1. Osnove objektno usmjerenog pogramiranja

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 potiče. Primjerice, u trgovinama se na blagajnama izdavanje računa obavlja računalom. 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 narudžba 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 slično. 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 igračke, stvarima u okolini itd. Objekti su prepoznatljivi po boji, obliku, zvuku, okusu i sl. – 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.

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

Primjerice, ako se radi o trgovini s pametnim mobitelima, tada se svi objekti – mobiteli – mogu svrstati u jednu **klasu**, nazovimo je: Mobitel. 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 mobitela 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 je prvi 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 za kao osnova za programiranje u bilo kojem objektno usmjerenom jeziku.

Konačno, zadnji korak je 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)`.[1]

Prisjetimo se djelovanja tih operatora, funkcija i metoda s nekoliko primjera u interaktivnom sučelju.

1 Vidjeti: Rješavanje problema programiranjem u *Pythonu* (RPPuP), odjeljak 7.2.

Prvo pogledajmo kako djeluju operatori:

```
>>> 'abc' + 'def'
'abcdef'
>>> 'abc' * 3
'abcabcabc'
>>> 'ab' in 'abcd'
True
>>> 'ab' not in 'abcd'
False
```

Evo i primjera uporabe nekih metoda:

```
>>> s1 = 'abcdef'
>>> print(s1.upper())
ABCDEF
>>> s2 = s1.upper()
>>> s2 == s1
False
>>> s2
'ABCDEF'
>>> s1.index('d')
3
```

Kao što možemo primijetiti 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 s 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 s dvjema donjim crticama čime je označeno da su to specijalne metode.

I osnovni brojčani tipovi podataka `int` i `float` su također 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
```

Na sljedećim primjerima može su uočiti razlika između metoda `__truediv__()` i `__floordiv__()`, odnosno operatora `/` i `//`. Prisjetimo se da operatori `%` i `//` djeluju i nad varijablama tipa `float`.

```
>>> c = 15.
>>> d = 2.15
>>> c.__truediv__(d)
6.976744186046512
>>> c.__floordiv__(d)
6.0
>>> c.__mod__(d)
2.1000000000000005
>>> _.__truediv__(d)
0.9767441860465119
```

Dakle, svi su tipovi podataka u *Pythonu* klase (engl. *class*), a sve pojedine vrijednosti su 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#. Skoro se svi veliki programski sustavi razvijeni u svijetu zasnivaju na programiranju uporabom klasa.

Svaka klasa ima svoj naziv. U skladu s dogovorom u *Pythonu* naziv klase piše se s 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:

```
class ImeKlase:
    definicija klase
```

Definicija klase obuhvaća definicije metoda koje su nad klasom definirane. 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 je postavljanje 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. Prisjetimo se da se metode klase pozivaju tako da se napiše prvi parametar, 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 krug te kreirajmo klasu `Krug` s pripadnim atributima i metodama.

Rješenje:

Za atribute kruga u ovom slučaju uzet ćemo samo duljinu polumjera kruga (`R`). Klasa će imati metode `opseg()` za računanje opsega pripadne kružnice i `povrsina()` za računanje površine kruga.

Definicija klase imat će u tom slučaju sljedeći oblik:

```
class Krug:
    def __init__(self, r = 0):
        self.R = r
        return
```

```
    def opseg(self):
        return 2 * self.R * 3.14159

    def povrsina(self):
        return self.R ** 2 * 3.14159
```

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 = naziv_klase(parametri)
```

Vrijednosti objekta možemo dobaviti tako da napišemo:

```
naziv_objekta.naziv_elementa
```

U našem bi primjeru klase `Krug` definiranje objekta s imenom `k` i s polumjerom vrijednosti 3 izgledalo ovako:

```
>>> k = Krug(3)
```

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

```
>>> k.R
3
```

Za kreirani objekt metode će dati sljedeće rezultate:

```
>>> k.opseg()
18.849539999999998
>>> k.povrsina()
28.27431
```

Vrijednosti atributa je moguće i naknadno mijenjati:

```
>>> k.R = 4
>>> k.R
4
>>> t.opseg()
25.13272
>>> t.povrsina()
50.26544
```

Vratimo se sada na samu definiciju klase. Konstruktor klase, kao što smo rekli, metoda je čije je ime `__init__()`. Parametar `r` u ovoj metodi je definiran kao opcijski parametar s inicijalnom

vrijednošću 0. Primijetimo se da se opcijski parametri mogu izostaviti pri pozivu funkcije te da će tada njihove vrijednost biti jednake odabranim opcijskim vrijednostima. Jednako to vrijedi i za metode. Prema tome, atribut `R` koji je u metodi `__init__()` određen sa `self.R = r` poprimit će vrijednost 0 ako pri kreiranju izostavimo parametar.

```
>>> k = Krug()
>>> k.R
0
>>> t.opseg()
0.0
```

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

```
>>> k.R = 3
>>> t.opseg()
18.849539999999998
```

Na prvi pogled moglo bi nas iznenaditi što metode za računanje opsega i površine nemaju parametar *r* (duljina polumjera kružnice). 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 `Krug()` preko konstruktora definirali vrijednost atributa `self.R`. U definiciji metode `opseg()` za računanje opsega piše: `2 * self.R * 3.14159`. Kada objekt `k` koristi metodu, naziv `self` bit će zamijenjen nazivom `k` (primjerice u izrazu `k.opseg()` će `self` predstavljati 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.R = 4`.

1.4. Metode s dodatnim parametrima

Ponekad ćemo kod nekih metoda imati i neke parametre koji nisu parametri klase (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 Krug dodajmo još jednu metodu koja će crtati pripadni krug polumjera R čije će središte biti u ishodištu kornjačinog koordinatnog sustava. Krug treba biti obojen nekom zadanom bojom B.

Rješenje:

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

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

```
def crtaj(self, B):
    pu()
    fd(self.R)
    lt(90)
    pd()
    color(B)
    begin_fill()
    circle(self.R)
    end_fill()
    ht()
    mainloop()
    return
```

Primijetimo da smo u metodi rabili parametar B bez prefiksa self, jer se radilo o lokanom parametru metode, dok smo uz parametar R, 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 Krug:

```
>>> k = Krug(50)
>>> k.opseg()
314.159
>>> k.povrsina()
7853.974999999999
```

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

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

a rezultirat će slikom 1.1:

Slika 1.1.

1.5. I objekti mogu biti parametri metoda

U prethodnom jednostavnom primjeru, metodu smo pozivali jednim jednostavnim parametrom. Mogu se osmisliti metode koje kao parametar imaju objekt i koje mogu vraćati objekt. Pogledat ćemo to na primjeru množenja dvaju razlomaka.

Primjer **1.3.**

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

Rješenje:

Klasa `Razlomak` će imati 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
        return
```

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 umnožak(), koja mora biti definirana ovako:

```
def umnozak(self, r):
    ##brojnik umnoška bit će jednak umnošku brojnika razlomka self
    ##i brojnika razlomaka r, a nazivnik, umnošku nazivnika razlomka self
    ##i nazivnika razlomka 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(1, 5)
>>> c = a.umnozak(b)
>>> c.b
3
>>> 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().

Rješenje:

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 b % n != 0:
        b, n = n, b % n
    t = n
    self.b //= t
    self.n //= t
    return
```

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 slučaju bi konstruktor imao sljedeći oblik:

```
def __init__(self, b = 0, n = 1):
    self.b = b
    self.n = n
    self.krati()
    return
```

1.5.1. Specijalne metode `__str__()` i `__repr__()`

Pri rješavanju raznih zadataka pojavit će nam se i potreba za izravnim ispisivanjem objekta. Primjerice, želimo ispis razlomaka u obliku brojnik/nazivnik. Pogledajmo što se ispisuje kada ispisujemo neki string.

```
>>> s = 'ABECEDA'
>>> s
'ABECEDA'
>>> print(s)
ABECEDA
```

Pokušamo li ispisati razlomak, u interaktivnom ćemo sučelju dobiti sljedeći izgled ispisa:

```
>>> a = Razlomak(3, 4)
>>> b = Razlomak(1, 4)
>>> c = a.umnozak(b)
>>> c
<__main__.Razlomak object at 0x00000000032FB748>
>>> print(c)
<__main__.Razlomak object at 0x00000000032FB748>
>>>
```

Kao što vidimo, niti jedan ispis objekta tipa `Razlomak` nije ono što smo htjeli. Problem bismo mogli riješiti na način da definiramo metodu `ispis()`, koja bi vraćala formatirani ispis razlomka. Ta metoda imala bi sljedeći oblik:

```
def ispis(self):
    if self.n == 1:
        return '{}'.format(self.b)
```

```
    else:
        return '{0} / {1}'.format(self.b, self.n)
```

Sada ćemo objekt tipa razlomak moći ispisivati na sljedeći način:

```
>>> a = Razlomak(3, 4)
>>> a.ispis()
'3 / 4'
>>> print(a.ispis())
3 / 4
>>>
```

Ovakav način ispisa je ono što smo htjeli, no nije elegantan. Problem ćemo riješiti s pomoću dviju specijalnih metoda: `__str__` i `__repr__`. Metoda `__str__` vraća tekstualni prikaz objekta kakav će se ispisivati pozivom tog objekta unutar funkcije `print()`, dok metoda `__repr__` vraća prikaz objekta kakav će se ispisati kada u interaktivnom sučelju navedemo samo ime objekta. Očito će obje ove metode u našem slučaju vraćati isti string, onaj koji vraća metoda `ispis()`. Definirajmo sada ove dvije metode:

```
def __str__(self):
    if self.n == 1:
        return '{}'.format(self.b)
    else:
        return '{0} / {1}'.format(self.b, self.n)

def __repr__(self):
    return self.__str__()
```

Kada smo definirali ove metode, objekte ćemo moći prikazivati i na sljedeći način:

```
>>> a = Razlomak(3, 4)
>>> a
3 / 4
>>> print(a)
3 / 4
```

Primijetimo da smo u metodi `__repr__()` pozvali metodu `__str__()` i to na način da smo ispred imena metode stavili prefiks self., baš kao što to radimo i kod korištenja svojstava.

Ako je prikaz objekta pri ispisu funkcijom `print()` isti kao prikaz navođenjem imena varijable u interaktivnom sučelju, nije potrebno posebno definirati metodu `__str__()`. U tom slučaju će se metoda `__repr__()` koristiti i umjesto metode `__str__()`, dok obrnuta situacija ne vrijedi. Stoga ćemo u nastavku kreirati samo metodu `__repr__()`.

Već smo pokazali da postoje i neke druge specijalne metode. Najčešće korištene metode za aritmetičke, relacijske i logičke operacije opisane su u sljedećoj tablici:

Naziv metode	Opis
`__add__(self, b)`	zbrajanje (+)
`__sub__(self, b)`	oduzimanje (-)
`__mul__(self, b)`	množenje (*)
`__floordiv__(self, b)`	cjelobrojno dijeljenje (//)
`__truediv__(self, b)`	dijeljenje (/)
`__mod__(self, b)`	ostatak cjelobrojnog dijeljenja (%)
`__pow__(self, n)`	potenciranje (**)
`__iadd__(self, t)`	+=
`__isub__(self, t)`	-=
`__imul__(self, t)`	*=
`__itruediv__(self, t)`	/=
`__ifloordiv__(self, t)`	//=
`__imod__(self, t)`	%=
`__lt__(self, b)`	manje (<)
`__le__(self, b)`	manje ili jednako (<=)
`__gt__(self, b)`	veće (>)
`__ge__(self, b)`	veće ili jednako (>=)
`__eq__(self, b)`	jednako (==)
`__ne__(self, b)`	različito (!=)
`__and__(self, b)`	logički I (and)
`__or__(self, b)`	logički ILI (or)

Tablica 1.1. *predefinirane metode za neke standardne aritmetičke, relacijske i logičke operacije*

Učinimo klasu `R` elegantnijom na način da preimenujemo metodu `umnozak()` u `__mul__()` i u tom slučaju ćemo dva objekta klase `Razlomak` množiti na isti način kao što bismo primjerice množili dva prirodna broja (koristeći operaciju *). Sada će klasa `Razlomak` imati oblik:

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

    def __mul__(self, r):
        b = self.b * r.b
        n = self.n * r.n
        return Razlomak(b, n)
```

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

    def __str__(self):
        if self.n == 1:
            return '{}'.format(self.b)
        else:
            return '{0} / {1}'.format(self.b, self.n)

    def __repr__(self):
            return self.__str__()
```

Sada dva razlomka možemo pomnožiti na sljedeći način:

```
>>> a = Razlomak(1, 2)
>>> b = Razlomak(2, 3)
>>> c = a * b
>>> c
1 / 3
>>> a * b
1 / 3
>>>
```

Primijetimo da smo kod izvođenja naredbe `a * b` zapravo osim metode `__mul__()` izveli i metodu `__repr__()` koja je odmah prikazala tekstualni prikaz objekta.

Na analogan način možemo definirati i ostale operacije (+, -, /, <, >, <=, >=, ==, !=,...). U nastavku ćemo implementirati operacije +, < i ==:

```
def __add__(self, r):
    b = self.b * r.n + self.n * r.b
    n = self.n * r.n
    return Razlomak(b, n)

def __lt__(self, r):
    return self.b * r.n < self.n * r.b

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