Upravljanje pogreškama i iznimkama u Pythonu

Burić, Matej

Sveučilište u Splitu
Prirodoslovno-matematički fakultet
Odjel za informatiku

Matej Burić

UPRAVLJANJE POGREŠKAMA I IZNIMKAMA U PYTHONU

Završni rad

Split, 2024.

Temeljna dokumentacijska kartica

Završni rad

Sveučilište u Splitu
Prirodoslovno-matematički fakultet
Odjel za informatiku
Ruđera Boškovića 33, 21000 Split, Hrvatska

Upravljanje pogreškama i iznimkama u Pythonu

Matej Burić

Sažetak

Tema ovog rada je upravljanje pogreškama i iznimka u Pythonu. U radu je prikazano kako upravljanje pogreškama i iznimkama u Pythonu omogućava programerima da identificiraju i obrađuju neočekivane situacije koje mogu nastati tijekom izvršavanja programa. Obradili smo tri osnovne vrste pogrešaka: sintaksne, sematičke i pogreške u izvođenju te smo prošli kroz korake za njihovo otklanjanje. Osim toga, obrađene su i iznimke koje se javljaju tijekom izvršavanja programa. `Try-except` mehanizmi omogućuju nam da na predviđene probleme odgovorimo prikladnim rješenjima.

Ključne riječi: Python, pogreške, iznimke, try-except

Rad je pohranjen u knjižnici Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu

Rad sadrži: 28 stranica, 0 grafičkih prikaza, 2 tablice, 10 literaturnih navoda.

Izvornik je na hrvatskom jeziku

Mentor: prof. dr. sc. Ani Grubišić, *redoviti profesor Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Ocjenjivači: prof.dr.sc. Branko Žitko, *redoviti profesor Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu,*

Ines Šarić-Grgić, dipl.ing. *, asistent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Split*

Basic documentation card

Thesis

University of Split

Faculty of Science

Department of Computer Science

Ruđera Boškovića 33, 21000 Split, Croatia

Exception & Error Handling in Python

Matej Burić

Abstract

The topic of this paper is error and exception handling in Python. The paper illustrates how managing errors and exceptions in Python allows programmers to identify and handle unexpected situations that may arise during program execution. We have covered three main types of errors: syntax errors, semantic errors, and runtime errors, and detailed the steps for their resolution. Additionally, we addressed exceptions that occur during program execution. Try-except mechanisms enable us to respond to anticipated problems with appropriate solutions. Besides try-except, else and finally blocks can be used to provide further elaboration and control over the code.

Key words: Python, error, debugger, exceptions, try-except

Supervisor: Ani Grubišić, Ph.D., Full Professor, Faculty of Science, University of Split

Reviewers: Branko Žitko, Ph.D, Associate Professor, Faculty of Science, University of Split

Ines Šarić-Grgić, Ph.D. Assistant, Faculty of Science, University of Split

IZJAVA

kojom izjavljujem s punom materijalnom i moralnom odgovornošću da sam završni rad s naslovom UPRAVLJANJE POGREŠKAMA I IZNIMKAMA U PYTHONU izradio samostalno pod voditeljstvom prof. dr. sc. Ani Grubišić. U radu sam primijenio metodologiju znanstvenoistraživačkog rada i koristio literaturu koja je navedena na kraju završnog rada. Tuđe spoznaje, stavove, zaključke, teorije i zakonitosti koje sam izravno ili parafrazirajući naveo u završnom radu na uobičajen, standardan način citirao sam i povezao s fusnotama s korištenim bibliografskim jedinicama. Rad je pisan u duhu hrvatskog jezika.

Student

Matej Burić

Sadržaj

1 UVOD

U razvoju programske podrške, upravljanje pogreškama i iznimkama ključna je komponenta koja osigurava stabilnost i pouzdanost sustava. U Pythonu su pogreške i iznimke različiti pojmovi. Pogreške su nedostaci u kodu koji sprječavaju njegovo uspješno izvršavanje. S druge strane, iznimke su događaji koji ometaju normalan tijek programa tijekom izvođenja, ukazujući na to da se dogodilo nešto neočekivano ili pogrešno. Iznimke se mogu obraditi pomoću `try-except` blokova kako bi se na prikladan način upravljalo neočekivanim situacijama, dok pogreške obično zahtijevaju ispravljanje osnovnog problema u kodu. Općenito, pogreške se otkrivaju tijekom faze kompilacije ili interpretacije, dok se iznimke javljaju za vrijeme izvođenja programa. Bez odgovarajuće obrade iznimki i pogrešaka, aplikacije mogu postati nestabilne, uzrokovati gubitak podataka ili čak potpuno prestati s radom. Za rješavanje problema, moderni programski jezici nude različite mehanizme za upravljanje pogreškama. Ovi mehanizmi omogućuju programerima da prepoznaju i obrade pogreške na način koji ne ometa rad aplikacije. U mnogim jezicima, kao što su Python, Java, i C#, koriste se specijalizirani blokovi koda za hvatanje i upravljanje iznimkama. Ovi blokovi omogućuju detaljnu obrada pogrešaka i pružaju korisnicima i programerima bolje informacije o problemima koji nastaju. U Pythonu, na primjer, korištenje `try-except` mehanizma koje ćemo detaljnije objasniti u nastavku omogućuje programerima da uhvate i obrade iznimke na način koji pomaže u održavanju stabilnosti aplikacije. Ovaj pristup omogućuje jednostavno i učinkovito upravljanje iznimkama, čime se poboljšava ukupna kvaliteta i pouzdanost koda.

2 POGREŠKE U PYTHONU

Pogreške u programiranju, poznate kao i bugovi, neizbježan su dio razvoja programske podrške. One nastaju, iz različitih razloga, a mogu biti ljudske pogreška, složenost koda, nedostatak testiranja i sl. Takve pogreške mogu uzrokovati neočekivano ponašanje programa, kvarove ili netočne rezultate. Termin „bug“ postao je popularan u kontekstu računalnog programiranja zahvaljujući incidentu 1947. godine kada je Grace Hopper, znanstvenica računalne znanosti, zajedno sa svojim timom otkrila moljca zaglavljenog u releju Mark II računala na Harvardu. Od tada, riječ „debugging“ je postao standardni izraz za proces pronalaženja i uklanjana grešaka u kodu kojeg je i osmislila Grace Hopper[1]. Pogreške u programiranju mogu se pojaviti u različitim fazama razvoja programske podrške, od dizajna i kodiranja do implementacije i održavanja. Danas postoje brojni alati i tehnike, uključujući automatizirano testiranje, statičku analizu koda i sl. koje pomažu u smanjenju broja pogreški i poboljšanju kvalitete softvera. Kada gledamo unatrag i razvijanje kroz povijest računalni sustavi su postajali sve složeniji i složeniji, tako da je rasla i potreba za sofisticiranijim metodama za upravljanje pogreškama. Rani računalni programi često su se izvodili na hardveru s vrlo ograničenim resursima, što je dodatno otežavalo otkrivanje i ispravljanje pogrešaka. Razvoj naprednijih „debugging“ alata i boljih praksi programiranja značajno je unaprijedio sposobnost programera da se nose s pogreškama. S napretkom tehnologije i metodologija za razvoj programske podrške, današnji programeri imaju pristup širokom spektru resursa koji im pomažu u prevenciji i rješavanju pogrešaka. Međutim, unatoč svim napredcima i tehnologiji, pogreške ostaju sastavi dio programiranja, podsjećajući nas na složenost i izazove koje nosi stvaranje sustava programske podrške. Pogreške koje mogu nastati unutar programa dijele se na tri glavne vrste: sintaksne pogreške, logičke pogreške te pogreške tijekom izvođenja programa (eng. run-time errors). Detaljnije ćemo objasniti svaku vrstu pogrešaka u sljedećim podpoglavljima.

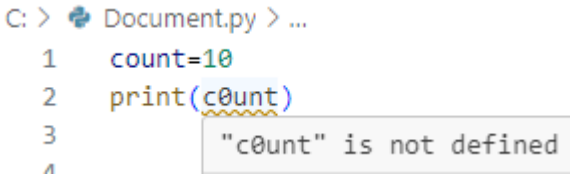
2.1 Sintaksne pogreške

Pogreške u sintaksi krše jedno od Pythonovih gramatičkih pravila i sprječavaju daljnje izvođenje programa. Sintaksne pogreške u Pythonu događaju se kada Python interpreter nije u mogućnosti prepoznati strukturu izraza u kodu. U Pythonu svaka se skripta prije izvršavanja kompajlira u bajt-kod (engl. Bytecode), jer Python nije 100% interpreterski jezik. Prilikom ovog procesa se već otkrivaju pogreške sintakse, i ako postoji pogreška interpreter podiže poruku zvanu „Traceback“[2]. I prije nego pokrenemo izvršavanje koda u blizini pogreške najčešće bude i valovita crvena linija koja ukazuje na pogrešku tako da je ovu vrstu pogreške lako uočiti i samim time lako i popraviti. Primjer sintaksne pogreške nalazi se na sljedećoj slici.

Slika 1 Primjer pogreške u sintaksi

U ovom primjeru imamo funkciju `print()` koja bi trebala ispisati zadani tekst . Međutim, nedostaje završna zagrada, zbog toga Python ne može ispravno interpretirati kod jer očekuje zagradu po završetak izraza. U slijedećem dijelu poglavlja analizirat ćemo najčešće pogreške koje se javljaju pri pisanju koda, s naglaskom na sitne, ali česte greške koje mogu uzrokovati probleme u izvršavanju programa. Ove greške uključuju zamjenu slova i brojeva, nedostatke u sintaksi poput viška ili manjka slova i zagrada, kao i pogrešno pridruživanje vrijednosti varijablama. Također ćemo se pozabaviti specifičnim slučajevima poput nepravilnog korištenja zareza i navodnika, te problemima koji nastaju zbog krivog uvlačenja i nepravilnog smještanja naredbi unutar petlji [3].

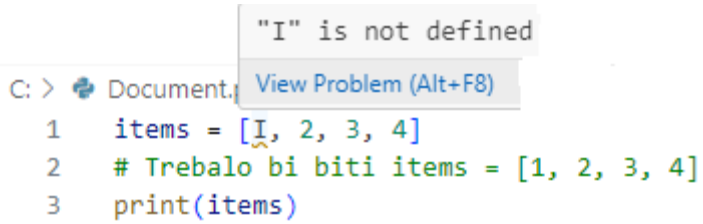
Slovo O umjesto 0 ili obratno



Slika 2 Slovo O umjesto 0 ili obratno

Ovo će rezultirati sintaksnom pogreškom jer Python očekuje slovo, a ne broj.

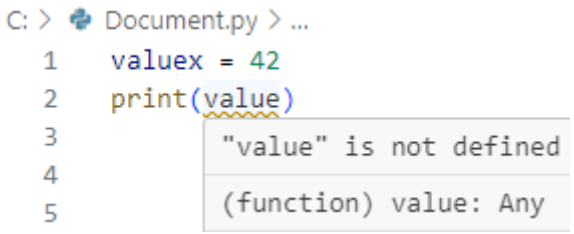
Slovo I umjesto 1 ili obratno



Slika 3 Slovo I umjesto 1 ili obratno

Slično kao u prethodnom primjeru, korištenje slova „I“ umjesto broja „1“ može uzrokovati pogrešku.

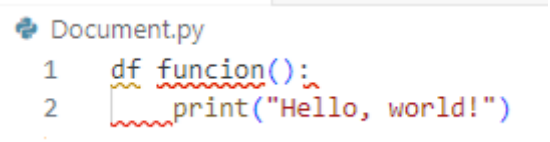
Upisali ste slovo viška



Slika 4 Upisali ste slovo viška

U ovom primjeru, slovo viška ili zamijenjeno slovo uzrokuje pogrešku jer `value` nije definirana varijabla.

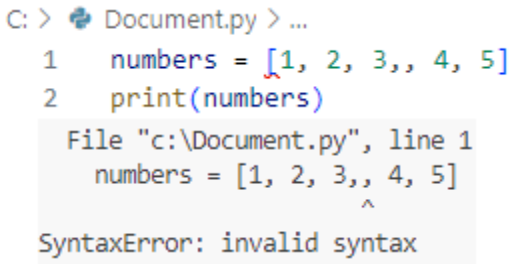
Zaboravili ste slovo



Slika 5 Zaboravili ste slovo

Ovdje Python ne može prepoznati ključnu riječ `def` zbog zaboravljenog slova „e“, što uzrokuje pogrešku.

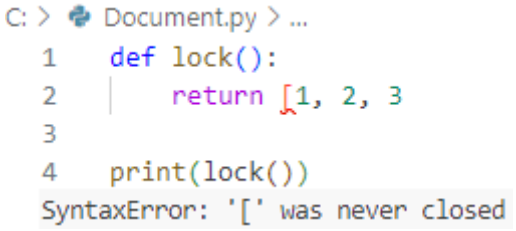
Stavili ste previše zareza



Slika 6 Stavili ste previše zareza

Dodatni zarez u nizu brojeva uzrokuje sintaksnu pogrešku jer Python očekuje vrijednost nakon svakog zareza.

Zaboravili ste zagrade

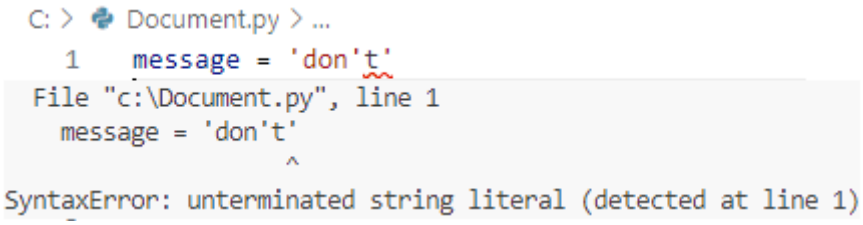


Slika 7 Nedostaje zagrada

Ovdje Python misli da nam se niz sastoji od tri elementa: `1, 2 i 3 print(lock())`. Iza broja tri nema uglate zagrade i Python automatski misli da se i `print` nalazi zajedno s brojem tri u listi i tretira ih kao treći element liste.

```
result = (firstVal - secondVal / factor
```

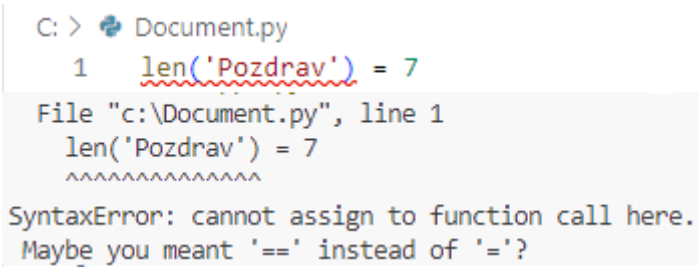
Niste stavili navodnike oko svakog stringa



Slika 8 Pogreška s navodnicima oko stringa

Imamo poruku „don't“ koja nam koristi tri jednostruka navodnika i izaziva grešku. Ovdje bi rješenje bilo korištenje dvostrukih navodnika kako bi izbjegli ovu pogrešku.

Krivo pridruživanje vrijednosti



Slika 9 Krivo pridruživanje vrijednosti

Funkcija `len(„Pozdrav“)`, vraća broj znakova u riječi „Pozdrav“, što iznosi 7. Međutim mi ne možemo pridijeliti vrijednost funkcijskom pozivu i na to nas upozorava pogreška `SyntaxError: cannot assign to function call`.

- x = 2
- y = 3.1415
- x * y = product

Pogrešno napisana ključna riječ

```
C: > Document.py
1    fro i in range(10):
  File "c:\Document.py", line 1
    fro i in range(10):
        ^
SyntaxError: invalid syntax
```

Slika 10 Pogrešno napisana ključna riječ

Ovdje poruka `SyntaxError: invalid syntax` možda i nije vrlo informativna, ali Python nas strelicom ispod `i` navodi gdje je došlo do pogreške, iako se pogreška najčešće dogodi prije mjesta na koje Python upozorava. Ovdje je slučaj da smo napisali `fro` umjesto ključne riječi `from` i Python javlja pogrešku. Uobičajena tri načina na koja se ključna riječ može pogrešno koristiti su:

- Pogrešno napisana ključna riječ
- Izostavljanje ključne riječi
- Nepravilno korištenje ključne riječi

Pogrešno korištena ključna riječ

```
C: > Document.py > ...
1   popis_gostiju = ['Luka', 'Petar', 'Karla']
2   if 'Petar' in popis_gostiju:
3       print('Petar je na popisu.')
4       break
  File "c:\Document.py", line 4
    break
    ^^^^^
SyntaxError: 'break' outside loop
```

Slika 11 Pogrešno korištenje ključne riječi

U ovom primjeru, Python vrlo jasno upućuje gdje je došlo do pogreške i što je točno pogrešno. U ovom slučaju to je ključna riječ `break` koja je izvan petlje.

Razlozi zašto računalo previše ispisuje

Krivo uvlačenje

Stavili ste print unutar petlje umjesto izvan nje

```
C: > Document.py > ...
1   def brojac():
2       for i in range(10):
3           print(i)
4           print('Završeno')
5   brojac()
6
```

```
#Ispis:
0
Završeno
1
Završeno
2
Završeno
```

```
#Ispravan ispis:
5
6
7
8
9
Završeno
```

Slika 12 Krivo uvlačenje i print unutar petlje

„`Ispravan ispis`“ nam prikazuje kako bi izlaz trebao izgledat, a nama ispisuje kao kod „`Ispis`“. Problem je u tome što nam se naredba `print('Završeno')` nalazi unutar `for` petlje i nakon svake iteracije zajedno s brojem ispisuje ono što se nalazi unutar naredbe `print`. Da bi napravili ispravan ispis potrebno je samo pravilno uvući `print` da nam bude u razini `for` petlje[3].

2.2 Semantičke pogreške

Logičke ili semantičke pogreške, poput krivog uvjeta u petlji, teško su uočljive, uzrokuju netočne rezultate, ali ne sprečavaju daljnje izvođenje programa[4]. Slika 13 prikazuje krivo postavljanje funkcije čiji je cilj kvadriranje, a obavlja funkciju množenja što rezultira neočekivanim rezultatom.

```
C: > Document.py > ...
1   def kvadriraj(broj):
2       return broj*broj
3
4
5   rezultat=kvadriraj(2)+kvadriraj(3)
6   print("Rezultat je: ", rezultat)
  Rezultat je:  13
```

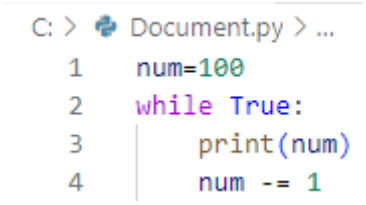
Slika 13 Primjer logičke pogreške

Semantičke ili logičke pogreške su vrste pogrešaka u programiranju gdje je sintaksa koda točna, ali ipak izlaz koji dobivamo nije očekivan ili je rezultat pogrešan. Npr. ako programer nije napisao kod potreban za pretvaranje ulaza u cijele brojeve, program će ih spremiti kao `string`, te spojiti u jedno i tako ispisati. Naravno poruku o grešci nećemo dobiti, iako dobiveni izlaz nije ono što smo zamislili. Prepoznavanje ovih grešaka može biti nezgodno jer se ne dobiva poruka o krivom izlazu ili mjestu gdje je nastala pogreška, pa ih je zato i najteže za otkriti.

U nastavku ovog poglavlja proučit ćemo kako prepoznati i ispraviti beskonačne petlje koje mogu uzrokovati da program nikada ne završi, te kako razumjeti prioritet operatora. Razjasnit ćemo i često zbunjujuće situacije s `else` izjavama koje mogu dovesti do nepredviđenih ponašanja programa. Također ćemo istražiti probleme poput neispravnog broja izvršavanja petlji, viška koda unutar petlji koji usporava rad programa, te pogrešnih indeksa u nizovima koji mogu uzrokovati greške u pristupu podacima.

Beskonačna petlja

Beskonačna petlja (engl. Infinite Loop) je situacija u kojoj petlja nema završetka, to znači da će se neprekidno izvršavati sve dok se program prisilno ne zaustavi. Pojavit će se u slučaju kada se unutar petlje ne postavi uvjet koji omogućuje izlaz iz petlje. Ovaj slučaj može dovesti do iscrpljivanja resursa (CPU, memorija) ili do zamrzavanja programa. Razumijevanje uvjeta i varijabli koje kontroliraju tok petlje ključ su za izbjegavanje beskonačne petlje. Primjer na Slici 14 prikazuje kako `while` petlja kreće od 100 pa minus jedan 99,98… i tako dok je god tvrdnja istinita, a to može biti beskonačno dugo:



Slika 14 Beskonačna petlja

Prioritet operatora

Prioritet operatora se pojavljuje u složenijim izrazima gdje se pojavljuje više operatora. Primjer: `>>> 10 - 4 * 2 rezultat je 2`. Množenje ima veći prioritet od oduzimanja i izvršava se prvo. Najjednostavniji način za rješenje ovog problema je postavljanjem zagrada. Primjer: `>>> (10 - 4) * 2 rezultat je 12`. Izraz u zagradama se izvršava prvi jer zagrade imaju najveći prioritet i definiraju da se izrazi unutar njih prvi obrađuju[7]. Kada postoje operatori s istim prioritetom tada se koristi pravilo asocijativnosti (engl. associativity), koje određuje hoće li se isti operatori obrađivati s desna na lijevo ili obratno.
Tablica 1 prikazuje razinu prioriteta i kreće od najvećeg ka manjem.

Tablica 1 Tablica asocijativnosti i prioriteta

Operatori	Naziv	Asocijativnost
()	Zagrade	S lijeva na desno
****	Eksponent	S desna na lijevo
***, /, //, %**	Množenje, dijeljenje, cjelobrojno dijeljenje, modul	S lijeva na desno
+, -	Zbrajanje, oduzimanje	S lijeva na desno
==, !=, >, <, >=, <=, is, is not, in, not in,	Operatori uspoređivanja, identiteta i članstava	S lijeva na desno
Not, and, or	Logički operatori	S lijeva na desno

Nejasni else

Nejasni `else` se često pojavljuje u situacijama gdje postoji više ugniježdenih `if-else` naredbi, a nije jasno koja „`IF`“ naredba odgovara kojoj „`ELSE`“ naredbi. Primjer nejasnog `else` bloka prikazan je na idućoj slici:

Document.py > ...

```
1   temperatura = 25
2
3   if temperatura > 30: #uvjet1
4       print("Vani je vruće!") #kod
5   if temperatura > 20: #uvjet2
6       print("Vani je toplo!") #kod
7   else:
8       print("Vani je hladno!")
9       #ovaj else blok je nejasan jer se ne zna,
10      #odnosi li se na uvjet1 ili uvjet2
```

Slika 15 Primjer nejasnog else-a

Ispravan način*:*

Document.py > ...

```
1   temperatura = 25
2
3   if temperatura > 30: #uvjet1
4       print("Vani je vruće!") #kod
5   elif temperatura > 20: #uvjet2
6       print("Vani je toplo!") #kod
7   else:
8       print("Vani je hladno!")
9       #sada je jasno da se ovaj else odnosi na uvjet2
```

Slika 16 Primjer ispravnog else-a

Petlja se izvršava jedan put više ili jedan put manje

Slika 17 pokazuje primjer gdje će se petlja izvršiti do broja 49, odnosno jednom manje od maksimalne vrijednosti, jer se drugi parametar u funkciji `range` gleda kao gornja granica koja se ne uključuje i petlja se završava neposredno prije te vrijednosti. Ako želimo da petlja uključuje gornju granicu moramo postaviti `maks_vrijednost + 1` kao drugi parametar u funkciji `range`.

C: > Document.py > ...

```
1   maks_vrijednost = 50
2   pocetak = 0
3   for i in range (pocetak, maks_vrijednost):
4       print(i)
```

Slika 17 Petlja se izvršava jednom manje

Višak koda u petlji

Primjer na Slici 18 prikazuje nam naredbu `print` koja nalazi unutar petlje i cijelo vrijeme se ispisuje nepotrebno. Naredba `print` bi se trebala nalaziti izvan `for` petlje i tako bi na kraju izlaz bio „Najveći broj je: 100“.

C: > Document.py > ...

```
1   niz_brojeva = [3, 29, 30, 41, 12, 9, 74, 15, 100]
2   max_vrijednost = niz_brojeva[0]
3
4   for br in niz_brojeva:
5       if br > max_vrijednost:
6           max_vrijednost = br
7           print(f"Najveći broj je: {max_vrijednost}")
8       if br <= max_vrijednost:
9           continue
10
11  #print(f"Najveći broj je: {max_vrijednost}")
```

```
#Ispis:
Najveći broj je: 29
Najveći broj je: 30
Najveći broj je: 41
Najveći broj je: 74
Najveći broj je: 100
Najveći broj je: 100
```

Slika 18 Višak koda u petlji

Pogrešan indeks elementa niza

U nizu poznanika pokušavamo pronaći tko ima najduže ime, ali zbog pogreške u kodu izlazimo izvan granica niza. Gdje dolazi do `IndexError`-a? U nizu poznanika imamo 5 elemenata s indeksima 0, 1, 2, 3, 4. Uvjet u liniji 5 omogućuje `i` da dosegne i 5 što je izvan niza jer `popis_poznanika[5]` ne postoji. Potrebno je promijeniti uvjet u `while` petlji iz „`<=`“ na „`i < len`“ da ne bismo izlazili izvan granica niza[3]. Slika 19 pokazuje primjer.

```
C: > Document.py > ...
1   popis_poznanika = ["Ana", "Antonija", "Ivana", "Marko", "Lana"]
2   i = 0
3   ime_len = ""
4   while i <= len(popis_poznanika):  # Pogreška je u uvjetu <=
5       if len(popis_poznanika[i]) > len(ime_len):
6           ime_len = popis_poznanika[i]
7       i = i + 1
8   print(f"Najduže ime: {ime_len}")
IndexError: list index out of range
```

Slika 19 Pogrešan indeks elementa niza

2.3 Greške u izvođenju

Pogreške u radu tijekom izvođenja programa (eng. run-time errors) su pogreške koje se događaju dok je program u nekoj operaciji. Neki od uzorka koji se često pojavljuju su aritmetičke pogreške, ne inicijalizirane varijable, nizovi indeksa izvan dometa i sl. Slika 20 je primjer koji pokazuje dohvaćanje elementa koji je izvan niza i naziva se IndexError.

```
C: > Document.py > ...
1   lista=[1,2,3]
2   element=lista[3]
3   element=lista[3]
  Traceback (most recent call last):
    File "c:\Document.py", line 2, in <module>
      element=lista[3]
              ~~~~~^^^
  IndexError: list index out of range
```

Slika 20 Primjer pogreške tijekom izvođenja programa

Pogreške u izvođenju su pogreške koje se pojavljuje tijekom izvršavanja programa, a uzrokuju prekid rada programa. Mogu biti izazvane iz različitih razloga poput dijeljenja s nulom, dohvaćanja elementa u listi koji ne postoji ili pokušaja otvaranja datoteke koja ne postoji. Provođenjem detaljnog testiranja i provjerom unesenih vrijednosti mogu se izbjeći ove pogreške. Najčešće poruke o greškama:

- **SyntaxError: invalid syntax** – pojavljuje se kada je sintaksa koda neispravna.

```
C: > Document.py > ...
1   ages = {
2       'pam': 24,
3       'jim': 24
4       'michael': 43
5   }
6   print(f'Michael is {ages["michael"]} years old.')
  SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

Slika 21 SyntaxError: invalid syntax

- **ZeroDivisonError: int or modulo by zero** – kada pokušamo dijeliti s nazivnikom koji ima vrijednost nula.

```
C: > Document.py
  1   print(10%0)
ZeroDivisionError: integer modulo by zero
```

Slika 22 ZeroDevisionError: integer modulo by zero

- **IndexError: list index out of range** – kada pokušavamo pristupiti indeksu koji je nevažeći. Obično jer indeks premašuje granice liste jer je prevelik.

```
C: > Document.py > ...
  1   lista_ime = ["Ivan", "Luka", "Darko"]
  2   print(lista_ime[5])
  IndexError: list index out of range
```

Slika 23 IndexError

- **ImportError** – kada interpreter nije u mogućnosti uvesti određeno ime ili modul unutar našeg koda.

```
C: > Document.py
  1   import mathx
ModuleNotFoundError: No module named 'mathx'
```

Slika 24 ImportError

- **NameError** – podiže se kada pokušamo pristupiti ili koristiti varijablu koja nije definirana ili kojoj nije dodijeljena vrijednost.

```
C: > Document.py > ...
  1   ime="Ivan"
  2   if Ime=="Ivan":
  3       print("True")
  4   else:
  5       print("False")
NameError: name 'Ime' is not defined. Did you mean: 'ime'?
```

Slika 25 NameError

- **AttributeError** – kada pokušamo pristupiti ili dodijeliti atribut koji ne postoji u klasi objekta.

```
C: > Document.py > ...
1   class Osoba:
2       def __init__(self, ime, prezime):
3           self.ime = ime
4           self.prezime = prezime
5
6   osoba1 = Osoba("Ana", "Anić")
7
8   print(osoba1.adresa)
AttributeError: 'Osoba' object has no attribute 'adresa'
```

Slika 26 AttributeError

- **Overflow** – pojavljuje se kada je eksponencijalni rast prevelik ili kada je vrijednost cijelog broja prevelika da bi se pretvorila u float vrijednost.

```
C: > Document.py > ...
1   large_integer = 10**1000
2   result = float(large_integer)
3   print(result)
OverflowError: int too large to convert to float
```

Slika 27 Overflow

- **Illegal function call**

```
C: > Document.py > ...
1   import random
2
3   totals = {4:0, 5:0, 6:0, 7:0, 8:0}
4
5   for _ in range(1000):
6       decades = random.choice((4,5,6,7,8))
7       totals(decades) += 1
8
SyntaxError: 'function call' is an illegal expression for augmented assignment
```

Slika 28 Illegal function call

- **Type mismatch** – neusklađenost tipova, očekuje se određeni tip podatka, ali umjesto toga dobije se drugi tip. Ova pogreška obično nastaje kada se vrši dodjela vrijednosti između nekompatibilnih podatka[9]. Primjer:

```
C: > Users > PCž > Desktop > python.py > ...
1   godine=input("Koliko godina imate: ")
2
3   buduce_godine=godine+10
4
5   print(buduce_godine)
TypeError: can only concatenate str (not "int") to str
```

Slika 29 TypeError

2.4 Koraci u otklanjanju pogrešaka

U nastavku ovog poglavlja istražit ćemo ključne korake za učinkovito otklanjanje pogrešaka. Prvo ćemo se fokusirati na identifikaciju pogrešaka i kako ih pravilno prepoznati. Zatim ćemo prijeći na proces replikacije, odnosno kako točno ponoviti uvjete pod kojima je greška nastala. Nakon toga, obradit ćemo metode razumijevanja programa i grešaka koje se pojavljuju, kako bismo mogli temeljitije pristupiti rješavanju problema. U daljnjim odjeljcima bavit ćemo se pronalaženjem i otklanjanjem grešaka, kao i tehnikama za učenje iz tih situacija, kako bi se spriječile buduće pogreške.

Identifikacija

Prvi korak u otklanjanju pogreške je njena identifikacija. To uključuje razumijevanje simptoma problema, kao što su poruke o greškama ili neobično ponašanje programa. Pogreške koje zaustavljaju program su uobičajeno jednostavnije za otkrit, za razliku od onih koje dopuštaju da se program izvrši do kraja s izlazom koji je pogrešan. Uklanjanjem pogreške, ali ne i njeno razumijevanje, ne dovodi do rješavanja problema te pogreške, jer istu možemo ponovit lako u daljnjem pisanju koda. Prije nastavljanja postavljamo dva pitanja:
Što je program napravio? Što smo očekivali da će program napraviti?

Replikacija

Drugi korak u otklanjanju pogreške je njena replikacija. Nakon identifikacije pogreške, važno je moći reproducirati istu. To znači ponovno stvaranje istih uvjeta koji su doveli do pojave pogreške. To radimo da bi razumjeli što točno uzrokuje grešku, jer ta se ista greška može pojaviti iz dva različita razloga.

Razumijevanje programa

Treći korak u otklanjanju pogreške je razumijevanje programa. Potrebno je dublje razumijevanje funkcioniranja programa kako bismo identificirali potencijalne uzroke pogreške. Pregled logike programa, proučavanje algoritama i razumijevanje različitih dijelova koda te postavljanje točaka prekida za pregled izvođenja programa nam može pomoću u razumijevanju rada programa.

Razumijevanje pogreške

Četvrti korak u otklanjanju pogreške je razumijevanje pogreške. Nakon što smo uspješno identificirali i replicirali grešku važno je otkriti što je uzrokuje i kako utječe na tijek izvršavanja programa.

Pronalaženje pogreške

Nakon razumijevanja pogreške, sljedeći peti korak je pronalaženje pogreške u programu. Možemo je tražiti na dva mjesta: u kodu koji vidno uzrokuje netočno ponašanje i u kodu koji je stvarno netočan. Uobičajeno su to isti dijelovi koda, ali postoje slučajevi i kad nisu.

Otklanjanje pogreške

Šesti korak je otklanjanje pogreške. Nakon pronalaska pogreške potrebno je ispraviti. Svaki ispravak je potrebno testirati i utvrditi da se greška više ne ponavlja. Ispravak može uključivati promjenu koda, popravak sintakse pogreške ili logike rada programa i sl.

Učenje od pogreške

Kada riješimo problem pogreške važno je izvući pouku iz nje kako bismo spriječili da se isti problem ponavlja u budućnosti[3].
Pokazati ćemo kako se otklanja pogreška u primjeru na Slici 30 koristeći prethodno navedene korake:

C: > Document.py > ...

```
1  def kalkulator_prosjeka(num):
2      ukupno = sum(num)
3      brojac = len(num)
4      average = ukupno / brojač
5      return average
6
7  podaci = [10, 20, 30, 40, 50]
8  print("Average:", kalkulator_prosjeka(podaci))
```

Slika 30 Koraci u otklanjanju grešaka - primjer

1. Identifikacija:

   Prilikom pokretanja programa, dobivamo poruku o pogrešci.

   ```
   >>>NameError: name 'brojač' is not defined. Did you
   mean: 'brojac' ?
   ```

2. Replikacija:

Ponovnim pokretanjem programa rezultira istom greškom, što potvrđuje problem.

3. Razumijevanje programa:

   Program računa prosječnu vrijednost brojeva u listi „podaci“

4. Razumijevanje pogreške:

   Greška nam govori da varijabla „brojač“ nije definirana i da je pogrešno napisana.

5. Pronalaženje pogreške:

   Pregledavamo kod kako bi našli pogrešku. Nalazi se u liniji 4:

   ```
   average = ukupno / brojač
   ```

6. Otklanjanje pogreške:

   Ispravljamo varijablu „brojač“ na brojac

Provjerimo da se greška opet ne ponavlja nakon otklanjana i nastavljamo s kodiranjem.

2.5 Program za otkrivanje grešaka (eng. debugger)

Debuggiranje (eng. debugging) je proces otklanjanja pogrešaka iz programa. Program za otkrivanje pogrešaka ima nekoliko metoda kojima traži pogreške u programu.

Linija po linija (eng. stepping)

Metoda linija po linija (engl. stepping) u programu za otkrivanje grešaka omogućava korisnicima da prate izvršavanje koda korak po korak, liniju po liniju. Kada koristimo ovu metodu, program će se zaustaviti nakon izvršavanja svake linije koda, omogućavajući nam da pratimo što se točno događa u svakom trenutku, te da precizno možemo odredimo gdje se greška dogodila. Imamo više opcija koje možemo koristi:

- **Step into** - pokreće program, liniju po liniju, tako da možete vidjeti točno što svaka instrukcija radi.
- **Step over** – opcija također izvršava sljedeću liniju koda, ali preskače ulazak u bilo koju funkciju ili metodu koja se može pozvati na toj liniji. Ovo je korisno kada želite brzo pratiti glavni tok izvršavanja programa, a ne ulaziti u detalje svake funkcije ili metode.
- **Step out** - će izvršiti preostali kod u trenutnoj funkciji ili metodi i zaustaviti se na sljedećoj liniji koda izvan te funkcije ili metode. Ovo je korisno kada ste već ušli u funkciju ili metodu koristeći "Step into", ali želite se vratiti na glavni tok programa.

Traganje (eng. tracing) od točke prekida (eng. breakpoint)

Metoda "Traganje od točke prekida" (eng. tracing from breakpoint) omogućava korisnicima da prate izvršavanje koda od određene točke prekida (engl. breakpoint). Ovo je korisno jer omogućava praćenje izvršavanja koda samo od trenutka koji nas zanima, umjesto da pratimo liniju po liniju od samog početka programa.

Promatranje (eng. watching)

Metoda "Promatranje" (eng. watching) omogućava korisnicima da prate vrijednosti određenih izraza ili varijabli tijekom izvršavanja programa. Kada postavite "watchpoint" na određenu varijablu ili izraz, izvršavanje programa će se zaustaviti svaki put kad se vrijednost te varijable ili izraza promijeni[3].

Princip rada Python debugger-a?

Python debugger pokrećemo odabirom opcije Debug->Debuger u alatnoj traci–u razvojnim alatima ili IDE-u. Neke od osnovnih naredbi su:

Naredba Go: izvrši program normalno

„Go“ ili „Continue“ naredba u debuggeru označava da se nastavlja izvršavanje programa do sljedeće točke prekida ili do kraja programa ako nema drugih točaka prekida.

Naredba Step: izvrši samo sljedeću instrukciju

„Step“ naredba izvršava trenutnu liniju koda i prelazi na slijedeću liniju koda unutar iste funkcije. Debugger ulazi u funkciju i nastavlja izvršavati liniju po liniju unutar te funkcije.

Naredba Over: slično kao step, ali ne ulazi u funkcije

„Over“ ili „Step over“ slična je naredbi „Step“, ali za razliku od naredbe „Step“ naredba „Over“ ako se poziva funkcija, ne ulazi u tu funkciju već je preskače.

Naredba Out: slično kao go, ali se može nastaviti sa step

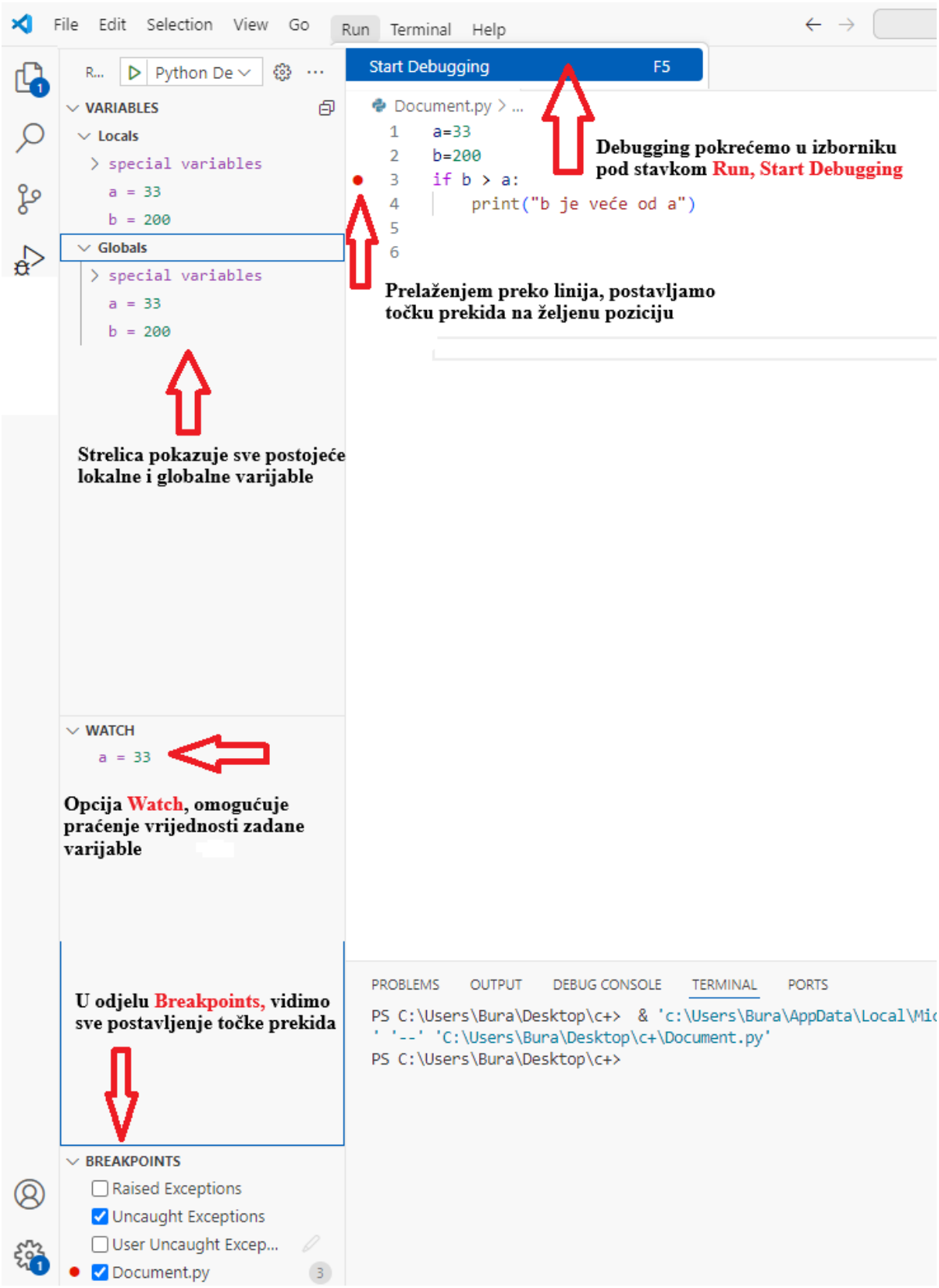
„Out“ naredba koristi se kada ste unutar funkcije i želimo završiti izvršavanje te funkcije. Korisno kada želite brzo izaći iz trenutne funkcije i nastaviti s analizom koda izvan te funkcije.

Naredba Quit: prekini izvršavanje

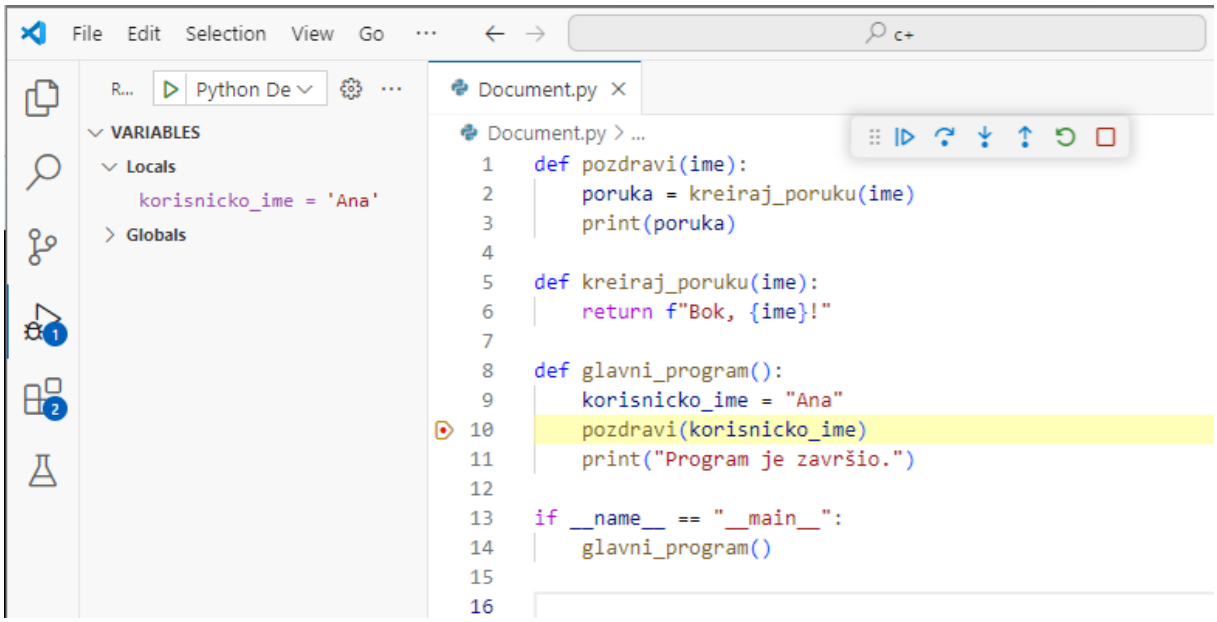
Naredba „Quit“ zaustavlja debug session i izlazi iz načina rada debuggiranja.

Primjer rada Python debugger-a

Na prvoj slici vezanoj za debugger, odnosno Slici broj 31 prikazano je kako se debugger pokreće u razvojnom okruženju, te kako se točka prekida može jednostavno postaviti klikom uz liniju koda. Vidljiva točka prekida (engl. breakpoint) omogućuje zaustavljanje izvršavanja programa na određenim linijama koda, kao i popis varijabli koje prikazuju njihove trenutne vrijednosti tijekom debuggiranja. Također, tu je opcija "Watch" koja omogućuje praćenje specifičnih varijabli ili izraza. Primjer prikazan kroz slike u nastavku detaljnije objašnjava i prikazuje opcije Step Into, Step Over, Step Out.

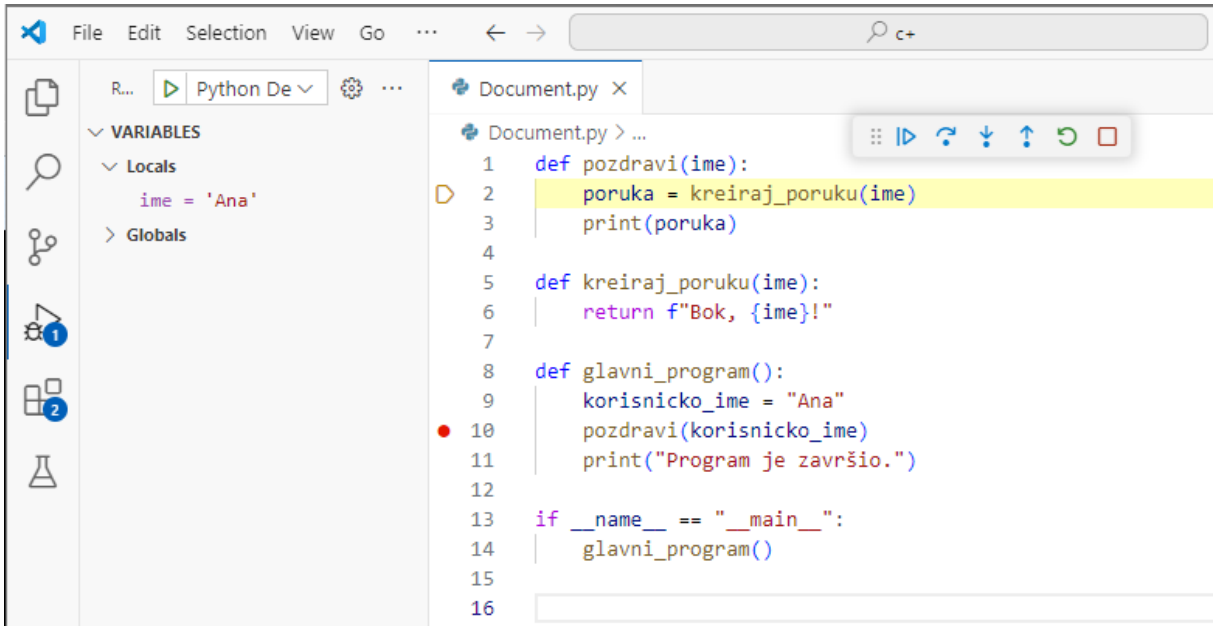


Slika 31 Korištenje Python Debuggera



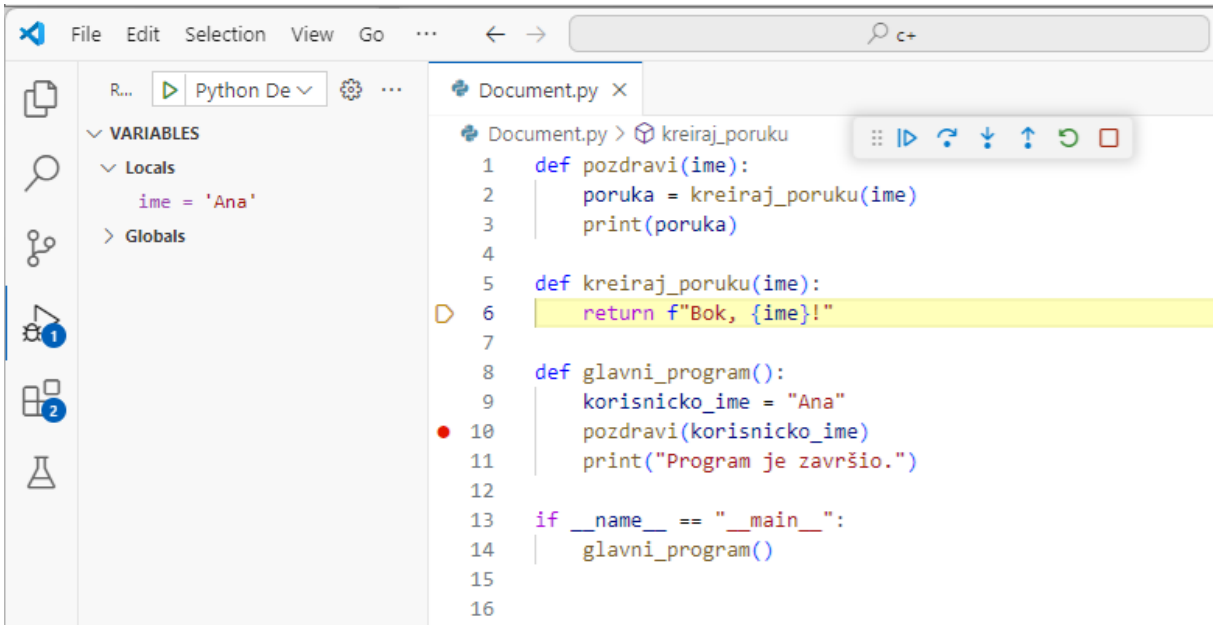
Slika 32 Postavljanje točke prekida

Postavljamo točku prekida u funkciji `glavni_program` na liniju gdje se poziva funkcija `pozdravi(korisnicko_ime)`. Upotrebom opcije `Step Into (F11)` ulazimo u funkciju pozdravi.



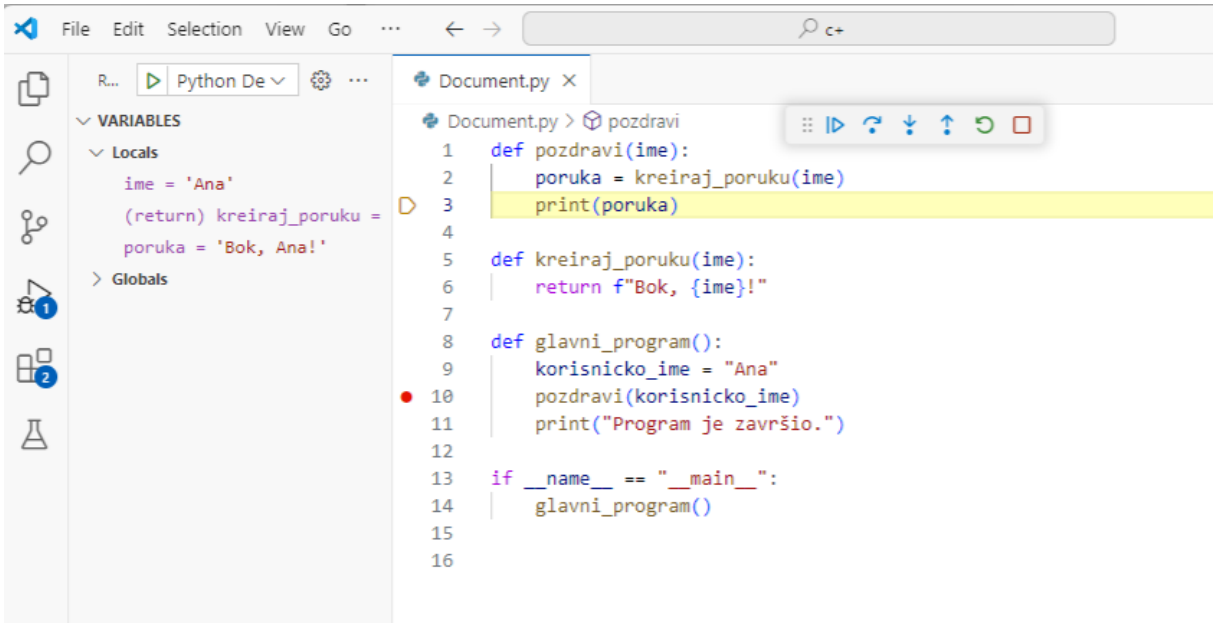
Slika 33 Ulazak u funkciju s Step Into

Kada smo u funkciji pozdravi i želimo vidjeti što se događa unutar funkcije `kreiraj_poruku`, tada opet koristimo `Step Into (F11)`. To će ući unutar funkcije `kreiraj_poruku`.



Slika 34 Povratak s Step Over

Kada smo unutar funkcije `kreiraj_poruku`, ako koristimo `Step Over (F10)` na liniji `return f"Hello, {name}!"`, preskočit ćemo ulazak u bilo kakve funkcije unutar te linije (ako ih ima) i vraća nas na funkciju koja je pozvala `kreiraj_poruku`, odnosno natrag u funkciju pozdravi.



Slika 35 Povratak s Step Out

Dok smo unutar funkcije `pozdravi` možemo upotrijebiti `Step Out (Shift + F11)` kako bismo izašli iz te funkcije i vratili se na mjesto odakle je funkcija pozvana `glavni_program`.

`Continue (F5)` nastavlja izvršavanje programa do kraja ili sljedeće točke prekida.

3 IZNIMKE U PYTHONU

Pogreške koje se aktiviraju prilikom izvršavanja programa nazivaju se iznimkama (engl. exceptions). Kod modernijih programskih jezika, pa tako i kod Pythona, imamo ugrađene mehanizme pomoću kojih je na jednostavan način moguće obraditi iznimke. U Pythonu, iznimke su organizirane hijerarhijski tako da ta struktura omogućuje razvrstavanje iznimki po njihovim sličnostima i zajedničkim karakteristikama. Glavni cilj hijerarhijske strukture je da olakša programerima rukovanje i obradu iznimki. Sve iznimke potječu od osnovnog razreda „BaseException“ koji je roditelj svim ostalim iznimkama. Ovaj razred ima nekoliko izravnih podrazreda, jedan od njih je „Exception“, on je roditeljski razred iznimkama koje najčešće ne uzrokuju izlaz iz programa i tipa iznimke s kojima se programeri najčešće susreću. Kao što je iz slike 36 moguće vidjeti, postoje brojne iznimke koje su složene u smislene cjeline, čime se programeru omogućava pristup i obrada iznimke na različite načine. Na primjer, podrazredi `IndexError`, `KeyError` pripadaju razredu `LookupError`. Ako se u programskom kodu hvata iznimka `IndexError`, tada se neće uhvatiti iznimka `KeyError`, no ako se hvata iznimka `LookupError` tada će se uhvatiti bilo koja od dviju iznimaka koje pripadaju razredu `LookupError`[2].

3.1 Primjeri iznimki

U Tablici 2 se nalaze neke od najčešćih iznimki koje se pojavljuju.

Tablica 2 Prikaz najčešćih iznimki

Iznimka	Opis
ZeroDivisonError	Ovaj tip pogreške se podiže kad se pokušava dijeliti s nazivnikom kojemu je vrijednost nula.
IndexError	Ovaj tip pogreške podiže se kada operacija prima argument iz liste, a vrijednost na željenom indeksu nije definirana.
ImportError	Ovaj tip pogreške podiže se kada interpreter nije u mogućnosti uvesti određeno ime ili modul unutar našeg koda.
NameError	Ovaj tip pogreške se podiže kada korištena varijabla nije definirana, tj. kada joj nije pridružena neka konkretna vrijednost.
AttributeError	Ovaj tip pogreške podiže se kada se nad nekim objektom poziva atribut koji ne postoji.
TypeError	Ovaj tip pogreške se podiže prilikom pokušaja izvršavanja operacije, funkcije ili metode nad varijablom pogrešnog tipa.

Slika 36 Prikaz hijerarhije iznimki

Rad s iznimkama uključuje prepoznavanje, hvatanje i obradu iznimki koje se mogu pojaviti tijekom izvođenja programa, te nam omogućava rješavanje situacija u kojima se javljaju neočekivane pogreške tijekom izvođenja koda. To je ključna komponenta robusnog programiranja koja pomaže spriječiti da se cijeli program sruši prilikom nailaženja na problem ili grešku.

3.2 Obrada iznimki

Sve iznimke spomenute u prethodnim poglavljima zaustavljaju daljnje izvršavanje programa, te ako ih se ne uhvate i obrade, same po sebi nisu korisne. Stoga postoji naredba `try … except` koja omogućava hvatanje i obradu iznimki (engl. exception handling). Naredba `try` sadrži više slojeva kao `if` ili `while` i slijedi pravila uvlačenja[4]. Slika 37 prikazuje sintaksu naredbe `try`.

```
try:
     <tijeloTryNaredbe>

except <iznimka1>:
     <tijeloExceptNaredbe1>
```

Slika 37 Sintaksa naredbe Try

Redoslijed izvršavanja naredbe `try`:

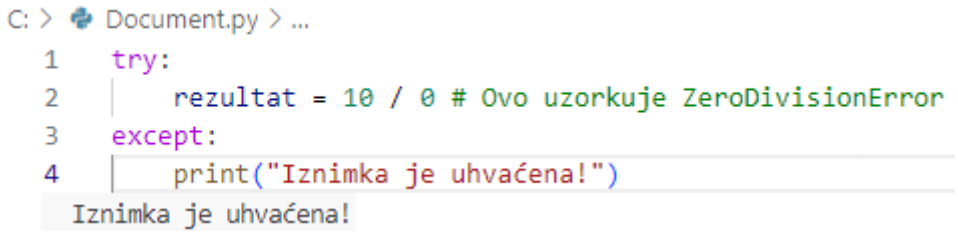
- ako se tokom izvršavanja ne pojavi iznimka odnosno `try` se uspješno izvrši, `except` linija se preskače i `try` završava, te se nastavlja izvršavanje koda koje slijedi.
- ako se iznimka pojavi tokom izvršavanja `try`-a, i ako odgovara iznimci navedenoj u except liniji, onda se ona izvršava .
- ako se iznimka pojavi, a ne odgovara `except` liniji provjeravaju se druge iznimke u `try` izrazu. Ako njih nema, program završava i pokazuje poruku o pogrešci[4].

Nakon uhvaćene iznimke možemo poduzeti neke od navedenih koraka za nastavak rada programa:

- **Obavijestiti korisnika**: Prikazati poruku koja opisuje grešku
- **Logirati grešku**: Zapisati grešku u log datoteku za kasniju analizu.
- **Ponoviti operaciju**: Pokušati ponovo izvršiti kod.
- **Završiti program sigurno**: Ako greška ozbiljno utječe na rad programa, možda će zahtijevati prekid rada programa.

Obrada svih iznimki

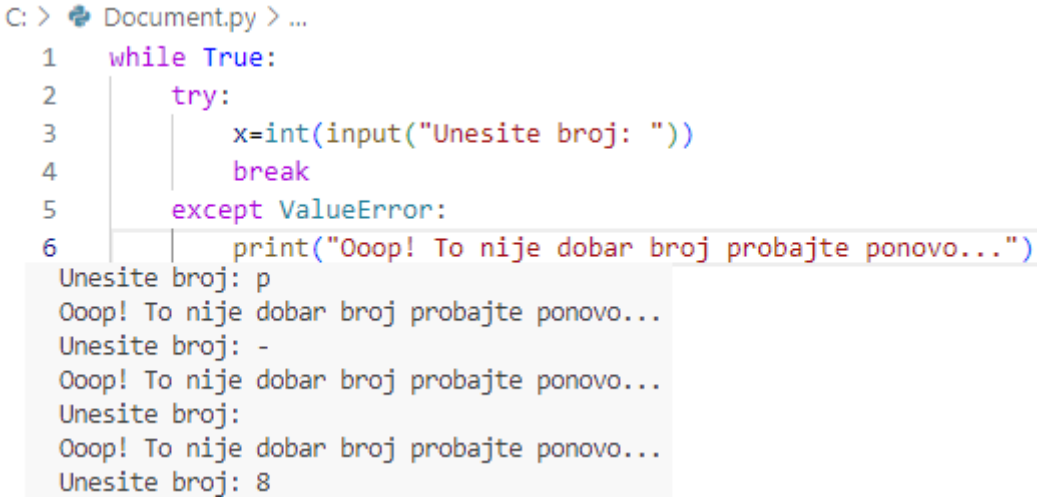
Da bi se obuhvatile sve iznimke, treba se specificirati odlomak `except` bez imena iznimke ili argumenta. Ovaj način korištenja `except` bloka i nije najbolje rješenje, jer obuhvaća sve iznimke i može prikriti stvarne pogreške korisničkog programa[2]. Sintaksa je prikazana u idućoj slici.



Slika 38 Primjer hvatanja svih iznimki

Obrada specifičnih iznimki

Pokazni primjer prikazuje Slika 39 u kojemu je uzrokovana iznimka `ValueError`. Unutar tijela naredbe `try` se nalazi varijabla `x`. Unošenjem bilo kojeg drugog tipa podatka koje nije cijeli broj (npr. slova p, znaka -, pritiskom tipke Enter) završava se naredba `try`, a pokreće se tijelo naredbe `except` i ispisiva se prikazana poruka. Unošenjem ispravnog tipa podataka odnosno cijelog broja, naredba `try` se završava bez podignute iznimke i tada se tijelo naredbe `except` preskače.



Slika 39 Prikaz rada naredbe try

Obrada više iznimki u jednom bloku

U jednom `except` bloku možemo definiranih više vrsta iznimki unutar n-terca, primjer Slika 40:

```
C: > Document.py > ...
1   ime ="Luka"
2   try:
3       rezultat=ime +1
4   except (RuntimeError, TypeError, NameError):
5       print("Pojavila se iznimka!")
Pojavila se iznimka!
```

Slika 40 Primjer više vrsta iznimki unutar except bloka

Slika 41 prikazuje primjer gdje se podiže iznimka `TypeError`, jer se pokušava zbrojit cijeli broj i varijabla „ime“ koja je tipa `string`. Izvođenje naredbe `try` se prekida i pokreće izvođenja tijela naredbe `except` te se ispisuje navedena poruka u primjeru. Na slici 41 također vidimo da iako smo u kodu napisali `try-except` naredbu svejedno nismo uhvatili iznimku i kod se prestao izvršavati. To se događa u slučaju kada se u naredbi `except` ne navede iznimka koja se pojavila i tad se prikazuje se `traceback` poruka i vrsta iznimke do koje je došlo. Da bi to izbjegli možemo na kraju navesti još jedan `except` blok bez specificirane iznimke koji nam služi za hvatanje svih iznimki.

```
C: > Document.py > ...
1   ime ="Luka"
2   try:
3       rezultat=ime +1
4   except (RuntimeError, NameError):
5       print("Pojavila se iznimka!")
Traceback (most recent call last):
  File "c:\Document.py", line 3, in <module>
    rezultat=ime +1
             ~~~~^~
TypeError: can only concatenate str (not "int") to str
```

Slika 41 Primjer kada except nije uhvatio iznimku

Višestruki except blokovi

Naredba `try` može imati više od jedne `except` linije (uvjeta), koji određuju rad za neke druge iznimke. Pri tome se najviše izvršava jedna od njih, i to samo za iznimke navedene u tom dijelu `except` linije, ali ne za druge. Primjer sintakse prikazan je na sljedećoj slici:

```
try:
     <tijeloTryNaredbe>

except <iznimka1>:
     <tijeloExceptNaredbe1>

except <iznimka2>:
     <tijeloExceptNaredbe2>

except:
     <tijeloExceptNaredbe>
```

Slika 42 Sintaksa s više except blokova

Primjer na Slici 43 pokazuje kod koji se sastoji od tri `except` bloka, dvaju blokova s precizirano traženim iznimkama i trećeg bloka od same naredbe `except` koja, ako dođe do ikakve dodatne iznimke uhvati je i ispiše poruku.

```
C: > Document.py > ...
1   niz = [0,1,2]
2   try:
3       print(niz[3])
4   except NameError:
5       print("NameError: Pokušali ste koristiti varijablu koja nije definirana!")
6   except IndexError:
7       print("IndexError: Pokušali ste pristupiti indeksu koji je izvan niza!")
8   except:
9       print("Upisani podaci se ne poklapaju")
IndexError: Pokušali ste pristupiti indeksu koji je izvan niza!
```

Slika 43 Primjer except naredbe bez preciziranje iznimke

3.3 Bacanje iznimki

Do sada sve pokazane iznimke su podignute od strane Pythona (npr. `TypeError`). Python ne mora biti taj koji uvijek podiže iznimku i u ovom poglavlju ćemo pokazati kako mi možemo bacati (engl. throw ili dizati engl. raise) iznimku. Iznimke od strane programera se podižu kada on predvidi da će se u nastavku koda za neke ulazne vrijednosti potencijalno dogoditi pogreška[2]. Iznimke bacamo ključnom riječi `raise` nakon koje trebamo navesti objekt koji predstavlja iznimku koju želimo prijaviti. Unutar tog objekta mogu biti sadržane dodatne informacije o podignutoj iznimci[5]. Primjer izgleda sintkase:

```
raise ImeIznimke (<dodatneInformacije>)
```

U primjeru prikazanom na Slici 44 pišemo funkciju koja izračunava kvadratni krojen broja. Želimo biti sigurni da ulazni broj nije negativan jer kvadratni korijen negativnog broja nije definiran u realnim brojevima. Ako je ulazni broj negativan kao u primjeru podiže se

`ValueError` s porukom „`Ulazni broj mora biti ne negativan`“. Ukoliko je pozitivan ispisuje se korijen toga broja.

```
C: > Document.py > ...
1   import math
2
3   def kvadratni_korijen(broj):
4       if broj < 0:
5           raise ValueError("Ulazni broj mora biti ne-negativan!")
6       return math.sqrt(broj)
7
8   # Pokušaj s nevalidnim ulazom
9   try:
10      rezultat = kvadratni_korijen(-4)
11      print(f"Kvadratni korijen od -4 je {rezultat}")
12  except ValueError as e:
13      print(e)
```

Slika 44 Naredba raise

Zbog važnosti automatskog testiranja, veliki broj jezika za takvu namjenu ima definiranu posebnu naredbu potvrde `assert`. Naredba prima dva argumenta: uvjet i izraz, a izraz sadrži `bool` vrijednost koja ako je `false` (neistina) označava pogrešku. Prvo se gleda uvjet naredbe koji je uvijek jednak jedinici odnosno istinit. Ako je rezultat uvjeta istinit program nastavlja s izvođenjem. Ako je uvjet izračunat kao neistina (`false`), onda Python podiže `AssertionError`[5].

Sintaksa naredbe:

`assert expr [,arg]`

`expr` je izraz koji se provjerava, a može biti `true` ili `false`. Ako je `false` onda se podiže `AssertionError` s argumentom `arg`. Ako je izraz istinit onda naredba `assert` ne poduzima ništa. Primjer naredbe `assert`:

```
C: > Document.py > ...
1   def zbroji_pozitivne_brojeve(a, b):
2       assert a > 0 and b > 0, "Oba broja moraju biti pozitivna"
3       return a + b
4
5   print(zbroji_pozitivne_brojeve(3, 5))  # Ovo će raditi
6   print(zbroji_pozitivne_brojeve(-3, 5))  # Ovo će podići AssertionError
```

Slika 45 Naredba assert

Naredba `assert` dozvoljava samo pozitivne brojeve, a linija 6 sadrži negativan broj -3 i iz tog razloga naredba `assert` podiže `AssertionError` i ispisuje:

`>>>AssertionError: Oba broja moraju biti pozitivna`

Naredba `assert` je dizajnirana za debuggiranje i testiranje u ranim fazama programa i nije zamjena za uobičajeno rukovanje iznimkama pomoću `try-except`.

3.4 Zaključna stavka naredbe za obradu iznimki

Ponavljanje istog niza naredbi u načelu izbjegavamo, jer se radi o plodnom tlu za unošenje suptilnih pogrešaka u program. Ovo ponavljanje se može izbjeći zadavanjem zaključnog stavka naredbe za obradu iznimke. Zaključni stavak zadajemo ključnom rječju `finally` i on se uvijek mora navesti pri kraju naredbe za obradu iznimke, dakle ispod svih stavka za hvatanje iznimaka[5]. Primjer sintakse je prikazan na slici ispod:

```
Try:
        <kod_kojeg_pokušavamo_izvesti>
except:
        <kod_za_obradu_iznimke>
finally:
        <kod_koji_se_uvijek_izvede_na_kraju>
```

Slika 46 Sintaksa zaključnog stavka

Tijelo `finally` se izvodi kao posljednja stavka i to bez obzira dogodila se iznimka ili ne, također ako i nije uhvaćena. Ako `data.txt` ne postoji ispisati će se prvo tijelo naredbe except, a zatim tijelo zaključnog stavka. Ako `data.txt` postoji ispisat će se ono što se nalazi u `txt` datotetci, a zatim i tijelo zaključnog stavka. Ukoliko nam `except` nije obradio iznimku, iznimka će nam se svejedno podići nakon izvršavanja tijela `finally`.

zavrsni.py > ...

```
1
2   try:
3       file = open("data.txt", "r")
4       # Obrada podataka iz datoteke
5       content = file.read()
6   except FileNotFoundError:
7       print("Datoteka nije pronađena.")
8   finally:
9       file.close()
10      print("Ovo se uvijek izvodi!")
```

Slika 47 Zaključna stavka – finally

Zaključni stavak omogućava izražavanje „čistačkog“ koda koji se izvodi u svim ishodima izvođenja programskog odsječka koji može baciti iznimku[5]. Vrlo koristan kod oslobađanja ili vraćanja zauzetih resursa natrag operacijskom sustavu ili okolini. Zatvaranjem otvorenih datoteka nakon što smo završili korištenje iste, zatvaranje mrežnih resursa i sl. Međutim,

problem je što sve to moramo odraditi mi „ručno“. Ključnom rječju `with` rješava se problem ručnog obavljanja, odnosno pomoću nje automatski se otvara i zatvara korišteni resurs. Najčešće se odnosi na rad s datotekama, mrežnim vezama, bazama podataka i sl.

U primjer datoteka `data.txt` se automatski zatvara nakon čitanja, što znači da se ne treba ručno pozivati `file.close()`.

```
with open("data.txt", "r") as file:
     content = file.read()
     print(content)
```

3.5 Puni oblik naredbe za obradu iznimke

Unutar `try – except` strukture se može koristi i alternativni stavak regularne obrade. Ključnom rječju `else` zadajemo alternativni stavak, a njegovo tijelo se jedino izvodi ukoliko izvođenje tijela `try` ne rezultira podizanjem iznimke ili ako ne završava jednom od naredbi `return, break i continue`[5].

```
try:
    # ovdje navodimo dio regularne obrade
    # u kojem želimo hvatati iznimke

except <Iznimka1> as iznimka:
    # Obrada specifične iznimke tipa <Iznimka1>

except (<Iznimka1>, <Iznimka2>) as iznimka:
    # Obrada specifičnih iznimki tipa <Iznimka1> i <Iznimka2>

except <Iznimka> as e:
    # Opća obrada za sve ostale neočekivane iznimke

else:
    # ovdje navodimo dio regularne obrade
    # u kojem ne želimo hvatati iznimke

finally:
    # Ovaj blok se uvijek izvršava, bez obzira na to je li došlo do iznimke
```

Slika 48 Puni oblik naredbe za obradu iznimke

4 ZAKLJUČAK

Pogreške su neizbježan dio razvoja programske podrške, a njihovo učinkovito upravljanje ključno je za održavanje stabilnosti i pouzdanosti sustava. Razumijevanje različitih vrsta pogrešaka omogućuje programerima da identificiraju i riješe probleme prije nego što utječu na korisničko iskustvo ili funkcionalnost aplikacije. Korištenjem odgovarajućih tehnika za otklanjanje pogrešaka, poput debuggiranja programeri mogu značajno smanjiti broj pogreški i poboljšati kvalitetu svog koda. Primjena ovih tehnika omogućava razvoj robusnijih i pouzdanijih aplikacija koje su bolje pripremljene za suočavanje s problemima u stvarnom svijetu. Efikasno upravljanje pogreškama ne samo da poboljšava iskustvo krajnjih korisnika, već i doprinosi održivosti i dugovječnosti programskih rješenja. S obzirom na sve ove aspekte, pravilna primjena mehanizama za upravljanje pogreškama predstavlja ključan dio procesa razvoja programske podrške i neophodan korak ka postizanju visokih standarda kvalitete.

POPIS SLIKA

POPIS TABLICA

LITERATURA

1) Adams, Vicki Porter, Captain Grace M. Hopper: the Mother of COBOL // InfoWorld October 5, 1981,33 str
2) Marko Hruška, Programiranje u Pythonu; Priručnik za polaznike, Srce, Sveučilište u Zagrebu, 2019.
3) Grubišić, Ani, "Osnove programiranja i algoritamskog razmišljanja: skripta sa zbirkom zadataka", interna skripta za kolegij Programiranje 1 na Prirodoslovno-matematičkom fakultetu u Splitu, 2021.
4) Mario Essert, Domagoj Ševerdija, Ivan Vazler: Digitalni udžbenik Python; Odjel za matematiku Sveučilišta Josipa Jurja Strossmayera Osijek, 2007.
5) Zoran Kalafatić, Antonio Pošćić, Siniša Šegvić, Julijan Šribar, Python za znatiželjne; Element, 2016
6) Trey Hunner: https://www.pythonmorsels.com/syntaxerror-invalid-syntax/ (zadnji put pristupljeno 23.8.2024.)
7) Ankthon: https://www.geeksforgeeks.org/precedence-and-associativity-of-operators-in-python/ (zadnji put pristupljeno 25.8.2024)
8) Nishab Harti: https://www.geeksforgeeks.org/runtime-errors/ (zadnji put pristupljeno 17.8.2024)
9) Serhiy Storchaka: https://docs.python.org/3/library/exceptions.html (zadnji put pristupljeno 13.8.2024)