` koji je u metodi `__init__()` određen sa `self.stranica = a` poprimit će vrijednost 0 ako pri kreiranju izostavimo parametar.

```
>>> k = Kvadrat()
>>> k.stranica
0
>>> k.opseg()
0
```

Ako vrijednost parametra promijenimo naknadno, dobivamo rezultat u skladu s unesenom vrijednosti:

```
>>> k.stranica = 4
>>> k.opseg()
16
```

Na prvi bi nas pogled moglo iznenaditi što metode za računanje opsega i površine nemaju parametar a (duljina stranice kvadrata). To je jedna od činjenica po kojoj se objektno usmjereno programiranje donekle razlikuje od strukturnog programiranja. U strukturnom programiranju pri svakom pozivu neke funkcije moramo napisati i njezine ulazne parametre. Ovdje smo prilikom kreiranja objekta odredili parametar koji se trajno pohranjuje unutar jedinke klase i tamo ostaje pohranjen. Prema tome, sve metode definirane unutar klase "znaju" tu vrijednost i nije ju potrebno unositi kao parametar pri pozivu metode. Ako kreiramo više objekata klase, onda će svaka od njih imati svoje vlastite kopije vrijednosti atributa.

Ime `self` u definiciji metoda klase bit će pri pozivu metoda klase nadomješteno imenom konkretnog objekta koji ju je pokrenuo (možemo reći: pozvao). Tako smo u klasi `Kvadrat()` preko konstruktora definirali vrijednost atributa `self.stranica`. U definiciji metode `opseg()` za računanje opsega piše: `4 * self.stranica`. Kada objekt `k` koristi metodu, naziv `self` bit će zamijenjen nazivom `k` (primjerice u izrazu `k.opseg()`, `self` predstavlja varijablu `k`).

Ponovimo, ako atributima ili metodama pristupamo u programima (izvan definicije klase), onda im pristupamo preko imena objekta i to na način da iza imena objekta (jedinke te klase) stavimo točku i nakon toga naziv atributa ili metode (`objekt.naziv_elementa`), primjerice `k.stranica = 4`.

1.4. Metode s dodatnim parametrima

Ponekad ćemo kod nekih metoda imati i parametre koji nisu atributi klase, ali su nam potrebni unutar tih metoda. Vrijednosti takvih parametara možemo proslijediti pojedinim metodama neposredno. Takve parametre nazivamo **parametrima metoda**. Za razliku od atributa koji su "poznati" svim metodama klase, takvi se parametri moraju pojaviti u pozivu metode i vidljivi su samo u toj metodi.

Ilustrirajmo to primjerom.

Primjer 1.2. Kreiranoj klasi **Kvadrat** dodajmo još jednu metodu koja će crtati pripadni kvadrat stranice **stranica** čije će sjecište dijagonala biti u središtu kornjačinog koordinatnog sustava. Kvadrat treba biti obojen nekom zadanom bojom **B**.

Primijetimo da nam ovdje za crtanje treba još boja kvadrata `B`. Ovaj parametar nije egzistencijalan za klasu `Kvadrat`, kao što je to primjerice stranica kvadrata `stranica`, međutim važan nam je za crtanje kvadrata, stoga će to biti parametri metode koju ćemo kreirati. Parametar `self.stranica` poznat je unutar cijele klase. Parametar `B` nije parametar klase i njegova će vrijednost biti definirana pozivom metode.

Dakle, metoda koja će crtati zadani kvadrat imat će sljedeći oblik:

```
def crtaj(self, B):
    pu()
    goto(-self.stranica//2, -self.stranica//2)
    pd()
    color(B)
    begin_fill()
    for i in range(4):
        fd(self.stranica)
        lt(90)
    end_fill()
    ht()
```

Primijetimo da smo u metodi `crtaj()` rabili parametar `B` bez prefiksa `self` jer se radi o parametru samo ove metode, dok smo uz parametar `stranica`, koji je atribut klase, rabili prefiks `self`. Važno je napomenuti da smo na samom početku definicije ove klase trebali pozvati modul `turtle` naredbom: `from turtle import *`.

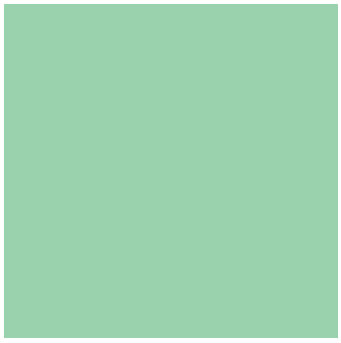
Ilustrirajmo sada uporabu tako nadograđene klase `Kvadrat`:

```
>>> k = Kvadrat(100)
>>> k.opseg()
400
>>> k.povrsina()
10000
```

Pri pozivu metode `crtaj` moramo definirati vrijednost parametra `B`. Tako će poziv izgledati:

```
>>> k.crtaj('green')
```

a rezultirat će slikom 1.2.



Slika 1.2.

1.5. I objekti mogu biti parametri metoda

Parametar metode može biti i objekt te metoda može vraćati objekt.

Primjer 1.3. Kreirajmo klasu **`Razlomak`** te metodu **`umnozak()`**, koja će vraćati umnožak dviju svojih jedinki (dvaju razlomaka).

Klasa `Razlomak` imat će dva atributa: brojnik `b` i nazivnik `n`, pri čemu ćemo početnu vrijednost brojnika postaviti u 0, a nazivnika u 1.

Početna definicija klase `Razlomak` imat će sljedeći oblik:

```
class Razlomak:
    def __init__(self, b=0, n=1):
        self.b = b
        self.n = n
```

Kreirajmo dva objekta (dvije jedinke) klase `Razlomak` i nazovimo ih `a` i `b`. Želimo izračunati njihov umnožak `c`. Taj bi umnožak opet trebao biti objekt iz iste klase. U skladu s pravilima objektno usmjerenog programiranja objekt `a` treba pozvati metodu svoje klase koja će kao parametar imati objekt `b`, a kao rezultat trebamo dobiti objekt iste klase. To izgleda ovako: `c = a.umnozak(b)`.

Objekt `a` će u metodi klase `Razlomak` biti predstavljen varijablom `self` dok će objekt `b` postati parametar metode `umnozak()` koja mora biti definirana ovako:

```
    def umnozak(self, r):
        b = self.b * r.b
        n = self.n * r.n
```

```
        #metoda vraća objekt tipa Razlomak
        t = Razlomak(b, n)
        return t
```

Uočimo da metoda koja se nalazi unutar definicije klase kreira objekt (jedinku) te iste klase.

Ilustrirajmo upotrebu ove klase i definirane metode na jednom primjeru:

```
>>> a = Razlomak(3, 4)
>>> b = Razlomak(2, 5)
>>> c = a.umnozak(b)
>>> c.b
6
>>> c.n
20
```

Dobro bi bilo imati metodu koja će skratiti razlomak prije davanja konačnog rezultata:

Primjer 1.4. Klasi **Razlomak** dodajmo metodu **krati()** koja će razlomak skratiti. Iskoristimo je u metodi **umnozak()**.

Metoda `krati()` treba pronaći najveći zajednički djelitelj brojnika i nazivnika pripadnog objekta te brojnik i nazivnik podijeliti tim brojem:

```
def krati(self):
    b = self.b
    n = self.n
    while n > 0:
        b, n = n, b % n
    self.b //= b
    self.n //= b
```

Na kraju ćemo ovu metodu u metodi `umnozak()` upotrijebiti tako da ispred naredbe `return` dodamo još naredbu `t.krati()`:

```
def umnozak(self, r):
    b = self.b * r.b
    n = self.n * r.n
    t = Razlomak(b, n)
    t.krati()
    return t
```

Metodu `krati` mogli bismo dodati u konstruktor klase tako da se prilikom kreiranja objekta klase `Razlomak` pripadni razlomak odmah i skrati. U tom bi slučaju konstruktor imao sljedeći oblik: