Naslov seminarskog rada

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

Prvi autor, drugi autor, treći autor, četvrti autor
kontakt email prvog, drugog, trećeg, četvrtog autora

13. 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

Uvodni deo seminarskog

2 Izuzeci u Pythonu

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 greški. Ono podrazumeva sledeće stvari:

- Znamo kako naš program bi trebao da radi
- Znamo da je došlo do baga
- Shvatamo da bag treba da uklonimo
- Uklanjamo bag

U Pythonu postoji 47 različitih izuzetaka, predstavljene kroz hijerarhiju [1]. Kada program izbaci izuzetak tada znamo da je došlo do greške i očigledno želimo da program taj izuzetak ne izbacuje. *Ako je program kuća, izuzetak bi označavao da je požar u kući* [2]. Izuzetke možemo da shvatamo kao bagove za koje znamo da postoje. Razmotrićemo tri osnovne strategije za debagovanje izuzetaka:

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

2.1 Sintaksne greške

Najlakši izuzeci za debagovanje su SyntaxError i IndentionError. U oba slučaja Python ne uspeva da prepozna neki deo programa. Ovakvi bagovi 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(ime):
    studenti = {
        'Pera': '107/2016',
        'Mika': '16/2016'
        'Laza': '252/2015'
    }

    print('Indeks studenta Pera je ' + studenti[ime])

student('Pera')
```

Listing 1: Primer neki

Program ne uspeva i izbacuje narednu grešku.

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

Listing 2: ispis

Python je izbacio *SyntaxError* jer smo zaboravili zarez u liniji 4, s obzirom da je sintaksni 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 *IndentionError*.
Sintaksne greške su često uzrok brzog kucanja, prelazak sa nekog drugog jezika, prelazak sa druge verzije jezika. Neki od saveta za debagovanje sintaksnih greški su:

- Pogledaj liniju greške, ili liniju iznad nje
- Prebaciti deo programa koji sadrži grešku u zasebam fajl
- Proveriri da li su sve zagrade uparene
- Proveriti da li su svi navodnici upareni
- Proveriti da li koristite dobru verziju Pythona
- Koristite neki dobar editor u kome se lepo vide sintaksne greške.

2.2 Poruka o grešci

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 podklasa klase *Exception* u hijerarhiji izuzetaka.
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 ununtar 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.

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 debagovati 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
```

Listing 3: Računanje sume prvih n brojeva

U prethodnom primeru program računa sumu prviih 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 debaguju u ovakvim lakim primerima mogu da se uoče, ali u nekim kompleksijim primerima potrebne su neke od naprednihih tehnika. U narednim delovima ćemo se posvetiti tehnikama debagovanja, tehnike i upotrebe debagera i upotreba IDE za debagovanje.

3 Debagovanje naučnom metodom

4 Debagovanje print metod

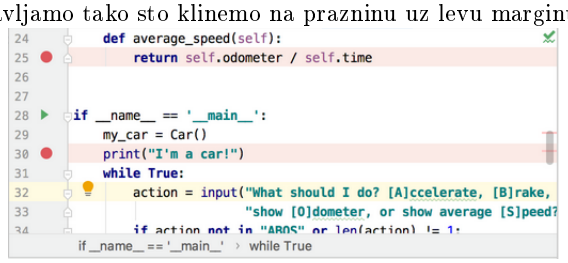
5 Tehnike debagera

6 PDB debager

7 Ostali python debageri

8 Debagovanje u okruzenju PyCharm

Da bi zapoceli debug sesiju prvo moramo postaviti *prekide (eng. breakpoint)* koji ce signalizirati debageru da treba da se zaustavi na odredjenom mestu u kodu i da nam da izvestaj stanja u tom trenutku. Breakpoint postavljamo tako sto klinemo na prazninu uz levu marginu.



Znacemo da je Breakpoint uspesno postavljen pojavom crvenog kruzica. Prilikom pokretanja main funkcije naseg programa mozemo izabrati opciju Debug, ovo ce otvoriti *Debug tool window* u kome mozemo pokrenuti nas python kod I gde cemo dobijati sve informacije o izvrsavanju. Informacije koje dobijamo mogu sadrzati poruke o greskama, ne uhvacene izuzeteke, vrednosti promenljivih (u svom zasebnom prozoru) I druge. PyCharm se automatski zaustavlja ukoliko naidje na izuzetak koji nije uhvacen inace se zaustavlja na lokaciji prvog breakpoint-a. Ukoliko program ima vise niti dobicemo posebne prozore za svaku od njih.

8.1 Detaljno debagovanje

Sta ako zelimo da posmatramo izvrsavanje naseg koda korak po korak? Da li ovo znaci da moramo postaviti breakpoint u svakoj liniji? Odgovor je

ne, PyCharm debugger poseduje , u svom Debug tool window, takozvani *Stepping toolbar*.

Cesto koriscene opcije koje su nam na raspolaganju su:

- Step Over
- Step Into
- Step Into My Code

Step Over jednostavno prelazi na sledecu liniju koda (linija na kojoj se trenutno nalazimo bice osencena u editoru). Step Into opcija ce nas voditi kroz biblioteke I funkcije koje koristimo kada na njih naidjemo

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

Listing 4: Primer neki

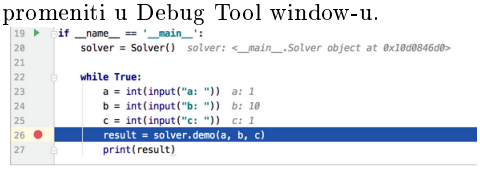
ovaj kod ce nas odvesti u biblioteku Random ako na ovoj liniji koristimo opciju Step Into tj u definiciju funkcije f, ovo cesto ne zelimo pa koristimo opciju Step Into My Code koja ce nas zadrzati u nasem kodu.

8.2 Posmatranja (Watches)

PyCharm nam omogucava da posmatramo promenljive kroz izvrsavanje naseg programa . U tabu debagera *Variables* se nalaze sve promenljive koje postoje I koje su vidljive u trenutnom stanju izvrsavanja 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 ce biti pracena uvek bez obzira na to gde se ona nalazi , da li je trenutno vidljiva I da li je uopste definisana.

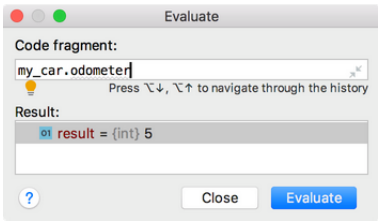
8.3 Inline Debugger

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



8.4 Evaluacija izraza

Poslednja opcija koja se nalazi na Stepping Toolbar-u je opcija za evaluaciju izraza. Ovo opcija nam omogucava da izracunamo vrednost neke promenljive koja nam je trenutno u opsegu ili nekog izraza. Pitanje koje se postavlja je zasto bi ovo koristili jer isto mozemo dobiti koriscenjem Posmatranja. Ovo je tacno ali evaluacijom mozemo uraditi nesto sto Posmatranje ne moze a to je da postavimo vrednost nekoj promenljivoj. Ovo je jako korisno jer mozemo testirati nas kod za neke kriticne vrednosti tako sto cemo na 'vestack' nacin da dodeljujemo vrednosti promenljivima koje ce nas dovesti do tog kriticnog stanja.



9 Zaključak

Ovde pišem zaključak.

Literatura

[1] Built-in Exceptions. on-line at: https://docs.python.org/3/library/exceptions.html.

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

A Dodatak

Ovde pišem dodatne stvari, ukoliko za time ima potrebe.