

Debugovanje Python programa

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

28. mart 2020

Sažetak

U ovom tekstu je ukratko prikazana osnovna forma seminarskog rada. Obratite pažnju da je pored ove .pdf datoteke, u prilogu i odgovarajuća .tex datoteka, kao i .bib datoteka korišćena za generisanje literature. Na prvoj strani seminarskog rada su naslov, apstrakt i sadržaj, i to sve mora da stane na prvu stranu! Kako bi Vaš seminarski zadovoljio standarde i očekivanja, koristite uputstva i materijale sa predavanja na temu pisanja seminarskih radova. Ovo je samo šablon koji se odnosi na fizički izgled seminarskog rada (šablon koji *morate* da koristite!) kao i par tehničkih pomoćnih uputstava. Pročitajte tekst pažljivo jer on sadrži i važne informacije vezane za zahteve obima i karakteristika seminarskog rada.

Sadržaj

1 Uvod	2
2 Izuzeci u Pythonu	2
3 Debugovanje naučnom metodom	4
4 Debugovanje print naredbama	5
5 PDB debugger	6
6 Debugovanje u okruženju PyCharm	9
7 Zaključak	11
Literatura	11
A Dodatak	11

1 Uvod

Uvodni deo seminarskog

2 Izuzeci u Pythonu

Svaki put kada program ne radi onako kako smo očekivali znamo da je došlo do greške, odnosno бага. Debugovanje je proces pronalaženja i rešavanja tih greški. Ono podrazumeva sledeće stvari:

- Da znamo kako program treba da radi
- Da znamo da je do бага došlo
- Shvatamo da баг treba da uklonimo
- Uklanjamо баг

U Pythonu postoji 47 različitih izuzetaka, predstavljene kao hijerarhija izuzetaka [2]. Kada program izbaci izuzetak tada znamo da je do greške sigurno došlo i želimo da program taj izuzetak ne izbacuje. *Ako je program kuća, izuzetak bi označavao da je požar u kući* [7]. Izuzetke možemo da shvatamo kao бagove za koje znamo da postoje. Razmotrićemo tri osnovne strategije za debugovanje izuzetaka:

- Čitanje koda na mestu бага
- Razumevanje poruke o grešci
- Hvatanje izuzetaka

2.1 Sintaksne greške

Najlakši izuzeci za debugovanje su *SyntaxError* i *IndentationError*, где u oba slučaja prevodilac ne zna da prevede kôd. Ovakvi багови mogu da budu i česta pojava u Pythonu zbog razlika koje imaju verzije *Python2* i *Python3*, na primer funkcija **print** nema istu sintaksu u obe verzije. Tako da se neki programi prevode sa verzijom 2, a sa verzijom 3 izbacuju sintaksne greške. Razmotrimo sledeći primer 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

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 prethodni primer

Python je izbacio *SyntaxError* jer smo zaboravili zarez u liniji 4, s obzirom da je sintakсни analizator očekivao da su elementi u mapi razdvojeni zarezom izbacio je izuzetak. Slično, da smo posle dvotačke u prvoj liniji zaboravili da sledeća linija treba da bude nazubljena program bi izbacio *IndentationError*.

Sintaksne greške su često uzrok brzog kucanja, prelazak sa nekog drugog jezika, prelazak sa druge verzije jezika. Kod pojave ovakvih greški najbolje je gledati liniju greške (ili liniju iznad nje), prebaciti deo programa u zaseban fajl, proveriti uparenost zagrada i navodnika, proveriti da li je dobra verzija samog Python-a i preporučuje se korišćenje nekog naprednijeg editora.

2.2 Poruka o grešci

Kao što smo već mogli da vidimo, kada u programu postoji sintakсна 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 podklasa klase *Exception* u hijerarhiji izuzetaka [2].

Nakon tipa greške sledi opis greške šta se desilo, ovi opisi su neki put veoma jasni, a neki put ne daju nikakvu informaciju. U gornjem primeru tip greške je *SyntaxError*, a opis je *invalid syntax*.

Traceback sadrži informaciju 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.

2.3 Hvatanje izuzetaka

Neki izuzeci se ne mogu izbeći, ako uzmemo za primer da učitavamo neku datoteku i unesemo loše putanju ili možda ta datoteka više ne postoji, prevodilac će nam izbaciti *FileNotFoundError*. Na ovakve greške najbolje je rešiti hvatanjem izuzetaka unutar programa. To možemo da postignemo sa **try** i **except** blokom. Sa **try** pokušamo da pročitamo datoteku, ako dođe do izuzetka **except** blok će "uhvatiti" taj izuzetak i na tom mestu reagovati najčešće nekom porukom.

Ono što treba izbegavati sa hvatanjem izuzetaka jeste da u **except** bloku stavimo **pass** i na taj način nastavimo dalje izvršavanje programa kao da do izuzetka nije došlo[7].

2.4 Semantičke greške u Python-u

Program se preveo i ne izbacuje izuzetak, međutim i dalje ne dobijamo željeni rezultat, ovakve greške nazivamo semantičkim greškama. Takve greške je obično teže debugovati jer nemamo nikakvu informaciju od prevodioca da je do greške došlo, jedina informacija koju imamo jeste da ne dobijamo željeni 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 prethodnom primeru program računa sumu prvih n brojeva. Nekih od semantičkih greški koje su mogle da se dese su da range ide do n umesto do $n+1$, umesto operatora $+=$ staviti samo $=$, pogrešna inicijalizacija početne vrednosti za k , inicijalizacija unutar petlje umesto pre petlje. Semantičke greške se obično teže debuguju u ovakvim lakim primerima mogu da se uoče, ali u nekim kompleksijim primerima potrebne su neke od naprednijih tehnika. U narednim delovima ćemo se posvetiti tehnikama debugovanja, tehnike i upotrebe debagera i upotreba IDE za debugovanje.

3 Debugovanje naučnom metodom

U prethodnom delu, videli smo neke osnovne tehnike debugovanja. 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 nekom formalnom načinu pronalaženja problema kao što je naučni metod. On traženje greške bazira na prikupljanju dokaza i predstavlja okruženje (eng. framework) u koji se uklapaju ostale metode i dobru bazu za kasnije testiranje i održavanje koda. Koraci su sledeći[7]:

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 trebao da radi, pod uslovom da je hipoteza tačna
4. Testiraj: Ispitamo tu predikciju puštanjem programa u odgovarajućim eksperimentalnim uslovima i posmatramo rezultat izvršavanja
5. Zaključ: Zavisno od rezultata 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 debugovanje? 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. Neko nepisano pravilo je da ga primenimo ako ne nađemo rešenje u 10-15 minuta.

Da bi efikasno primenili ovaj način debugovanja, potrebno je da dobro vladamo tehnikama reprodukcije grešaka, automatizacije (pogotovo ako imamo složenije sisteme koji uključuju komunikaciju preko mreže ili nekakvu nasumičnost) i izolacije grešaka(strip-down strategijom ili strategijom binarne pretrage) [7]. Strip-Down strategija odlikuje se iterativnim uprošćavanjem koda komentarisanjem ili uklanjanjem linija, dok ne dođemo do minimalnog broja linija potrebnog za reprodukciju 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 o kojima ćemo pričati kasnije.

4 Debugovanje print naredbama

Print je mnogim programerima metoda broj jedan za debugovanje. Razlog za to leži u lakoći korišćenja, relativno čestom pronalaženju greške i prostom prikazivanju nedostatka informacija o podacima i izvršavanju (rother). 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 veći. 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; 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*[2]. 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 Pajtonovoj standardnoj biblioteci postoji moduo zvani *pprint*[6]. 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 debugovanja, ali olakšavajući čitljivost, umnogome ga olakšava.

4.1 Uključivanje i isključivanje print naredbi

Tokom debugovanja, 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 manualnog dodavanja, problem je i što stalno pisanje i brisanje nosi rizik novih grešaka u kodu. Zato mora da postoji neki način da uključimo i isključimo print naredbe u kodu. Najprimitivnije rešenje je definisanje neke indikatorske promenljive koja se postavlja na true kad se debuguje 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 debugovanje samo dodavanjem opcije -d argumentima komandne linije.

Ovaj koncept se može proširiti pajton bibliotekama logging [3] 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 biblioteke. Te klase su hendleri koji šalju log zapise na odgovarajuću destinaciju i filteri koji na sofisticiraniji način od levela određuju koje log zapise staviti na izlaz. Tu su i formateri koji određuju formu log zapisa na konačnom izlazu, mapirajući LogRecord objekat u nešto čitljivo čoveku ili nekom eksternom sistemu, najčešće string.

5 PDB debager

Pdb je deo Python standardne biblioteke i predstavlja interaktivni program za otklanjanje grešaka (eng. debugger) [4]. Ukoliko dođe do greške u kodu predstavlja neophodan alat i poseduje velike mogućnosti poput praćenja izvršavanja programa korak po korak što predstavlja pomoć pri rešavanju bagova na koje nailazimo ili pak za bolje razumevanje tuđeg koda. S obzirom da je u praksi u velikoj upotrebi cilj je da Vas kroz ovaj tekst upoznamo sa svim beneficijama upotrebe debagera i da klasičnu print metodu zaboravimo. Koristeći pip većinu verzija instaliramo sa:

```
pip install ipdb
```

ipdb-naprednija verzija standardnog Python pdb debagera.

5.1 Pokretanje debagera

Debugovanje 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
```

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

Pokretanje debagera iz samog programa možemo otpočeti na 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 debugovanja.

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

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 možemo uneti komandu. Od verzije 3.7 možemo pokrenuti debager funkcijom breakpoints() koja importuje pdb i poziva pdb.set_trace() [5]. Definisanjem promenljive BREAKPOINTS=0 onemogućavamo breakpoints() čime prekidamo debugovanje.

5.2 Debugovanje korak po korak

Ovu priču započinjemo primerom 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 6: prvi.py

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

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

Primer 7: ispis

CLI(eng comand 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 8: ispis

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 9: ispis

```
(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 10: ispis

Ovde zapravo mozemo uočiti razliku izmedju `n`(next) i `s`(step) komande. `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 11: ispis

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, u ovom primeru je n. Ukoliko unesemo jos 3 puta n zaredom prijavice nam grešku `ZeroDivisionError`. Komandom `q` prekidamo debugovanje i izlazimo iz programa.

5.3 Postavljanje tacaka 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 12: prvi.py

Ako sada program prevedemo sa python prvi.py prva linija za izvršavanje korišćenjem debagera biće `experiment = sum(new_list) / sub(diff,my_list_sum)`. U slučaju da to uradimo dopunom `-m pdb` dobijamo

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

Primer 13: ispis

Komanda `c` 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 14: ispis

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 15: ispis

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

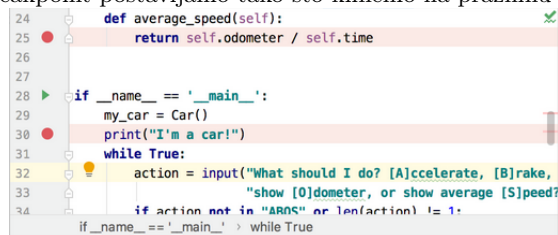
komanda	opis i komentar
a(args)	Ispisuje listu argumenata trenutne funkcije.
cl(ear) <number>	Uklanja tačku prekida obeleženu brojem.
w(here)	Prikazuje stanje steka(eng. stack trace). Ono što je najskorije dodato na stek se prikazuje na kraju. Pomaže nam da uočimo gde je izvršavanje zaustavljeno i koji 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 debugovanje. Jedni od najpoznatijih i najkorišćenijih su bdb [1], pydbgr, pudb, ipdb, pdb++.

6 Debugovanje u okruženju PyCharm

U ovom delu pokazaćemo kako debugovati u razvojnom okruženju PyCharm. Da bi započeli debug sesiju prvo moramo postaviti *prekide* (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. Breakpoint postavljamo tako što klinemo na prazninu uz levu marginu.



Znaćemo da je Breakpoint uspešno postavljen pojavom crvenog kružića. Prilikom pokretanja main funkcije našeg programa možemo izabrati opciju Debug, ovo će otvoriti *Debug tool window* u kome možemo pokrenuti naš python kôd i gde ćemo dobijati sve informacije o izvršavanju. Informacije koje dobijamo mogu sadržati poruke o greškama, ne uhvaćene izuzetke, vrednosti promenljivih (u svom zasebnom prozoru) i druge. PyCharm se automatski zaustavlja ukoliko naiđe na izuzetak koji nije uhvaćen inače se

zaustavlja na lokaciji prvog breakpoint-a. Ukoliko program ima više niti dobićemo posebne prozore za svaku od njih.

6.1 Detaljno debugovanje

Šta ako želimo da posmatramo izvršavanje našeg kôda korak po korak? Da li ovo znači da moramo postaviti breakpoint u svakoj liniji? Odgovor je ne, PyCharm debugger poseduje, u svom Debug tool window, takozvani *Stepping toolbar*.



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.

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

Primer 16: Primer neki

ovaj kod će nas odvesti u biblioteku Random ako na ovoj liniji koristimo opciju Step Into odnosno u definiciju funkcije f, ovo često ne želimo pa koristimo opciju Step Into My Code koja će nas zadržati u našem kôdu.

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 njihov tip i vrednost. 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 ona nalazi, da li je trenutno vidljiva i da li je uopšte definisana.

6.3 Inline Debugger

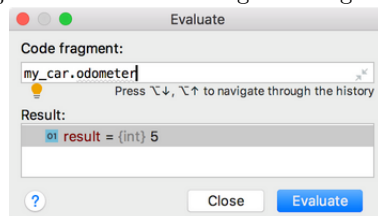
Jedna od opcija koju nam pruža PyCharm jeste da klikom na Break point odmah dobijamo informacije o našim promenljivima i objektima odmah u editoru u vidu komentara. Ova opcija je podrazumevana i može se promeniti u Debug Tool window-u.

```
19 ▶ if __name__ == '__main__':  
20     solver = Solver() solver: <__main__.Solver object at 0x10d0046d0>  
21  
22     while True:  
23         a = int(input("a: ")) a: 1  
24         b = int(input("b: ")) b: 10  
25         c = int(input("c: ")) c: 1  
26 ● result = solver.demo(a, b, c)  
27         print(result)
```

6.4 Evaluacija izraza

Poslednja opcija koja se nalazi na Stepping Toolbar-u je opcija za evaluaciju izraza. Ovo opcija nam omogućava da izračunamo vrednost neke promenljive koja nam je trenutno u opsegu ili nekog izraza. Pitanje

koje se postavlja je zašto bi ovo koristili jer isto možemo dobiti korišćenjem Posmatranja. Ovo je tačno ali evaluacijom možemo uraditi nešto što Posmatranje ne može, a to je da postavimo vrednost nekoj promenljivoj. Ovo je jako korisno jer možemo testirati naš kod za neke kritične vrednosti tako što ćemo na ‘vestack’ način da dodeljujemo vrednosti promenljivima koje će nas dovesti do tog kritičnog stanja.



7 Zaključak

Ovde pišem zaključak.

Literatura

- [1] BDB. on-line at: <https://docs.python.org/2/library/bdb.html>.
- [2] Built-in Exceptions. on-line at: <https://docs.python.org/3/library/exceptions.html>.
- [3] Logging in Python. on-line at: <https://docs.python.org/3/library/logging.html>.
- [4] PDB. on-line at: <https://docs.python.org/3/library/pdb.html>.
- [5] PDB. on-line at: <https://realpython.com/python-debugging-pdb/>.
- [6] Pretty Printing. on-line at: <https://docs.python.org/3/library/pprint.html>.
- [7] Kristian Rother. *Pro Python Best Practices Debugging, Testing and Maintenance*. 2017.

A Dodatak

Ovde pišem dodatne stvari, ukoliko za time ima potrebe.