8. Culture Shock Festival

Radionica - "Uvod u programiranje s Pythonom"

Ivan Radiček[1]
Andrej Dundović[2]

20. ožujka 2012.

Sažetak

Ova radionica u tri dijela namijenjena je početnicima u programiranju, ali i onima koji su već programirali, a žele naučiti programski jezik Python.

Python je danas vrlo raširen zbog svoje jednostavnosti, fleksibilnosti, široke primjenjivosti i neograničavajuće licence. Python se može iskoristiti za izradu računalnih igara, web aplikacija, raznih proširenja drugih programa, kao skriptni jezik, ali ga se često može naći u znanosti i inženjerstvu gdje obavlja numeričke simulacije.

Ukratko, ako se netko želi početi baviti programiranjem bez velikog teorijskog uvoda i razumijevanja internog funkcioniranja računala, najbolje je da krene od Pythona. Ova je radionica zamišljena upravo da to omogući. Oni iskusniji možda neće puno naučiti, ali i oni su dobrodošli na druženje i razmjenu iskustva.

Radionica će se održati u tri termina po tri sata u Klubu Kulture Križevci, a obradit će se osnove programiranja (kontrola toka programa, osnovni tipovi podataka, stringovi, liste i rječnici) te uvod u objektno orijentirano programiranje dok bi se u drugom dijelu radionice realizirali praktični programi uz pomoć Pygamea. Radionice će biti popraćene i domaćim zadaćama za polaznike.

[1]apsolvent računarstva na FER-u u Zagrebu, ivan@radicek.com
[2]apsolvent fizike na PMF-u u Zagrebu, andrej@dundovic.com.hr

Sadržaj

1 Prva radionica **3**

1.1 Uvod i motivacija . . . 3

1.1.1 Što je programiranje? . . . 3

1.1.2 Zašto programirati u Pythonu? . . . 3

1.1.3 Gdje se sve Python koristi? . . . 4

1.2 Prvi koraci u Pythonu . . . 4

1.3 Zadaća . . . 6

1.3.1 Prvi zadatak . . . 6

1.3.2 Drugi zadatak . . . 7

1.3.3 Treći zadatak . . . 7

2 Druga radionica **8**

2.1 Sustav kontrole verzija - Git . . . 8

2.1.1 Korištenje Gita . . . 9

2.2 Osnove objektno orijentiranog programiranja . . . 10

2.2.1 Što je OOP? . . . 10

2.2.2 Programske paradigme . . . 11

2.2.3 Objekt . . . 12

2.2.4 Nasljeđivanje . . . 14

2.3 Igre u Pythonu - Pygame . . . 14

2.3.1 Logika grafičkih programa . . . 15

2.3.2 Minimalni program u Pygameu . . . 15

2.3.3 The igra . . . 18

2.4 Zadaća . . . 18

2.4.1 Prvi zadatak . . . 18

2.4.2 Drugi zadatak . . . 20

2.4.3 Treći zadatak . . . 21

1 Prva radionica

Cilj je prve radionice ukratko motivirati zašto uopće programirati, a onda i zašto to raditi upravo u Pythonu te ostali dio vremena posvetiti objašnjavanju osnova i sintakse Pythona.

1.1 Uvod i motivacija

1.1.1 Što je programiranje?

Uvod je napravljen po tekstu o programiranju s Wikipedije [1] i transkriptu kratkog govora Larryja Walla [2] o programiranju.

1.1.2 Zašto programirati u Pythonu?

(TODO: napraviti sažetak prednosti [3])

Filozofija Pythona sažeta je na humorističan način u Zenu Pythona [4]:

> The Zen of Python
>
> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one– and preferably only one –obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than *right* now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea – let's do more of those!

1.1.3 Gdje se sve Python koristi?

Python se, zbog svoje proširivosti i zbog mnoštva gotovih modula, može koristiti u bilo kojem području - doslovno.[1]

U slobodnom softveru, Python se nalazi na svakom koraku - u nekim distribucijama GNU/Linuxa većina je sistemskih alata napisana u Pythonu (Fedora, CentOS), koristi se kao jezik za pisanje web aplikacija (npr. Django, reddit, MoinMoin), u znanosti i inženjerstvu obavlja numeričke proračune (biblioteke numpy i scipy) itd.

Python se koristi i u komercijalnim igrama, npr. Sid Meier's Civilization IV (2005), EVE Online (2003), Freedom Force (2002) i drugima.[2]

Međutim teško je reći gdje se sve još Python može naći jer nitko nema obavezu objaviti da je nešto napisao u Pythonu, no tu i tamo procuri informacija o primjeni Pythona u potpuno neočekivanim slučajevima, kao npr. da Industrial Light & Magic koristi Python.[3]

1.2 Prvi koraci u Pythonu

Preporuka besplatno dostupnih knjiga za učenje Pythona [5, 6]. Brzi tečaj kroz interaktivni Python na webu: https://try-python.appspot.com/.

Sintaksa u natuknicama:

1. aritmetika (posebno naglasiti razliku između floata i inta)
2. stringovi (igranje sa stringovima, pretraživanje - find)
3. varijable (čemu služe, imenovanje, načini imenovanja)
4. liste (što su, čemu služe, opis dohvaćanja podlisti - ':' notacija)
5. grananje - if, else, elif
6. petlje - for (objasniti funkcije: xrange, len koje su korisne kod lista), while
7. funkcije (što su, čemu služe, imenovanje i sl)

Praktičan je zadatak na radionici bio pronaći najkraću riječ u rečenici. Riječ je definirana kao rečenica razdvojena razmacima. Primjer interakcije programa:

[1] Pythonova zajednica održava takve popise primjene ovdje http://www.python.org/about/apps/.

[2] Lista igara napravljenih dijelom ili u cijelosti u Pythonu može se naći ovdje http://wiki.python.org/moin/PythonGames.

[3] Industrial Light & Magic Runs on Python, http://www.python.org/about/success/ilm/.

```
Unesite recenicu> Ovo je recenica
Najkraca rijec je: je

Unesite recenicu> Jako jako dugacka recenia, bla.
Najkraca rijec je: Jako

Unesite recenicu> Jako, jako dugacka recenica, bla
Najkraca rijec je: bla
```

Jedno od rješenja navedenog zadatka[4]:

[4]Izvorni je kod dostupan na adresi http://ur1.ca/8jsaj

```
def najkraca_rijec(recenica):
    """ Vraca najkracu rijec iz recenice. """

    # Razdvojimo recenicu po razmacima u rijeci
    rijeci = recenica.split(" ")

    # Pamtimo najkracu - na pocetku je nemamo
    najkraca = None

    # Pregledamo sve rijeci
    for rijec in rijeci:

        # Ukoliko nemamo jos rijec ili je trenutna kraca od
        # do sada najbolje - zapamtimo je
        if najkraca is None or len(najkraca) > len(rijec):
            najkraca = rijec

    # Vratimo najkracu
    return najkraca


def main():

    # Trazimo recenicu od korisnika
    recenica = raw_input("Unesite recenicu> ")

    # Nadjemo i ispisemo najkracu rijec
    print "Najkraca rijec je:", najkraca_rijec(recenica)


if __name__ == "__main__":
    main()
```

1.3 Zadaća

1.3.1 Prvi zadatak

Definirajte funckciju 'print_round' koja za ulazni parametar (float) ispisuje najbliži cijeli broj. U implementaciju funkcije dozvoljeno je koristiti:

- funkciju 'str' koja pretvara bilo koji objekt (npr. float) u string (doc:str)

- funkciju 'int' koja pretvara objekt u cijeli broj (doc:int)
- metodu .find() nad stringovima koja vraća prvu poziciju pojavljivanja nekog znaka (doc:str.find)
- izvlačenje podstringa (podsjetnik: s[i:j] dio s-a od i do j)

Primjer izvođenja:

```
>> print_round(100)
100
>> print_round(100.1)
100
>> print_round(100.5)
101
>> print_round(100.9)
101
>> print_round(101.001)
101
```

1.3.2 Drugi zadatak

Napišite funkciju koja će učitati tekst iz tekstualne datoteke čiju putanju[5] će primiti kao argument. Funkcija će vratiti broj svih riječi u tekstu. Riječ je definirana kao niz znakova odvojen razmakom ili novim redom.

Hint: zgodno je iskoristiti rješenje zadatka s prve radionice.

Datoteka se otvara s funkcijom 'open', npr. za čitanje ('r')

```
dat = open('datoteka.txt', 'r')
```

a dobro je u petlji iskoristiti metodu 'readlines()' (doc:readlines).

1.3.3 Treći zadatak

Napišite funkciju *sudar* koja prima dvije dvojke brojeva (x_1, y_1), (x_2, y_2), te dva broja r_1 i r_2. Dvojke su koordinate središta krugova, dok su brojevi r_1 i r_2 radijusi tih krugova. Funkcija vraća *True* ako se krugovi preklapaju ili *False* ako se uopće ne dodiruju.

Hint: Udaljenost između dvije točke računa se formulom: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Funkcija korijen (*sqrt*) nalazi se u modulu *math* (doc:math). Modul se učitava i koristi ovako:

```
import math as m # ucitava modul math s nazivom imenskog prostora ''m''

print m.sqrt(2) # ispisuje 1.4142135623730951
```

[5] Putanja je dana u obliku stringa, npr. '/moja/datoteka/je/tu.txt'.

2 Druga radionica

U drugoj radionici ideja je uvesti polaznike u sustav kontrole verzija koji će olakšati izradu zajedničkog projekta, objasniti načela objektno orijentiranog programiranja kojim je prožet cijeli Python te započeti Pygame - skup modula za Python koji omogućuju relativno brzo i jednostavno stvaranje računalnih igara s grafičkim sučeljem.

2.1 Sustav kontrole verzija - Git

Pri pisanju dokumenata ili programskog kôda, a pogotovo kad u tom procesu sudjeluje više ljudi, čest je problem snaći se u različitim verzijama koje nastaju. Npr. naći najnoviju verziju koja sadrži sve do tad učinjene promjene ili naći prijašnju verziju, prije nego je uvedena neka promjena koja se pokazala krivom ili suvišnom. Zgodno je tada uvesti konzistentno imenovanje datoteka po verzijama i po autoru (npr. v.1, v.2, ili v.1-Petar, v.2-Mate) i pri svakoj promjeni tada promijeniti brojač verzije za jedan na više. Takva sistematizacija zahtjeva dosta discipline od autora, ali i dalje ne rješava sve probleme (npr. kod koga naći najnoviju reviziju koja sadrži sve prethodne?).

Rješenje dolazi u obliku **sustava za kontrolu verzija** (VCS[6]). Takav se sustav obično postavi na neko centralno, svima mrežno dostupno mjesto (repozitorij) te tada sustav brine o tome koja verzija je najnovija pamteći uz to cijelu povijest promjena. Svaku promjenu na datoteci koju učini, autor šalje u centralni repozitorij i provjerava jel netko već u međuvremenu modificirao istu datoteku, ako nije - šalje svoju verziju koja postaje najnovija i svima dostupna, ako je - autor rješava koliziju u dogovoru s drugima. Najpoznatiji su takvi sustavi kontrole verzija CVS[7] i SVN[8].

Nedostatak je takvih sustava što su centralizirani čime je moguće evidentirati reviziju samo u slučaju veze na glavni poslužitelj, a osim toga nemoguće je koristiti konrolu verzija za lokalni razvoj bez "obavještavanja" svih drugih sudionika. Te nedostatke popravljaju **distributivni** sustavi kontrole revizija. Najpoznatiji su Git[9] i Bazaar[10].

Git je razvio Linus Torvlads, začetnik Linuxa[11], za potrebe razvoja Linuxa nakon što je prijašnje rješenje zakazalo.

[6] Krat. en. *version control system*.

[7] Krat. en. *Concurrent Versions System*, službene stranice http://savannah.nongnu.org/projects/cvs.

[8] Krat. en. *Apache Subversion*, službene stranice http://subversion.apache.org/, SVN je naljednik CVS-a u koji je popravio sve mane prethodnika, ali i dalje ostao s njim kompatibilan.

[9] Git koriste mnogi projekti slobodnog softvera, od Linuxa i Androida preko grafičkih okružanja KDE i Gnome pa sve do gomile ostalog softvera (Git, X.org, Eclipse, PostgreSQL, Qt, ...); službene stranice http://git-scm.com.

[10] Bazaar je u cijelosti napisan u Pythonu, službene stranice http://bazaar.canonical.com/en/.

[11] Samo "Linux" naziv je jezgre (en. *kernel*) popularnih operacijskih sustava kao Ubuntu, Fedora, Android i dr., dok se GNU/Linux koristi kao naziv cijelog operacijskog sustava temeljenog na Linuxu i slobodnom softveru pa su tako Ubuntu, Fedora i dr. distribucije GNU/Linuxa.

Prednosti su Gita distributivnost, brzina, dobra podrška za nelinearni razvoj, sveprisutnost i postojanje besplatnih servisa koji nude *hosting* Git repozitorija.

Poznatiji su servisi za *hosting*, mega popularan, GitHub i slobodan, Gitorious. Slobodna licenca garantira dostupnost kôda Gitoriousa što omogućuje postavljanje istoga na vlastiti poslužitelj sa svim pratećim mogućnostima ukoliko se za to pokaže potreba.

Na ovoj radionici koristit će se javni repozitorij na Gitoriousu koji se nalazi na linku `https://gitorious.org/cshck-python/`. Na danom linku upravo Gitorious omogućuje pregledavanje sadržaja repozitorija kroz web sučelje, praćenje aktivnosti suradnika, uređivanje meta podataka o projektu itd.

2.1.1 Korištenje Gita

Git postoji za sve popularne i ne tako popularne platforme i jednostavno se instalira na svakoj od njih stoga nije potrebno to posebno opisivati. Svi primjeri ovdje napravljeni na repozitoriju Radionice radit će i s drugim repozitorijima, no za dodatne informacije i mogućnosti Gita dobro je pogledati dokumentaciju [7].

Za stvaranje lokalne kopije javnog repozitorija nije potrebna registracija, već ga je moguće jednostavno klonirati naredbom **clone**:

```
git clone git://gitorious.org/cshck-python/cshck-python.git cshck-python
```

što će u trenutnom direktoriju stvoriti mapu imena *cshck-python* u kojoj se nalazi sav sadržaj repozitorija pretraživ standardnim upraviteljima datoteka. Git istovremeno stvori skrivenu mapu *.git* unutar *cshck-python* u koju pohranjuje cijelu povijest revizija i druge meta podatke što omogućuje neometan lokalni rad na repozitoriju. Svaku naknadnu promjenu u glavnom repozitoriju nakon kloniranja moguće je povući s naredbom **git pull**.

U tom lokalnom direktoriju moguće je sada nastaviti normalno mijenjati datoteke i dodavati nove. Upisivanjem **git status** Git ispisuje sve datoteke nad kojima je napravljena neka promjena u odnosu original ili na zadnju snimku stanja.

Ako se želi napraviti snimka trenutnog stanja kao revizije na koju se uvijek moguće vratiti, prvi je put potrebno konfigurirati Git unoseći podatke o korisniku na ovaj način:

```
git config --global user.name "korisničko_ime"
git config --global user.email "e-mail@adresa.com"
```

a kako bi promjene bilo moguće poslati na Gitorious i tako objaviti u glavnom repozitoriju, ti podaci trebaju odgovarati onima unesenim pri registraciji na Gitorious.[12]

Tek nakon postavljanja korisničkog imena i e-mail adrese moguće je napraviti snimku trenutnog stanja, odnosno **commit**:

```
git commit -a -m "Kratka informacija o napravljenoj promjeni"
```

[12] Registracija na Gitorious može se provesti na `https://gitorious.org/users/new`.

gdje **-a** označava opciju da se spreme sve promijenjene datoteke koje su u sustavu gita, a **-m** označava opciju za dodavanje poruke o kakvoj je promjeni riječ. Dodavanje nove datoteke u sustav kontrole verzije radi se naredbom **git add ime_datoteke1 ime_datoteke2 ime_datoteke3 (...)** nakon čega se opet treba snimiti novonastalo stanje s **commit**.

Slanje promjena u javni repozitorij zahtjeva malo više koraka i to prvenstveno radi sigurnosnih razloga.

Prvo je potrebno generirati par ključeva (javni i tajni) naredbom u GNU/Linuxu **ssh-keygen**, ako već nisu generirani za neke druge svrhe. Tada treba prekopirati javni ključ koji se obično nalazi u direktoiju */home/username/.ssh/id_rsa.pub* u profil na Gitoriousu pod "SSH keys". Time je napravljen preduvjet za sigurnu komunikaciju i mogućnost autentifikacije na Gitorious kroz Git.

Drugi je korak javljanje voditeljima radionice (npr. e-mailom) korisničkog imena s kojeg će se slati promjene u javni repozitorij kako bi ga mogli odobriti, dok je treći korak dodavanje tog javnog repozitorija u već postojeći lokalni:

```
git remote set-url --push origin git@gitorious.org:cshck-python/cshck-python.git
git push origin master
```

nakon čega je moguće naredbom **git push** poslati sve promjene (*commitove*) na javni repozitorij kako bi postali javno dostupni i kako bi ih mogli skinuti i ostali suradnici.

Osim kroz naredbeni redak sa Gitom je moguće intuitivno upravljati i kroz grafičke programe kao npr. Gitk, git-cola, Giggle, QGit i dr.

Ovime je završena kratka ekskurzija u Git i sustav kontrole verzija, a ono što će trebati zapamtiti za ovu radionicu samo su naredbe **pull**, **commit** i **push**, dok se za ostale mogućnosti i situacije dobro posavjetovati s literaturom o Gitu [7] ili šalabahterom [8].

2.2 Osnove objektno orijentiranog programiranja

2.2.1 Što je OOP?

"Object-oriented programming (OOP) is a programming paradigm using "objects" - data structures consisting of data fields and methods together with their interactions ... Programming techniques may include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance" [9]

Cilj nam je prikazati motivaciju za uvođenjem "nove" paradigme kroz probleme koji proizlaze iz postojeće imperativne (proceduralne) paradigme te nakon toga dati osnovni pregled principa i mogućnosti objektne paradigme.

2.2.2 Programske paradigme

Prvo je pitanje koje si možemo postaviti: što je to uopće programska paradigma? Najjednostavnije, programska je paradigma osnovni "stil" pisanja programa. Paradigme se razlikuju u konceptima i razini apstrakcije koja se koristi za osnovne elemente programa (kao što su objekti, funkcije, varijable, ograničenja, ...) i načine izračuna (pridruživanje, evaluacija, tokovi podataka, ...).

Neke od najpoznatijih (najpopularnijih) programskih paradigmi:

- Deklarativna (funkcijska, logička, ...)
- **Imperativna (proceduralna)**
- **Objektno orijentirana**
- Aspektno orijentirana
- Pokretano događajima
- Oko 60 različitih na Wikipediji

Najraširenija i najjednostavnije je imperativna/proceduralna pradigma. Sastoji se od niza naredbi koje "mijenjaju stanje". Osnovni koncept je procedura/rutina/metoda/funkcija/... - parametrizirani komad kôda koji ima povratnu vrijednost i može se pozvati sa bilo kojeg mjesta u programu.

Dalje ćemo kroz nekoliko primjera pokazati kako isti problem riješiti kroz različite dvije paradigme. Primjer kroz koji ćemo prolaziti je vrlo jednostavan: Za danu središnju točku i n drugih točaka, odrediti koja od n točaka je najbliža središnjoj.

U prvom primjeru (oop_neproc_1.py) je najjednostavnije rješenje problema kada imamo dvije točke. Točke su definirane sa šest varijabli (po dvije za svaku točku) te je svakoj od dvije nesredišnje točke izračunata udaljenost od središnje. Nakon toga program samo ispisuje koja točka je bliže (ili su točke podjednako udaljene). U drugom primjeru (oop_neproc_2.py) stvar je zakomplicirana dodavanjem treće nesredišnje točke. Rješenje ostaje slično, izračunava se i treća udaljenost s razlikom da se sada udaljenosti spremaju u listu i traži najveća.

Problem je uočljiv već sada: sa svakim povećavanjem broja točaka mi ponovo pišemo funkciju za izračun udaljenosti dvije točke (i time uvelike povećavamo prostor za grešku). Rješenje problema je vrlo jednostavno: definiranje funkcije koja izračunava udaljenost tako da ne moramo svaki puta formulu pisati ponovo. Takvo rješenje dano je u primjeru oop_proc_1.py Time smo upravo uveli apstrakciju koja je svojstvena **proceduralnoj / imperativnoj** paradigmi.

Očito nam paradigme mogu pomoći u savladavanju složenosti koda. Uvođenjem nove apstrakcije smanjili smo si prostor za pogrešku, ali i pojednostavili kod tako da je lakši za čitanje.

No, naš program i dalje ima mjesta za poboljšanja. Na primjer, jednu točku predstavljamo sa dvije varijable, iako je to jedan koncept za sebe. Tom problemu možemo lagano doskočiti tako da malo bolje strukturiramo naše podatke, te u jednu varijablu spremimo i x i y vrijednosti (pomoću n-torke). Upravo to je i napravljeno u primjeru oop_proc_2.py .

Proceduralni programi obično se sastoje od mnogo različitih struktura (kao što je naša točka) i funkcija koje djeluju nad njima (naša metoda *dist*). Za razliku od našeg programa sa jednom strukturom i funkcijom, program sa par stotinjaka struktura i funkcija nije lagano održavati. Strukture i funkcije koje nad njima djeluju povezane su samo time što je programer koji ih je napisao tako odlučio. Osim toga ne postoji ništa drugo što ih veže i to predstavlja problem za razumijevanje nekom drugom programeru koji će raditi na istom programu ili čak originalnom programeru nakon par mjeseci.

Još jedan od problema je što naša struktura točka u biti nije točka (osim, opet, u glavi originalnog programera) nego samo n-torka koja sadrži 2 vrijednosti.

Iako ovo nije jedini problem, dovoljan je da pogledamo kako objektnom paradigmom možemo poboljšati naš program.

2.2.3 Objekt

Osnovni koncept objektno orijentirane paradigme je **objekt**. Objekt je struktura koja sadrži podatke (stanje) i ponašanje (metode ili funkcije).

U našem programu, definirali smo 4 objekta koji svi imaju 2 podatka: x i y te jednu metodu: udaljenost od druge točke.

Kako naša 4 objekta imaju isto ponašanje i podatke kažemo da su to objekti iste **klase**. U stvari, u objektnom programiranju mi definiramo klase koje su predlošci iz kojih kreiramo pojedine konkretne objekte. Točku je definirana u primjeru oop_oo_1.py.

U Pythonu klasa se definira ključnom riječi *class* nakon slijedi ime klase, a u tijelu klase (uvučeno, naravno) definiraju se medote pripadne klase (po svemu iste kao i funkcije osim što im je prvi parametar obavezno **self**).

self (u stvari može se zvati bilo kako, bitno da je prvi parametar svake funkcije) je varijabla trenutnog objekta.

Instanciranje objekta (kreiranje po predlošku iz klase) radi se na način da pozovemo funkciju koja se zove isto kao i klasa (u našem slučaju *Point*). Kada kreiramo objekt Python u stvari pozove konstruktor (*__init__* metodu) dane klase. U našem primjeru konstruktor prima 2 dodatna parametra (x i y) - može ih biti proizvoljan broj (bitno je samo da postoji variajbla self kao prvi argument).

Primjer definiranja klase *Point* sa konstruktorom koji prima 2 paramtera i metodom dist koja račun udaljenost točke do neke druge točke (*other*) te instanciranja 2 objekta iste klase:

```
class Point:

    def __init__(self, x, y):
        """Constructs point with x and y"""
        self.x = x
        self.y = y

    def dist(self, other):
        """Returns distance between self and other point"""
        return math.sqrt((self.x - other.x)**2 + (self.y - other.y)**2)

point_a = Point(3, 5)
point_b = Point(2, 2)
```

Nad objektom možemo pozivati metode ili pristupati podacima. Sintaksa u Pythonu, ako i u brojnim drugim jezicima, je: *objekt.ime_funkcije()*. Ukoliko je t naš objekt instanciran iz klase *Point* tada nad njim možemo pozvati metodu *dist* na način *t.dist(t2)*, gdje je $t2$ neki drugi objekt klase **Point**.

U svim metodama klase/objekta pojavljuje se objekt *self* kojeg nismo posebno objasnili. *self* je referenca na sam objekt, što znači da kada u bilo kojoj metodi napišemo *self.x* dohvaćamo varijablu x tog objekta nad kojim smo pozvali metodu. To je najbolje vidljivo u pozivu metode *t.dist(t2)*. Kada u tijelu metode napišemo *self.x* tada je to varijabla x objekta t (onog nad kojim smo pozvali metodu), a kada napišemo *other.x* tada dohvaćamo varijablu prvog argumenta funkcije - u ovom slučaju *t2*.

U primjeru oop_oo_2.py dan je primjer sa 2 klase: *Rect* i *Square*. Klase predstavljaju pravokutnik i kvadrat i obje imaju metodu *area* koja računa površinu tog lika. Klase se razlikuju po podacima (kvadrat ima samo jednu stranicu, dok pravokutnik ima 2) i u implementaciji metode za izračun površine.

U primjeru vidimo da je instancirano nekoliko objekata od obje klase te je izračunata njihova ukupna površina. U primjeru vidimo da ne moramo brinuti koju metodu *area* (onu pravokutnika ili kvadrata) pozivamo te to umjesto nas radi Python na temelju vrste (klase) objekta.

Vrlo je vžano primjetiti da isti (ili dovoljno sličan) program možemo napisati i bez objekte paradigme, što je pokazano u primjeru oop_proc_3.py .

Iz zadnjeg primjera jasno je vidljivo koje prednosti smo dobili uvođenjem nove paradigme. Prvo, dobili smo konceptualni objekt koji predstavlja točku, kvadrat ili pravokutnik (a ne samo struktru sa 2 broja koja ne predstavlja ništa van glave originalnog programera). Drugo, imamo metode koje su vezane baš uz taj objekt. I treće možemo pozivati funkcije nad objektima bez da previše razmišljamo o tome koji je to stvarno objekt i koja stvarna metoda će se pozvati (o tome se za nas brine Pyton).

Isto tako je bitno znati da nam objekta paradigma neće riješiti sve probleme u programiranju i da svakako nije *sliver bullet* programiranja. Štoviše, objektna paradigma se vrlo često koristi u kombinaciji sa drugim paradigmama (naročito proceduralnom).

2.2.4 Nasljeđivanje

Nasljeđivanje je važna i moćna mogućnost objektno orijentirane paradigme, ali nije ključna za njezino razumjevanje te je navedena zasebno.

Ideja se temelji na iskorištavanju već postojećeg programskog kôda na način da pišemo klase tako u drugim klasama možemo to ponovo iskoristiti (odnosno naslijediti).

Pošto prepoznavanje nasljeđivanja već zadire u područje *objektnog dizajna*, ovdje ćemo dati samo osnovni primjer za razumijevanje principa.

U Pythonu klase nasljeđujemo tako da iza ključne riječi *class* i imena klasa, a prije dvotočke u zagradi navedemo klase koje želimo naslijediti. Kada klasa A nasljeđuje klasu B tada u klasi A imamo na raspolaganju sve podatke i metode već definirane u klasu B te ih najnormalnije možemo pozivati (i dohvaćati varijable) kao da su definirane u klasi B.

Primjer definiranja klase *Dog* koja nasljeđuje klasu *Animal* (re)definira metodu *voice*:

```
class Dog(Animal):


    def voice(self):
        print "Bark, bark"
```

U koliko u klasi A definiramo metodu istog imena kao i neka metoda koju smo naslijedili iz klase B tada nova metoda sakriva staru. To nam koristi kada u novoj klasi hoćemo redefinirati neko ponašanje iz klase koju smo naslijedili.

Kao što je prikazano u primjeru oop_oo_3.py nasljeđivanjem se obično modelira hirearhija, u ovome slučaju životinja. Životinja općeniti pojma dok su pas, mačka i zec vrste životinje i sl.

Nasljeđivanje može i ići i na više razina, a mogu se naslijediti i više od jedne klase, no to nam trenutno nije zanimljivo.

Ovdje navede nisu sve ideje niti mogućnosti koje pruža objekta paradigma (a ne postoji ni jasan standard, svaki jezik implementira ovu paradigmu na drugačiji način), ali bi, nadamo se, trebala biti dovoljna pogloga programiranju igare u *PyGameu*.

2.3 Igre u Pythonu - Pygame

Pygame je skup modula za Python koji omogućuju jednostavnu izgradnju igara s grafičkim sučeljem. Pygame koristi multiplatformski skup biblioteka SDL[13] za pristupanje multimedijal-

[13] Krat. en. *Simple DirectMedia Layer*, službene stranice `http://www.libsdl.org/`.

nom hardveru računala, ali donosi i neke nove funkcionalnosti da bi pojednostavio inače vrlo kompleksan razvoj grafičkih i multimedijalnih aplikacija. [10]

Nakon svladavanja Pythonove sintakse i objektno orijentirane paradigme programiranja, ništa više ne stoji na putu ka praktičnom programiranju s Pygameom i izradi igre. Doduše, prije početka potrebno je prvo dati neku ideju igre i postaviti cilj koji se želi ostvariti.

Na primjer, radi jednostavnosti, cilj je napraviti dvodimenzionalnu svemirsku arkadu u kojoj će igrač upravljati svemirski brod, uništavati neprijatelje i skupljati bodove. U kasnijim stadijima razvoja mogu se dodavati razno razni dodaci i proširenja. Svemirskim brodom igrač će upravljati pomoću tipkovnice, dok će planeti i drugi objekti biti prepreke za kretanje.

2.3.1 Logika grafičkih programa

Svaki grafički program sastoji se od beskonačne petlje koja se vrti sve dok se program izvršava, a unutar nje nalazi se sva logika koju korisnik poziva kroz interakciju s programom. Kod grafičkih programa prirodna je podjela kôda u tri dijela: dio koji se brine za grafički prikaz i iscrtavanje programa na ekranu (*View*), dio koji prati korisničke naredbe i akcije (*Controller*) te dio koji sadrži svu logiku odgovornu da na temelju korisničkog unosa, izračuna potreban rezultat i proslijedi ga dijelu za prikaz (*Model*). Takva raščlamba i arhitektura naziva se "*Model View Controller*" (MVC) [11] te se, osim u grafičkim/*desktop* aplikacijama koristi i kod web aplikacija. Ovakva separacija omogućuje lakšu kontrolu kôda i neovisni razvoj programske logike, korisničkog sučelja i prezentacijskog dijela. Prikazano u kôdu, glavna petlja uvijek će imati ove elemente:

```
while True:
        events()
        logic()
        render()
```

gdje će funkcija *events()* preuzeti "brigu" o korisničkom unosu, *render()* o iscrtavanju i prezentaciji, a *logic()* o programskoj logici.

2.3.2 Minimalni program u Pygameu

Prije nego se krene u kompleksniji kôd željene igre, dobro je vidjeti kako Pygame funkcionira na minimalnom primjeru ball.py[14].

Objašnjenje primjera:

[14] Primjer je preuzet iz originalnog Pygameovog vodiča uz par preinaka, http://www.pygame.org/docs/tut/intro/intro.html.

```
import pygame


# inicijalizacija pygame modula
pygame.init()
```

služi za učitavanje skupa modula pygame i njihovu inicijalizaciju;

```
# parametri igre
size = width, height = 640, 480
speed = [1, 1]
black = 0, 0, 0
```

ovi parametri obične su brojčane konstante koje će se iskoristiti u kôdu niže dolje;

```
# stvaranje prozora velicine size
screen = pygame.display.set_mode(size)
```

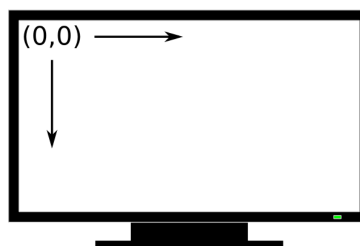
stvara objekt **screen** koji predstavlja glavni *Surface*, odnosno prozor u kojem se sve ostalo iscrtava, a kao argument uzima **size** (rezoluciju prozora);

```
# ucitavanje slike lopte
ball = pygame.image.load("../../resources/ball.png").convert_alpha()
```

prva naredba učitava sliku *ball.png* i konvertira je u interni Pygameov format radi bržeg iscrtavanja, a *alpha* verzija služi da sačuva transparentnost koju sadrži originalni format (png) što u konačnici daje novi objekt **ball** klase *Surface* (iste klase kao i **screen**);

```
# stvaranje objekta velicine slike
ballrect = ball.get_rect()
```

stvara novi objekt za spremanje koordinata "nevidljiog" pravokutnika koji omeđuje objekt **ball** i pozicionira ga na koordinate $(0, 0)$ što odgovara gornjem lijevom kutu prozora, slika (1);



Slika 1: Koordinatni sustav na ekranu kao i u prozoru Pygamea

```
run = True

# glavna petlja
while run:

    # kontroler
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False

    # programska logika
    ballrect = ballrect.move(speed)
    if ballrect.left < 0 or ballrect.right > width:
        speed[0] = -speed[0]
    if ballrect.top < 0 or ballrect.bottom > height:
        speed[1] = -speed[1]

    # render
    screen.fill(black)
    screen.blit(ball, ballrect)
    pygame.display.flip()
```

glavna je petlja samo vizualno podijeljena u tri dijela po ideji MVC-a:

- kontroler prolazi kroz sve događaje koje mu servira funkcija *event.get()* i ako je jedan od njih tipa *QUIT* postavlja varijablu **run** na *False* što uzrokuje prekidanje beskonačne petlje;
- programska logika u ovom programu pomoću metode *move* koja je definirana u objektu **ballrect** samo pomiče loptu za pomak u smjeru (x, y) koji je definiran listom od dva broja **speed** dok dvije provjere ispod služe samo za provjeru nalazi li se lopta još uvijek unutar vidljivog dijela prozora (ako se ne nalazi, obrne smjer brzine u smjeru x, odnosno y);
- crtački dio (render) prvo ispuni objekt *screen* crnom bojom[15], zatim crta objekt **ball** na objekt **screen** na koordinatama danim **ballrect-om** i na kraju sve te promjene iscrtava u prozoru s naredom *display.flip()*.

[15]Crna boja prethodno je definirana RGB-om - trojkom brojeva od kojih svaki predstavlja količinu pojedine osnovne boje od 0 do maksimalno 255, npr. (0,0,0) je crna, (0,0,255) je plava, a (255,255,255) je bijela boja.

2.3.3 The igra

(u izradi)

2.4 Zadaća

2.4.1 Prvi zadatak

Ideja je ovog zadatka modelirati svemirske brodove kroz OO paradigmu. Dana je osnovna klasa, SpaceShip (primjer dz_2_1_template.py):

```
class SpaceShip:
        def __init__(self, x, y, orient):
                """Creates space ship in location x, y and orientation orient
                and sets HP to initial value."""
                START_HP = 100

                self.x = x
                self.y = y
                self.orient = orient
                self.hp = START_HP

        def move(self, turn, distance):
                """Ship turns and moves by certain distance"""
                pass

        def damage(self, amount):
                """Damages ship by amount (decreases hp by amount)"""

                # Cannot go bellow 0
                self.hp = min(self.hp-amount, 0)

        def fire(self, other_ship):
                """Ship fires on other ship"""
                pass

        def colide(self, colide):
                """Detect colision between two ships"""
                pass
```

Brod je definiran u 2D svijetu sa x i y koordinatama te smjerom u kojem je okrenut. Osnovne metode broda su kretanje (računa se novi x i y na temelju starih vrijednosti parametara te

kuta za koji se okrene (*turn*) i udaljenosti kretanja *distance*). Nova se pozicija može računati sljedećim izrazima:

$$\Theta' = \Theta + \alpha$$

$$x' = x + cos(\Theta) \cdot r$$

$$y' = y + sin(\Theta) \cdot r$$

gdje je Θ trenutna orijentacija, x i y trenutna lokacija, α kut okretanja (*turn parametar*) te r udaljenost kretanja (*distance*). (*Napomena: kutevi se zadaju u radijanima*).

Brod može biti oštećen pri čemu mu se smanjuje *hp* parametar (to je već implementirano, ali može se redefinirati za npr. jače brodove koji imaju štit :))

Brod može pucati na neki drugi brod pri čemu ga vjerojatno oštećuje (možda npr. ovisno o udaljenosti).

Cilj je naslijediti osnovnu klasu broda i implementirati metode koje nisu implementirane (*Hint: imaju ključnu riječ pass u tijelu funkcije*). Naravno da možete, čak je i poželjno, redefinirati neke metode. Možete definirati i više razina nasljeđivanja.

Niže je dan primjer kako se može isprobati simulacija:

```
# Definiramo brod vrste Enterprise u sredistu svemira (:))
# okrenut za 90 stupjneva.
my_ship = Enterprise(0, 0, 90)

# Ispisujemo lokaciju
print "Enterprise", my_ship.x, my_ship.y

# Okrecemo brod za -90 stupnjeva i pomocemo ga
# za 100
my_ship.move(-math.pi/2., 100.0)

# Ponovo ispisujemo lokaciju
print "Enterprise", my_ship.x, my_ship.y

# Kreiramo neprijateljski broj vrste Cube
enemy_ship = Cube(0, 50)

# Ispisujempo njegov HP
print "Enemy", enemy_ship.hp

# Nas brod puca na neprijateljski
my_ship.firePhaseCanon(enemy_ship)

# Ispisujemo ponovo HP
print "Enemy", enemy_ship.hp
```

Ispis ovakvog programa (pod uvjetom da smo definirali gore navedene klase da dopuštaju ovakvo korištenje) mogao bit biti:

```
Enterprise 0 0
Enterprise 100 0
Enemy 1000
Enemy 0
```

Završna napomena: Unutar metode klase koja kao parametar prima drugi objekt **nemojte** direktno mijenjati varijable drugog objekta. Npr.: U metodi *fire* nemojte direktno smanjivati *hp* drugog broda koji je napadnut nego to izvedite kroz metode drugog broda.

2.4.2 Drugi zadatak

Potrebno je preurediti minimalni primjer dan za Pygame (*ball.py*) tako da se lopta pretvori u pravi objekt kojem se prilikom inicijalizacije postavlja početni položaj i početna brzina. Tada

se jednostavno može dodati nova, druga lopta:

```
lopta1 = Ball( pos1, speed1 )
lopta2 = Ball( pos2, speed2 )
```

Nakon toga potrebno je napraviti proširenje programa takvo da se te dvije lopte mogu sudariti (posljedica je sudara da se svaka lopta odbije na svoju stranu).

2.4.3 Treći zadatak

Cilj je zadatka preimenovati datoteku koja sadrži rješenje drugog zadatka iz ove zadaće u *dz2-ime_autora.py*, dodati je u sustav kontrole verzija unutar mape *sources/homework/* i poslati u javni repozitorij Radionice na Gitoriousu kako je opisano u tekstu uz pomoć Gita. Tada će svaki rješavač koji završi taj korak dobiti svoj vlastiti zadatak koji se direktno tiče izrade zacrtane igre.

Literatura

[1] Wikipedia.org, "Computer programming," Mar., 2012. http://en.wikipedia.org/wiki/Computer_programming.

[2] BigThink, "Larry Wall - Computer Programming in 5 Minutes," Aug., 2010. http://bigthink.com/ideas/21746.

[3] Wikipedia.org, "Python (programming language)," Mar., 2012. https://en.wikipedia.org/wiki/Python_(programming_language.

[4] T. Peters, "The Zen of Python," Aug., 2004. http://www.python.org/dev/peps/pep-0020.

[5] M. Pilgrim, *Dive into Python*. Apress, Berkeley, Calif., 2004. http://www.diveintopython.net/download/diveintopython-pdf-5.4.zip.

[6] A. B. Downey, J. Elkner, and C. Meyers, *How to Think Like a Computer Scientist: Learning with Python*. Green Tea Press, Wellesley, Massachusetts, 2nd edition ed., April, 2002. http://www.openbookproject.net/thinkcs/python/english2e.

[7] The Git Community, *Git Community Book*. http://book.git-scm.com.

[8] GitHub, "Git Cheat Sheets," Mar., 2012. http://help.github.com/git-cheat-sheets.

[9] Wikipedia.org, "Object-oriented programming," Mar., 2012. http://http//en.wikipedia.org/wiki/Object-oriented_programming.

[10] "Pygame - About," Mar., 2012. http://pygame.org/wiki/about.

[11] Wikipedia.org, "Model–view–controller," Mar., 2012. https://en.wikipedia.org/wiki/Model--view--controller.