Načini debagovanja u programskom jeziku Python

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Dimitrije Sekulić, Sandra Radojević, Maja Gavrilović, Matija Pejić
sekulic_dimitrije@yahoo.com, tetejesandra@gmail.com,
majamaj@live.com, matija.pejic@yahoo.com

18. april 2020

Sažetak

Debagovanje je postupak pronalaženja grešaka i rešavanja ostalih problema unutar programa. Shodno tome u ovom radu su obrađene različite tehnike debagovanja, od nekih opštih do tehnika specifičnih za *Python* kao programski jezik. Izložićemo njihove prednosti i mane, kao i prigodne uslove za njihovo korišćenje. Da bi se postigao optimalan rad softvera neophodno je ovladati tehnikama i alatima za brzo uočavanje propusta pri pisanju kôda. Takvim tehnikama štedimo vreme i učimo kako postati bolji programer.
Ključne reči: Python, debagovanje, debager, pdb, print, greška, PhyCharm

Sadržaj

1 Uvod

Greške pri pisanju kôda se svima nama dešavaju, bilo da smo početnici u programiranju ili programiramo već duži niz godina. Čak i najjednostavnije greške umeju da nas dovedu do frustracija u traženju i rešavanju istih. Svaki put kada program ne radi onako kako smo očekivali znamo da je došlo do greške, odnosno baga. Debagovanje je proces pronalaženja i rešavanja tih grešaka. Ono podrazumeva sledeće:

1. Znamo kako program treba da radi
2. Opažamo da je do baga došlo
3. Pronalazimo bag
4. Uklanjamo bag

Mi ćemo prikazati nekoliko opštih tehnika debagovanja i korišćenje debagera.

2 Osnovni koncepti debagovanja u Python-u

U Python-u postoji 47 različitih izuzetaka koji zajedno formiraju hijerarhiju izuzetaka [2]. Kada program izbaci izuzetak, tada znamo da je do greške sigurno došlo. *Ako je program kuća, izuzetak bi označavao da je požar u kući* [9]. Ako u programu dođe do izuzetka tada znamo da u programu ima bag. Razmotrićemo tri osnovne strategije za debagovanje izuzetaka:

- Čitanje kôda na mestu baga
- Razumevanje poruke o grešci
- Hvatanje izuzetaka

Najlakši izuzeci za debagovanje su *SyntaxError* i *IndentionError*. Oba predstavljaju slučaj u kome prevodilac ne zna da prevede kôd. Ovakvi bagovi mogu da budu česta pojava u Python-u zbog razlika između verzija *Python2* i *Python3*. Recimo, funkcija **print** nema istu sintaksu u ove dve verzije. Zbog toga se neki programi prevode sa verzijom 2, a sa verzijom 3 izbacuju sintaksne greške. Razmotrimo primer 1 u kome funkcija student treba da ispiše broj indeksa za zadato ime studenta.

```
def student(name):
    students = {
        'Pera': '107/2016',
        'Mika': '16/2016'
        'Laza': '252/2015'
    }

    print('Index of student Pera is ' + studenti[name])

student('Pera')
```

Primer 1: Funkcija student ispisuje broj indeksa za zadato ime studenta

Ovaj program ne uspeva da se prevede i izbacuje narednu grešku.

```
  File "primer.py", line 5
    'Laza': '252/2015'
          ^
SyntaxError: invalid syntax
```

Primer 2: Ispis iz konzole za primer 1

Python je izbacio *SyntaxError*, jer smo zaboravili zarez u liniji 4. Sintaksni analizator očekuje da su elementi u mapi razdvojeni zarezom, i kako to nije ispunjeno, izbačen je izuzetak. Slično, da smo posle dvotačke u prvoj liniji zaboravili da sledeća linija treba da bude nazubljena, program bi izbacio *IndentionError*.

Sintaksne greške su često uzrok brzog kucanja, prelaska sa nekog drugog jezika ili druge verzije jezika. Kod pojave ovakvih grešaka, najbolje je gledati liniju greške (ili liniju iznad nje), prebaciti deo programa u zaseban fajl, zatim proveriti uparenost zagrada i navodnika, proveriti da li je dobra verzija samog Python-a, a preporučuje se i korišćenje nekog naprednijeg editora [9]. Kao što smo već mogli da vidimo, kada u programu postoji sintaksna greška prevodilac izbacuje izuzetak i ispisuje **poruku o grešci**. Svaka poruka o grešci sadrži: **tip greške**, **opis greške** i **traceback**.

Tip greške jeste tip izuzetka koji je program izbacio. Svi izuzeci su podklase klase *Exception* u hijerarhiji izuzetaka [2].

Nakon tipa greške sledi opis greške koji nam kazuje šta se desilo. Ovaj opis može biti jako značajan i jasan, dok ponekad ne sadrži nikakvu korisnu informaciju. U primeru 2 tip greške je *SyntaxError*, a opis je *invalid syntax*.

Traceback sadrži informaciju o tome gde je program pukao. Ispisuju se segmenti programa koji sadrže grešku, broj linije gde je program pukao i niz funkcija koje su pozvane da bi program stigao do linije sa greškom. Neki izuzeci se ne mogu izbeći. Na primer, neminovno je da će nam biti izbačen *FileNotFoundError* ako učitavamo neku datoteku i unesemo loše putanju ili ta datoteka ne postoji. Na ovakve greške najbolje je reagovati **hvatanjem izuzetaka** unutar programa. To možemo da postignemo sa **try** i **except** blokom. Sa try pokušamo da pročitamo datoteku, a ako dođe do izuzetka, except blok će „uhvatiti“ taj izuzetak i na tom mestu reagovati, najčešće ispisom odgovarajuće poruke. Ako pritom koristimo neke resurse, možemo dodati finally deo gde ćemo ih korektno osloboditi.

Ono što treba izbegavati kod hvatanja izuzetaka jeste da u except bloku stavimo pass naredbu kojom nastavljamo dalje izvršavanje programa kao da do izuzetka nije ni došlo[9]. Kada se program prevede i ne izbacuje izuzetak, ali mi ne dobijamo željeni rezultat, kažemo da smo napravili **semantičku grešku**. Takve greške je obično teže debagovati, jer nemamo nikakvu informaciju od prevodioca da je do greške došlo. Jedina informacija koju imamo jeste pogrešan rezultat.

```
def suma(n):
    k = 0
    for i in range(n+1):
        k += i
    return k

print(suma(3)) # 6
```

Primer 3: Funkcija koja računa sumu prvih n brojeva

U primeru 3, program računa sumu prvih n brojeva. Neke od semantičkih grešaka koje su mogle da se dese su da range ide do n, umesto do n+1, da je umesto operatora += stavljeno samo =, zatim pogrešna inicijalizacija početne vrednosti za k, inicijalizacija unutar petlje umesto pre petlje itd [9].

Semantičke greške se obično lako debaguju u ovako prostim primerima. U kompleksijim primerima su teško uočljive i potrebne su neke od naprednijih tehnika za njihovo pronalaženje.

3 Naučni pristup debagovanju

U prethodnom delu, videli smo neke osnovne tehnike debagovanja. Ali šta ako nismo i dalje sigurni u čemu je problem? Šta ako naše nagađanje nije dovoljno dobro i ne znamo lokaciju problema, ni iz koda greške ni iz semantike koda? Tada se treba okrenuti formalnom načinu pronalaženja problema kao što je naučni metod. On traženje greške bazira na prikupljanju dokaza i predstavlja okvir (*eng. framework*) u koji se uklapaju ostale metode, kao i dobru bazu za kasnije testiranje i održavanje koda. Koraci su sledeći[9]:

1. Posmatraj: Počinjemo posmatranjem ponašanja programa
2. Napravi hipotezu: Posmatranjem dobijamo ideju, tj postavljamo hipotezu koja objašnjava ponašanje programa
3. Predvidi: Na osnovu hipoteze, pravimo predikciju šta bi drugo naš program trebalo da radi, pod uslovom da je hipoteza tačna
4. Testiraj: Ispitamo tu predikciju puštanjem programa u odgovarajućim eksperimentalnim uslovima i posmatramo rezultat izvršavanja
5. Zaključi: Zavisno od rezulatata ćemo prihvatiti ili odbaciti našu hipotezu. Ako smo odbacili hipotezu, vraćamo se na korak 2, gde postavljamo novu hipotezu ili refiniramo postojeću i sledimo dalje korake

Moć ovog metoda sastoji se u tome što on instinktivno nagađanje pretvara u formalnu i sistematičnu dedukciju. Njegovim pomnim praćenjem, dolazimo do pronalaženja i jako složenih i komplikovanih grešaka. Osim toga, dolazi se do čistijih rešenja i koda koji je lakši za održavanje.

Zašto onda ne koristimo uvek ovu metodu za debagovanje? [9] U praksi će se dešavati često da pravimo sitne, lako uočljive bagove, koji se nalaze u par minuta ako im se posvetimo. Korišćenje naučnog metoda pre nego što bar pokušamo neformalno da nađemo bag, ovde će nam doneti više štete nego koristi i oduzeti dragoceno vreme. Opšti savet je da ga primenimo ako ne nađemo rešenje u 10-15 minuta.

Da bi efikasno primenili ovaj način debagovanja, potrebno je da dobro vladamo tehnikama reprodukcije grešaka [9], automatizacije (pogotovo ako imamo složenije sisteme koji uključuju komunikaciju preko mreže) i izolacije grešaka(strip-down strategijom ili strategijom binarne pretrage) [9].

Strip-Down strategija odlikuje se iterativnim uprošćavanjem koda komentarisanjem ili uklanjanjem linija, dok ne dođemo do minimalnog broja linija potrebnog za reprodukcuju greške. Strategija binarne pretrage sastoji se iz modulacije koda na dva dela približno jednake veličine i proveravanja u koji deo se greška dalje propagira tokom izvršavanja. U tom delu se rekurzivno dalje nastavlja pretraga. Dobijene test skriptove je zgodno čuvati i za kasnije, jer oni mogu da se razviju u test funkcije[9].

4 Debagovanje print naredbama

Print je mnogim programerima metoda broj jedan za debagovanje. Razlog za to leži u lakoći korišćenja, relativno čestom pronalaženju greške i prostom prikazivanju nedostatka informacija o podacima i izvršavanju [9]. Iako jednostavan i nedvosmislen, to ne znači da je bez greške. Print se može previše koristiti i tako poremetiti ceo kod i njegovu čitljivost i eleganciju, pogotovo ako je kod duži. Da bi se to izbeglo, treba ga

disciplinovano koristiti sa već pomenutom binarnom pretragom i naučnom metodom.

Hipoteze koje tvrde da neki deo koda nije izvršen lako možemo odbaciti ako se izvrši print naredba nakon tog koda. Takođe, štampanjem vrednosti neke promenljive često možemo prihvatiti ili odbaciti hipotezu vezanu za njenu vrednost u određenom trenutku.

Ono što je takođe loše je to što mi dodajemo stvari koje naš program i ne treba da radi. Time u nekom smislu činimo kod više pogrešnim da bi ga popravili. *Zamislite pucanje i pravljenje rupa u zidu da bi proverili da li ima vatre u zgradi*[9].

Ako imamo složene strukture podataka u programu kao što su liste, rečnici, skupovi, torke, ili bilo koji tipovi podataka sačinjeni od prethodnih, možemo koristiti **pretty-printing** za njihov lepši ispis. Naime, u Python-ovoj standardnoj biblioteci postoji moduo zvani *pprint*[7]. Možemo ga koristiti da isforsiramo ispis u jednoj liniji, kao i da prilagodimo ispis našoj strukturi i podacima, zahvaljujući mnogobrojnim opcijama koje sadrži. Ovaj moduo možda nije direktno sredstvo debagovanja, ali olakšavajući čitljivost, umnogome ga olakšava.

4.1 Uključivanje i isključivanje print naredbi

Tokom debagovanja, dodali smo dosta linija u cilju dijagnoze koje kasnije treba obrisati radi čistoće koda. Međutim, ove linije mogu biti korisne kada želimo da ispitamo stanje programa na ovom mestu posle dodavanja novog koda, i zato njihovo brisanje predstavlja lošu ideju. Osim neefikasnosti ponovnog manuelnog dodavanja, problem je i što stalno pisanje i brisanje nosi rizik novih grešaka u kodu [9]. Zato mora da postoji neki način da uključimo i isključimo print naredbe u kodu.

Najprimitivnije rešenje je definisanje neke indikatorske promeljive koja se postavlja na true kad se debaguje i na false u suprotnom. Ovakav pristup dovodi do stavljanja kondicionih naredbi uz svaki print, čime se program usložnjava, usporava i postaje teško čitljiv. Alternativa je zameniti print naredbu sa **debug_print** naredbom koja se brine o proveravanju stanja neke DEBUG promenljive i, shodno tome, prosleđuje argumente regularnoj print funkciji.

```
def debug_print(*args):
    if DEBUG:
        print(*args)
```

Primer 4: Definisanje nove print funkcije

```
import sys

DEBUG = "-d" in sys.argv
```

Primer 5: Deklarisanje DEBUG promenljive

Ovakvom deklaracijom izbegavamo ručnu promenu vrednosti ove promenljive i pokrećemo program u modu za debagovanje samo dodavanjem opcije -d argumentima komandne linije.

Ovaj koncept se može proširiti Python bibliotekama **logging** [4] i **argparse** koje nam daju veću kontrolu nad štampanjem, uz mnogo novih opcija. Bibliotekom logging formira se Logger objekat koji kontroliše ispis postavljanjem nivoa štampanja na jednu od numeričkih vrednosti CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET. Ako postavimo

na DEBUG, štampaće se sve. Umesto postavljanja nivoa, mogu se samo pozvati istoimeni metodi nad Logger objektom.

Ono što on stvara su objekti LogRecord klase. Svaki ovaj log zapis sadrži informacije o događaju koji se loguje, i sa njim kasnije rade druge klase iz ove bilioteke. Te klase su Handler-i koji šalju log zapise na odgovarajuću destinaciju i Filter-i koji na sofisticiraniji način od setLevel funkcije određuju koje log zapise staviti na izlaz. Tu su i Formatter-i koji određuju formu log zapisa na konačnom izlazu, mapirajući LogRecord objekat u nešto čitljivo čoveku ili nekom eksternom sistemu, najčešće string[4].

5 PDB debager

Pdb je deo Python standardne biblioteke i predstavlja interaktivni program za otklanjanje grešaka (*eng. debugger*)[5]. Ukoliko dođe do greške u kodu, predstavlja neophodan alat i poseduje velike mogućnosti poput praćenja izvršavanja programa korak po korak, pružajući nam pomoć pri rešavanju bagova ili pri razumevanju tuđeg koda. On rešava nedostatke prethodnih metoda i zbog svojih beneficija je u praksi u širokoj upotrebi.

5.1 Pokretanje debagera

Debagovanje našeg programa možemo pokrenuti:

- iz komandne linije
- iz samog programa

Što se tiče komandne linije, ukoliko program imenujemo kao prvi.py pokrećemo ga pod kontrolom debagera sa:
python -m pdb prvi.py arg1 arg2

Zapravo, pdb.py poziva se kao skript za debagovanje drugih skriptova. U ovom slučaju izvršavanje programa u debageru kreće od prve linije.

Pokretanje debagera iz samog programa možemo izvršiti na samom početku programa, iz proizvoljne linije ili nakon ispaljivanja izuzetka. To činimo tako što umetnemo određeni deo koda u program na mesto odakle želimo da započnemo proces debagovanja.

```
import pdb; pdb.set_trace()
```

Primer 6: Započinjemo debagovanje

pdb.set_trace() postavlja debager za pozivajući stek okvir. Debager je proširiv i zapravo je definisan kao klasa Pdb. Kada se izvrši linija iznad, program se zaustavlja i čeka sledeću naredbu. Prikazuje se pdb prompt. To znaci da je postavljena pauza u interaktivnom debageru i da mozemo uneti komandu. Od verzije 3.7 možemo pokrenuti debager funkcijom breakpoints() koja importuje pdb i poziva pdb.set_trace() [6]. Definisanjem promenljive BREAKPOINTS=0 onemogućavamo breakpoints(), čime prekidamo debagovanje [6].

5.2 Debagovanje korak po korak

Ilustovaćemo ga na primeru prvi.py.

```
my_list = [1,9,13,3,12]
new_list = list(map(lambda x: x*2,my_list))

def sub(a,b):
```

```
  print(a)
  return a-b

diff = sub(40,2)
my_list_sum = sum(my_list)
experiment = sum(new_list) / sub(diff,my_list_sum)
```

Primer 7: Primer za ilustrovanje narednih komandi (prvi.py)

Kada pokrenemo skript koristeći Python debager komandom *python -m pdb prvi.py* vidimo u konzoli sledeće:

```
> prvi.py(1)<module>()
-> my_list = [1,9,13,3,12]
(Pdb)
```

Primer 8: Ulazak u Pdb prompt

CLI(eng. command line interface) nam govori:
> započinje prvi red i govori u kojoj izvornoj datoteci se nalazimo, nakon toga u zagradama se nalazi broj linije u kodu na kojoj se nalazimo, a potom i ime funkcije. U slučaju da nismo upali u neku funkciju pisaće <module>()
–> započinje drugu liniju i to je trenutna linija u kojoj je program pauziran. Ta linija još uvek nije izvršena. Debager nam prikazuje sledeću liniju koja će biti izvršena (–> my_list = [1,9,13,3,12]). Komandom n (next) izvršavamo sledeću liniju.

```
(Pdb) n
> prvi.py(2)<module>()
-> new_list = list(map(lambda x: x*2,my_list))
```

Primer 9: Izvršavanje prve linije kôda

S obzirom da je prva linija sada izvršena, komandom p možemo prikazati vrednost neke promenljive.

```
(Pdb) p my_list
[1,9,13,3,12]
```

Primer 10: Štampanje promenljive *my_list*

```
(Pdb) n
> prvi.py(3)<module>()
-> def sub(a,b):
(Pdb) n
> prvi.py(6)<module>()
-> diff = sub(40,2)
(Pdb) s
--Call--
> prvi.py(3)<module>()
-> def sub(a,b):
```

Primer 11: Izvršavanje naredne dve linije i ulazak u funkciju *sub(40, 2)*

Ovde zapravo mozemo uočiti razliku izmedju n(next) i s(step) komande. Komanda n izvršava narednu liniju, dok s izvršava narednu liniju, ali ukoliko ona sadrži poziv neke funkcije, onda se vrši skok na prvu liniju te funkcije. Iz prethodnog možemo videti da smo trenutno pauzirani na funkciji sub() i sad prolazimo kroz nju.

```
(Pdb) n
> prvi.py(4)sub()
->print(a)
(Pdb) n
```

```
40
>prvi.py(5)sub()
->return a-b
(Pdb) n
--Return--
>prvi.py(5)sub->38
->return a-b
```

Primer 12: Izvršavamo prve tri linije u funkciji *sub(40, 2)*

I n i s će prekinuti izvršavanje kada dođemo do kraja trenutne funkcije i štampa se —Return —zajedno sa povratnom vrednošću na kraju sledećeg reda nakon –>. Komanda Enter pamti poslednju unetu komadu, a u ovom primeru je to n. Ukoliko unesemo jos 3 puta n zaredom prijaviće nam grešku ZeroDivisionError. Komandom q prekidamo debagovanje i izlazimo iz programa.

5.3 Postavljanje tačaka prekida

Tačke prekida(eng. breakpoints) mogu nam uštedeti puno vremena. Postavljamo ih tamo gde želimo da istražujemo. U naš prethodni primer dodajemo:

```
import pdb; pdb.set_trace()
experiment = sum(new_list) / sub(diff,my_list_sum)
```

Primer 13: U primer prvi.py na ovom mestu dodata prva linija

Ako sada program pokrenemo sa *python prvi.py* (slučaj 1), prva linija za izvršavanje korišćenjem debagera biće *experiment = sum(new_list) / sub(diff,my_list_sum)*. U slučaju da dodamo i ove argumente *–m pdb* (slučaj 2) dobijamo:

```
>prvi.py(1)<module>()
->my_list = [1,9,13,3,12]
(Pdb)
```

Primer 14: U slučaju 2 izvršavanje kreće od prve linije

Komanda c (continue) nastavlja izvršavanje dok se ne naiđe na tačku prekida.

```
(Pdb) c
40
>prvi.py(9)<module>()
->experiment = sum(new_list) / sub(diff,my_list_sum)
(Pdb) n
38
ZeroDivisionError: division by zero
```

Primer 15: Nastavlja se izvršavanje do tačke prekida i izvršava sledeću liniju

Tačke prekida možemo postavljati i komandom b (break). Navodi se broj linije ili ime funkcije u kojoj je izvršavanje zaustavljeno. U našem primeru bi to bilo, ako zakomentarišemo deo koji smo dodali:

```
(Pdb) b 9
```

Primer 16: Na liniji 9 postavljamo tačku prekida

jer je linija koju istražujemo deveta u kodu. Opis nekih komandi koje nismo naveli u primeru možete pogledati u tabeli 1.

Tabela 1: pdb komande

komanda	opis i komentar
a(rgs)	Ispisuje listu argumenata trenutne funkcije.
cl(ear) <number>	Uklanja tačku prekida obeleženu brojem.
w(here)	Prikazuje stanje steka(eng. stack trace). Ono što je najskorije dodato na stek se prikazuje na kraju. Pomaže nam da uočimo gde je izvršavanje zaustavljeno i koji je trenutni stek okvir.
u(p) [count]	Menja trenutni stek okvir i pomera ga za count nivoa iznad (ranije dodato) na stek okviru.
d(own) [count]	Menja trenutni stek okvir i pomera ga za count nivoa ispod (kasnije dodato) na stek okviru.
display [expression]	Ukoliko dođe do promene vrednosti izraza kada se izvršavanje zaustavi, display komandom automatski prikazujemo vrednost izraza. Ova komanda prikazuje sve izraze trenutnog stek okvira.
undisplay [expression]	Služi za brisanje prikaza trenutnog stek okvira.

5.4 Ostali python debageri

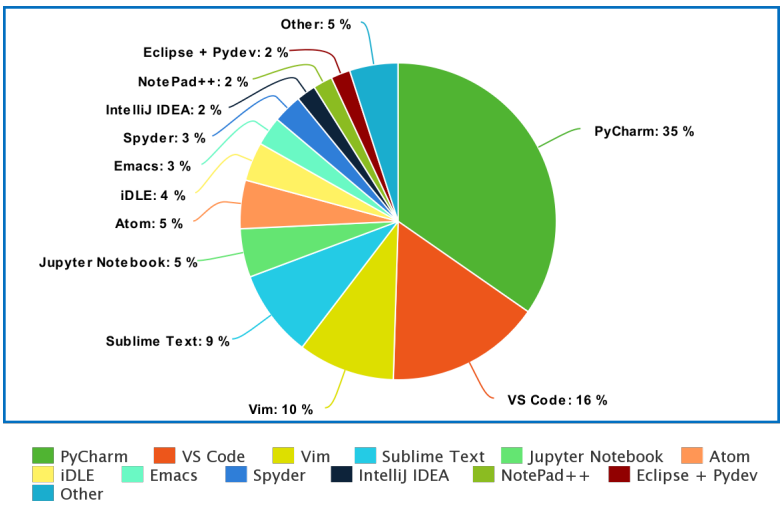
Do sada smo videli kako se koristi PDB debager, ali postoje i drugi alati za debagovanje. Neki od najpoznatijih i najkorišćenijih su bdb [1], pydbgr, pudb, ipdb, pdb++.

Moguće je debagovati korišćenjem nekih editora koji podržavaju jezik Python. Na prethodnoj slici 1 možemo videti neke od njih.

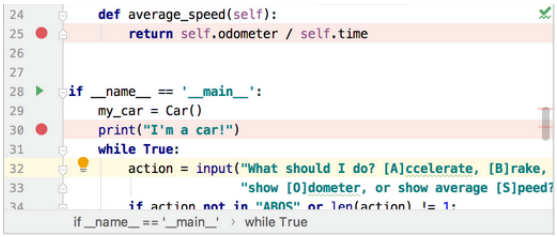
6 Debagovanje u okruženju PyCharm

PyCharm je integrisano razvojno okruženje koje se koristi za programiranje u jeziku Python. Pruža analizu koda, grafički debager, integraciju sa verzijom kontrolnog sistema(git) i druge pogodnosti. Da bi započeli debagovanje prvo moramo postaviti tačke prekida (*eng. breakpoint*) koji će signalizirati debageru da treba da se zaustavi na određenom mestu u kôdu i da nam da izveštaj stanja u tom trenutku. Tačku prekida postavljamo tako što klinemo na prazninu uz levu marginu.

Znaćemo da je tačka prekida uspešno postavljena pojavom crvenog kružića[3] kao na slici 2. Prilikom pokretanja main funkcije našeg programa možemo izabrati opciju Debug, čime otvaramo *Debug tool window* u kome možemo pokrenuti naš Python kôd i tu ćemo dobijati sve informacije o izvršavanju. Te informacije mogu sadržati poruke o greškama,



Slika 1: Populari Editori i IDE za rad sa Python-om



Slika 2: Postavljanje Tačaka prekida.

neuhvaćene izuzetke, vrednosti promenljivih (u svom zasebnom prozoru) itd[3]. PyCharm se automatski zaustavlja ukoliko naiđe na izuzetak koji nije uhvaćen, inače se zaustavlja na lokaciji prve tačke prekida[3]. Ukoliko program ima više niti, dobićemo posebne prozore za svaku od njih[3].

6.1 Detaljno debagovanje

Šta ako želimo da posmatramo izvršavanje našeg kôda korak po korak? Kao i pomenuti pdb, PyCharm debager poseduje mehanizam za to u svom Debug tool window-u, takozvani *Stepping toolbar*.

Slika 3: Stepping Tool Bar.

Često korišćene opcije koje su nam na raspolaganju su:

- Step Over
- Step Into

- Step Into My Code

Step Over jednostavno prelazi na sledeću liniju koda (linija na kojoj se trenutno nalazimo biće osenčena u editoru). Step Into opcija će nas voditi kroz biblioteke i funkcije koje koristimo kada na njih naiđemo[3].

```
x = random.nextInt();
y = f(x);
```

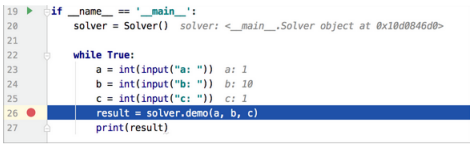
Primer 17: Primer korišćenja bilioteke i poziva funkcije

Ako primenimo opciju Step Into na prvoj liniji prethodnog kôda onda će nas ona odvesti u biblioteku Random,dok u drugoj liniji ista opcija bi nas odvela u definiciju funkcije f. Ako ovo ne želimo, koristimo opciju Step Into My Code koja će nas zadržati u našem kôdu (u zavisnosti od mesta definicije funkcije f, Step Into My Code moze imati isti efekat kao Step Into)[3].

6.2 Posmatranja (Watches)

PyCharm nam omogućava da posmatramo promenljive kroz izvršavanje našeg programa. U tabu debagera *Variables* se nalaze sve promenljive koje postoje i koje su vidljive u trenutnom stanju izvršavanja i na trenutnoj lokaciji u kodu, kao i tip i vrednost svake od njih[3]. Ako klinknemo na *plus* u gornjem levom uglu, dobijamo opciju da dodamo bilo koju promenljivu i ona će biti praćena uvek, bez obzira na to gde se nalazi, da li je trenutno vidljiva i da li je uopšte definisana[3].

6.3 Inline Debugger

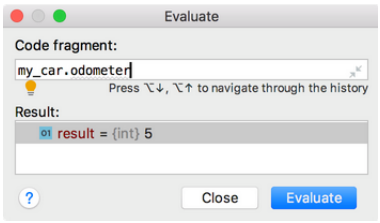


Slika 4: Inline Debugger.

Jedna od opcija koju nam pruža PyCharm jeste da klikom na tačku prekida odmah u editoru dobijemo informacije o našim promenljivama i objektima, u vidu komentara. Ova opcija je podrazumevana i može se promeniti u Debug Tool window-u[3].

6.4 Evaluacija izraza

Poslednja opcija koja se nalazi na Stepping Toolbar-u (slika 3) je opcija za evaluaciju izraza. Ova opcija nam omogućava da izračunamo vrednost neke promenljive koja nam je trenutno u opsegu ili nekog izraza[3]. Pitanje koje se postavlja je zašto bi ovo koristili ako isto možemo dobiti korišćenjem prethodno pomenutih posmatranja. Možemo, ali evaluacijom možemo uraditi i nešto što posmatranje ne može, a to je postavljanje vrednosti nekoj promenljivoj[3]. Ovo je veoma korisno, jer mozemo testirati naš kod za neke kritične vrednosti tako što ćemo na „vestacki“ način dodeljivati vrednosti promenljivima koje će nas dovesti do tog kritičnog stanja.



Slika 5: Evaluacija izraza.

7 Zaključak

Python je popularan jezik koji omogućava pisanje na višem, apstraktnijem nivou [8]. To nekada znači da nam on više "prašta" nego što bi trebalo i da se mogu "potkriti" razne greške. Zato je neophodno imati dobre alate za debagovanje koji mogu delovati kao ključni posrednici u pisanju korektnog kôda, čineći taj proces efikasnim i manje napornim. Tako imamo odličnu kombinaciju udobnosti u pisanju i pouzdanosti kôda. Od preventivnih mera kao što je hvatanje izuzetaka i jednostavnijih metoda debagovanja kao što su interakcija sa programom preko terminala i korišćenje print naredbi, zatim formalne naučne metode, pa sve do složenijih biblioteka logging, pdb i okruženja PyCharm, čitaocu je sada poznato više mehanizama debagovanja, koji predstavljaju samo jedan deo širokog opusa mogućnosti. Postoji mnogo više alata i oni sa pomenutim dele ključne koncepte. Za većinu njih postoji opsežna dokumentacija i dosta pomoći na internetu, što zbog popularnosti alata, što zbog popularnosti samog jezika. Ostavlja se čitaocu da preko datih objašnjenja izabere odgovarajući metod na osnovu svojih potreba i preferenci.

Literatura

[1] BDB. https://docs.python.org/2/library/bdb.html.

[2] Built-in Exceptions. https://docs.python.org/3/library/exceptions.html.

[3] Debug your first Python application. https://www.jetbrains.com/help/pycharm/debugging-your-first-python-application.html.

[4] Logging in Python. https://docs.python.org/3/library/logging.html.

[5] PDB. https://docs.python.org/3/library/pdb.html.

[6] PDB. https://realpython.com/python-debugging-pdb/.

[7] Pretty Printing. https://docs.python.org/3/library/pprint.html.

[8] Adawadkar Kalyani. Python Programming-Applications and Future. *International Journal of Advance Engineering and Research Development (IJAERD)*, 2017.

[9] Kristian Rother. *Pro Python Best Practices Debugging, Testing and Maintenance*. 2017.