

Profajliranje Python programa korišćenjem modula *cProfile*

Seminarski rad u okviru kursa
Verifikacija softvera,
Matematički fakultet

Marija Bogavac, 1068/2023

marijabogavac001@gmail.com

25. januar 2024.

Sažetak

Profajleri su alati koji mere prostornu ili vremensku složenost programa, upotrebu određenih instrukcija ili učestalost i trajanje funkcijskih poziva. *cProfile* je profajler za programski jezik Python. Dalje u radu proučićemo detaljnije šta je profajliranje, čemu služi, o implementaciji *cProfile*, upotrebi i razlikama u odnosu na druge Python profajlere.

Sadržaj

1	Uvod	2
2	Instrumentalizacija i uzorkovanje	2
3	Determinističko i statističko profajliranje	2
4	Python i profajliranje	3
5	Šta je <i>cProfile</i> i kako radi	3
6	Upotreba <i>cProfile</i>	4
6.1	Primer	4
6.2	Primer	5
6.3	Primer	6
6.3.1	Vizualizacija pomoću alata TUNA	7
6.3.2	Vizualizacija pomoću alata graphviz	9
6.4	Primer	9
6.4.1	Vizualizacija alatom SnakeViz	10
7	Poređenje sa ostalim Python profajlerima	11
8	Zaključak	12
	Literatura	12

1 Uvod

Profajliranje („programsko profajliranje“, „profajliranje softvera“) je oblik dinamičke analize programa koji meri prostornu ili vremensku složenost programa, upotrebu određenih instrukcija ili učestalost i trajanje funkcijskih poziva.

Danas programeri pišu hiljade linija koda za nekoliko dana. Kompleksnost novih programa i aplikacija stalno se razvija i kodovi uključuju više funkcija, od kojih neke mogu usporiti performanse celog programa. Identifikovanje delova koda koji rade sporo i potencijalnih uskih grla, a zatim optimizacija ovog koda može značajno poboljšati performanse softvera, smanjiti upotrebu memorije i potrošnju resursa. U programiranju, usko grlo se odnosi na deo u kodu u kome su performanse značajno ograničene ili usporene, što često uzrokuje slabljenje ukupne efikasnosti programa.

Ovo su neki uobičajeni tipovi uskih grla i strategije za njihovo rešavanje:

1. Usko grlo CPU-a:
 - Identifikacija: Visoka upotreba CPU-a ukazuje da je procesor usko grlo.
 - Rešenje: Optimizacija algoritama, korišćenje efikasnije strukture podataka ili paralelizovanje zadataka da bi se bolje iskoristili dostupni CPU resursi.
2. Algoritamsko usko grlo:
 - Identifikacija: Neefikasni algoritmi koji uzrokuju sporo izvršenje.
 - Rešenje: Analiziranje i optimizovanje algoritama za bolju vremensku složenost, korišćenje efikasnijih algoritama ili paralelne obrade gde je to moguće.
3. Neefikasne strukture podataka:
 - Identifikacija: Neefikasne strukture podataka mogu dovesti do sporih operacija, na primer ako se posmatra da li element pripada strukturi, bolje je koristiti skupove i mape jer je složenost $O(1)$ umesto listi gde bi složenost pretrage bila $O(n)$; u slučaju da je potrebno brisanje elemenata bolje je koristiti povezane liste umesto nizova
 - Rešenje: Važno je izabrati strukturu prema potrebama specifičnog zadatka i posmatrati vremensku složenost pri raznim operacijama.

2 Instrumentalizacija i uzorkovanje

Instrumentalizacija predstavlja ubacivanje novog koda u postojeći kôd s ciljem da se omogućí prebrojavanje određenih događaja odnosno prikupljanje podataka koji su od interesa. [8] Ovaj dodatni kôd omogućava profajleru da meri aspekte kao što su pozivi funkcija, vreme izvršenja i upotreba memorije. Može se vršiti ručno ili automatski pomoću alata.

Profajliranje zasnovano na uzorku podrazumeva periodično uzimanje “slika” tokom izvršavanja programa kako bi se prikupile informacije o njegovom stanju. Tokom svakog prekida, profajler beleži stek poziva funkcije i druge relevantne podatke. Profajliranje zasnovano na uzorku je efikasna metoda za analizu ponašanja programa bez značajnog uticaja na vreme njegovog izvršavanja, ali je manje precizna od instrumentalizacije.

3 Determinističko i statističko profajliranje

Determinističko profajliranje odražava činjenicu da se prate svi događaji poziva funkcije, povratka u funkciju i izuzetaka i precizno se određuju intervali između ovih događaja i za koje vreme se izvršava kôd.

Nasuprot tome, statističko profajliranje nasumično uzorkuje i zaključuje gde se vreme troši. Tradicionalno uključuje manje troškove (pošto kôd ne treba da bude instrumentalizovan), ali pruža samo relativne indikacije o tome gde se vreme troši. [1]

4 Python i profajliranje

Zašto je pogodno profajlirati u Python-u?

- Python dolazi sa ugrađenim modulima za profajliranje kao što su *cProfile* i *profile* koji olakšavaju analizu performansi koda bez potrebe za alatima treće strane.
- Python može lako da se integriše sa spoljnim alatima za profajliranje, kao što su *Py-Spy*, *line_profiler* i drugi.
- Python ekosistem ima razne alate i biblioteke za profajliranje nezavisnih proizvođača koje zadovoljavaju različite potrebe. Na primer, *memory_profiler* za profajliranje korišćenja memorije, *snakeviz* za vizualizaciju rezultata *cProfile* i *Py-Spy*. Python zajednica aktivno razvija i održava različite alate za profajliranje. To znači da je moguće pronaći alat ili biblioteku koji odgovara nekim specifičnim potrebama profajliranja i verovatno će biti dobro podržan i dokumentovan.
- Python ima više implementacija, kao što su CPython, Jython i PyPy. Svaka implementacija može imati različite karakteristike performansi. Profajliranje može pomoći optimizaciji koda za određenu implementaciju ako je potrebno.
- Python se široko koristi u naučnom računarstvu, analizi podataka i mašinskom učenju. Alati za profajliranje su neophodni u ovim domenima za optimizaciju koda koji je kritičan za performanse, a Python obezbeđuje neophodnu infrastrukturu za takve optimizacije.

O implementacijama

- CPython je podrazumevana i najčešće korišćena implementacija Python-a. Napisan je u C-u. Performanse CPython-a su generalno dobre za većinu aplikacija. Veoma je kompatibilan sa Python bibliotekama i ekstenzijama napisanim u C.
- Jython je implementacija Python-a koja radi na Java virtuelnoj mašini (JVM). Omogućava da se Python kod besprekorno integriše sa Java kodom, što obezbeđuje korišćenje Java biblioteka iz Python-a i obrnuto.
- PyPy je alternativna implementacija Python-a koja koristi Just-In-Time (JIT) kompajler za poboljšanje performansi. Cilj mu je da bude kompatibilan sa CPython-om i može ponuditi značajna poboljšanja brzine u odnosu na CPython, posebno za zadatke vezane za CPU.

5 Šta je *cProfile* i kako radi

cProfile je deterministički profajler za Python programe. To je ugrađeni modul u Python-u i koristi se za analizu performansi Python koda i identifikaciju uskih grla. Interno, *cProfile* koristi instrumentalizaciju za profajliranje Python programa. Korišćenjem funkcije `sys.setprofile()` dodeljuje se 'event listener' koji će biti obavešten kad god se neka od funkcija pozove.

Funkcije kao argumenti funkcija

Povratni poziv (callback) je mehanizam koji omogućava da se funkcija prosledi kao parametar, da bi kasnije bila pozvana po potrebi. [7]

Dok Python interpreter izvršava program, funkcija povratnog poziva koju je postavio *cProfile* se poziva za svaki poziv funkcije i povratak. Funkcija povratnog poziva beleži informacije kao što su vreme početka, vreme završetka i ime funkcije koja se poziva.

cProfile prikuplja podatke za profajliranje u strukturi podataka. Vodi evidenciju o vremenu provedenom u svakoj funkciji, broju poziva svakoj funkciji i drugim relevantnim informacijama.

Prikupljeni podaci se čuvaju u hijerarhijskoj strukturi podataka, gde je svaki poziv funkcije predstavljen kao čvor u stablu. Ovo omogućava detaljnu analizu hijerarhije poziva funkcija programa.

Nakon što program završi sa radom, prikupljeni podaci o profajliranju mogu se analizirati pomoću *stats* modula, koji je deo standardne biblioteke. Ovaj modul pruža funkcije za štampanje, sortiranje i manipulisanje statistikom profajliranja.

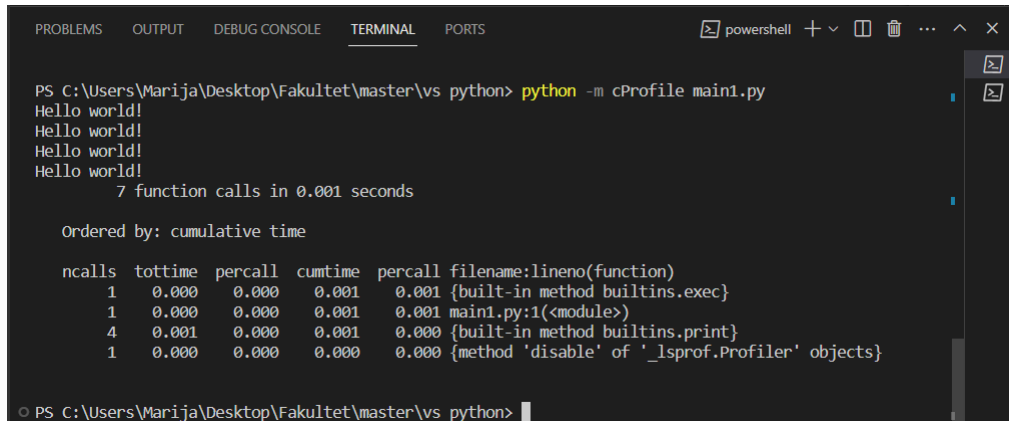
6 Upotreba *cProfile*

6.1 Primer

```
i=0
for i in range(4):
    print("Hello world!")
    i+=1
```

Listing 1: main1.py

Izvršavamo kôd u terminalu pokretanjem *cProfile*:



```
PS C:\Users\Marija\Desktop\Fakultet\master\vs python> python -m cProfile main1.py
Hello world!
Hello world!
Hello world!
Hello world!
 7 function calls in 0.001 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   1    0.000    0.000    0.001    0.001 {built-in method builtins.exec}
   1    0.000    0.000    0.001    0.001 main1.py:1(<module>)
   4    0.001    0.000    0.001    0.000 {built-in method builtins.print}
   1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Slika 1: Profajliranje main1.py korišćenjem *cProfile* u terminalu

Pokretanjem *cProfile* u terminalu (slika 1) dobili smo rezultat izvršavanja programa (“Hello world!” - ispisano 4 puta) i tabelu koja opisuje koje metode su pozvane, izvršene i njihovo trajanje. [2]

U fajlu main1.py izvršeno je 7 funkcijskih poziva za 0.001 sekundi.

Ordered by označava način na koji će biti sortirani podaci u tabeli. Mogu biti sortirani po kolonama **ncalls**, **filename**, **cumtime**.

Zaglavlja kolona tabele su:

1. **ncalls** – broj poziva funkcije
2. **tottime** (total time) – ukupno vreme provedeno u funkciji (bez vremena provedenog u podfunkcijama tj. u funkcijama koje poziva trenutna funkcija)
3. **percall** – prosečno vreme potrošeno po pozivu (odnos **tottime** i **ncalls**)
4. **cumtime** (cumulative time) – ukupno vreme potrošeno u trenutnoj funkciji i funkcijama koje ona poziva
5. **percall** – prosečno vreme potrošeno po pozivu (odnos **cumtime** i **calls**)
6. **filename:lineno(function)** – ime fajla, broj linije, ime funkcije u kojoj je napravljen poziv

Posmatrajmo poslednju kolonu:

- {built-in method builtins.exec} označava vreme provedeno u ugrađenom exec metodu, često povezano sa dinamičkim izvršavanjem koda.
- main1.py:1(<module>) prikazuje kumulativno vreme provedeno u kodu najvišeg nivoa modula (red 1 u kodu main1.py u ovom slučaju)
- {built-in method builtins.print} označava vreme provedeno u ugrađenoj metodi štampanja i u ovom primeru bilo je 4 poziva funkciji štampanja
- {method 'disable' of '_lsprof.Profiler' objects} je povezan sa samim *cProfile* modulom.

Zašto prevodimo sa -m?

Opcija -m služi za pokretanje modula kao skripta: python -m module.name. Kada se pokrene python -m cprofile main.py, poziva se modul *cProfile* kao skript i daje main.py kao argument.

6.2 Primer

```
import cProfile      #ugradjeni Python paket, nema potrebe za dodatnim instaliranjem

def my_func():
    i=0
    for i in range(10000000):
        i+=1

cProfile.run('my_func()')
```

Listing 2: main2.py

```
4 function calls in 0.692 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      0.000    0.000    0.692    0.692 <string>:1(<module>)
1      0.692    0.692    0.692    0.692 main2.py:3(my_func)
1      0.000    0.000    0.692    0.692 {built-in method builtins.exec}
1      0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}

PS C:\Users\Warija\Desktop\Fakultet\master\vs python>
```

Slika 2: Profajliranje Python funkcija sa cProfile

Možemo lako da pokrenemo profajler za konkretnu funkciju `my_func()` tako što je prosledimo kao argument ugrađenoj funkciji `cProfile.run()`. Primitimo u izveštaju da se samim pozivom funkcije `my_func()` iz `main2.py` utrošilo 0.692 sekundi zbog velikog opsega brojeva u petlji.

`--name--`

Pre izvršavanja programa, Python interpreter dodeljuje ime Python modula specijalnoj promenljivoj pod nazivom `--name--`. U zavisnosti od toga da li se program izvršava preko komandne linije ili se uvozi modul u drugi modul, dodela za `--name--` će se razlikovati. Ovo možemo primetiti na slikama 3 i 4.

```
print("*****my_module*****")
print(__name__)
```

Listing 3: my_module.py

```
import my_module
print("*****another_module*****")
print(__name__)
```

Listing 4: another_module.py

Kada u terminalu izvršimo `my_module.py` biće ispisano

```
*****my_module*****
__main__
```

Slika 3: Terminal

A kada izvršimo `another_module.py` dobijamo

```
*****my_module*****
my_module
*****another_module*****
__main__
```

Slika 4: Terminal

Sa slike 4 možemo primetiti da je promenljivoj `--name--` prvo dodeljena vrednost `my_module` (prvo je prikazan rezultat izvršavanja `my_module.py`), a zatim je `--name--` jednako `__main__`.

6.3 Primer

```
import cProfile
import pstats
import time
## Nekoliko funkcija cije cemo performanse testirati
def multiply(x,y):
    return x*y

def factorial(n):
    result=1
    for i in range(1,n+1):
        result*=i
    return result

def recursive_factorial(n):
    if n==1:
        return n
    return n*recursive_factorial(n-1)

def sleeping_time():
    time.sleep(5)
    print("Hello")

## Praviceemo listu brojeva na dva nacina i uporediti koji je bolji po performansama

def make_list1():
    result=[]
    for x in range(1000000):
        result.append(x**2)
    return result

def make_list2():
    return [x**2 for x in range(1000000)] #list comprehension [f(x) domain for x]

if __name__=="__main__":
    profile=cProfile.Profile() # kreiramo profajler koji moze
    profile.enable() # poceti sa sakupljanjem podataka o performansama
    print(multiply(1000,500000))
    print(factorial(900))
    print(recursive_factorial(900))
    print(make_list1())
    print(make_list1())
    print(make_list2())
    sleeping_time()
    results=pstats.Stats(profile) #kreiranje pstats.Stats objekta pomocu objekta
    cProfile.profile # poceti sa sakupljanjem podataka o performansama
    results.sort_stats(pstats.SortKey.CUMULATIVE) # moze i
    results.sort_stats('cumulative')
    results.print_stats() # prikaze izvestaj u terminalu
    profile.disable()

    results.dump_stats("results.prof") # cuvanje statistike u fajlu results.prof
```

Listing 5: main1509.py

U ovom primeru testiramo performanse nekoliko funkcija da bismo utvrdili šta oduzima najviše vremena u izvršavanju programa i kako to možemo da popravimo.

Funkcije su:

- `multiply(x,y)` - služi za množenje dva broja;
- `factorial(n)` i `recursive_factorial(n)` - iterativni i rekurzivni faktorijal;
- `sleeping_time()` - funkcija koja troši pet sekundi koristeći ugrađenu funkciju iz biblioteke `time`;
- `make_list1()` i `make_list2()` - funkcije koje prave niz celih brojeva od 0 do 1000000 (biramo veliki broj jer nam je bitno analiziranje vremena izvršavanja za veće vrednosti ili ulaze)

Ispisujemo rezultate ovih funkcija, tj. pozivamo ih dok radi profajler kog smo kreirali i dali mu dozvolu (`enable`) za prikupljanje podataka o performansama.

- `results` je `pstats.Stats` objekat, a `profile` je `cProfile.profile` objekat.
- `sort_stats()` - po kojoj koloni ćemo sortirati izveštaj. Sem `CUMULATIVE`, postoje i `NCALLS`, `FILENAME`, `TIME`...
- `print_stats()` - ugrađena funkcija koja prikazuje izveštaj u terminalu

- `results.dump_stats(ime_fajla)` - čuvanje izveštaja u fajlu. Ovaj fajl će nam poslužiti za bolju vizualizaciju izveštaja.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	5.008	5.008	c:\Users\Marija\Desktop\Fakultet\master\vs py
thon/main1509.py:29(sleeping_time)					
1	5.007	5.007	5.007	5.007	{built-in method time.sleep}
2	1.060	0.530	1.231	0.615	c:\Users\Marija\Desktop\Fakultet\master\vs py
thon/main1509.py:19(make_list1)					
7	0.564	0.081	0.564	0.081	{built-in method builtins.print}
1	0.000	0.000	0.431	0.431	c:\Users\Marija\Desktop\Fakultet\master\vs py
thon/main1509.py:25(make_list2)					
1	0.431	0.431	0.431	0.431	c:\Users\Marija\Desktop\Fakultet\master\vs py
thon/main1509.py:26(<listcomp>)					
2000000	0.171	0.000	0.171	0.000	{method 'append' of 'list' objects}
900/1	0.002	0.000	0.002	0.002	c:\Users\Marija\Desktop\Fakultet\master\vs py
thon/main1509.py:14(recursive_factorial)					
1	0.001	0.001	0.001	0.001	c:\Users\Marija\Desktop\Fakultet\master\vs py
thon/main1509.py:8(factorial)					
1	0.000	0.000	0.000	0.000	C:\Python\Python37\lib\pstats.py:89(_init_)
1	0.000	0.000	0.000	0.000	C:\Python\Python37\lib\pstats.py:99(init)
1	0.000	0.000	0.000	0.000	C:\Python\Python37\lib\pstats.py:118(load_stats)
1	0.000	0.000	0.000	0.000	C:\Python\Python37\lib\cProfile.py:50(create_stats)
1	0.000	0.000	0.000	0.000	{built-in method builtins.isinstance}
1	0.000	0.000	0.000	0.000	{built-in method builtins.hasattr}
1	0.000	0.000	0.000	0.000	{built-in method builtins.len}
1	0.000	0.000	0.000	0.000	c:\Users\Marija\Desktop\Fakultet\master\vs py
thon/main1509.py:5(multiply)					
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Slika 5: python main1509.py

Kada pokrenemo main1509.py, prvo ćemo u terminalu (slika 5) dobiti ispis funkcija koje smo pozvali. Zbog velikih brojeva koje smo prosledili funkcijama, vremena spavanja od 5 sekundi i većeg broja funkcija, izvršavanje ovog programa trajaće 7.234 sekundi. Pošto smo sortirali po koloni **cumtime**, prvi red izveštaja će biti funkcija koja se najduže izvršava a to je **sleeping_time()**.

Sada posmatramo treći i peti red tj. upoređujemo izvršavanja dveju funkcija koje imaju istu upotrebu: **make_list1()** i **make_list2()**. **make_list2()** je brža funkcija jer koristi *list comprehension* koji je implementiran u C-u i optimizovan. Ovo možemo primetiti upoređivanjem pete kolone **percall** iz ta dva reda. Pošto se **make_list1()** izvršava 2 puta (vidimo po **ncalls**) onda je **cumtime=2*percall**.

Vidimo da se metod **append** poziva 2000000 puta jer se **make_list1()** poziva dva puta.

Dalje, **factorial()** se brže izvršava od **recursive_factorial()** i **ncalls** za **recursive_factorial()** pokazuje da postoji jedan primitivni poziv i 900 rekurzivnih. Generalno, iterativne funkcije memorijski i vremenski zahtevaju manje resursa od rekurzivnih.

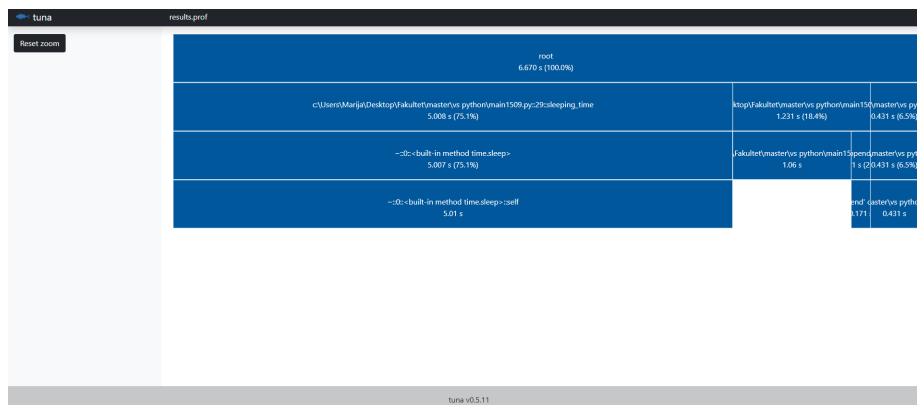
6.3.1 Vizualizacija pomoću alata TUNA

Potrebno je da instaliramo ovaj alat [6]. Za Windows operative sisteme se u terminal unosi

```
pip install tuna.
```

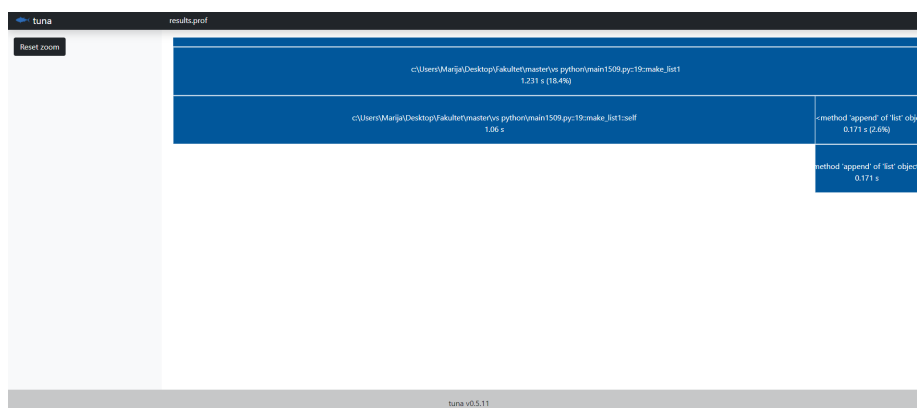
Zatim se pokreće naredba **tuna results.prof** gde smo results.prof dobili putem ugrađene funkcije **dump_stats()**.

Na veb pretraživaču se prikazuju blokovi koji pokazuju količinu vremena potrošenog za izvršavanje svake pojedinačne funkcije i to se lako vidi po veličinama blokova.



Slika 6: Prikazano na veb pretraživaču

Na slici 6 primećujemo da funkcija `sleeping_time()` troši najviše vremena i da bi nju trebalo optimizovati, a u našem slučaju možemo i da je obrišemo. Ako želimo da saznamo narednu funkciju koja je po potrošenom vremenu među prvim, kliknemo plavi pravougaonik koji je s desne strane od kvadrata `sleeping_time()` i to je funkcija `make_list1()` (slika 7).



Slika 7: Prikazano na veb pretraživaču

Možemo dalje da istražujemo koja je sledeća funkcija koju je potrebno optimizovati itd.

Razlika u verzijama

Za python verzije >3.8 ne mora da se piše enable/disable već:

```
...
with cProfile.Profile() as profile:
    print(multiply(1000,500000))
    print(factorial(900))
    print(recursive_factorial(900))
    print(make_list1())
    print(make_list1())
    print(make_list2())
    sleeping_time()
```

Listing 6: main1509.py

6.3.2 Vizualizacija pomoću alata graphviz

Graphviz [4] koristi DOT jezik za opisivanje grafova. Instalira se u Linux-u sa

```
sudo apt-get install python3-graphviz
```

```
sudo apt-get install graphviz.
```

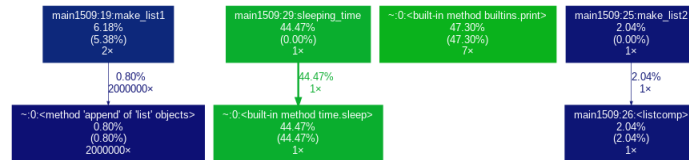
Koristimo gprof2dot za konvertovanje podataka dobijenih profajliranjem u DOT fajl:

```
gprof2dot -f pstats results.prof -o results.dot
```

Ovim je generisan fajl results.dot. Koristimo dot za vizuelnu reprezentaciju DOT fajla:

```
dot -Tpng results.dot -o results.png
```

i ovim dobijamo sliku 8 na kojoj je graf.



Slika 8: Graf za main1509.py

6.4 Primer

Sinhrona i asinhrona funkcije

- Sinhrona funkcije:

U sinhronoj operaciji, program čeka da se zadatak završi pre nego što pređe na sledeći. Izvršenje je sekvencijalno i svaki zadatak mora da se završi pre nego što počne naredni. Međutim, to može dovesti do problema sa performansama, posebno u scenarijima u kojima zadatak oduzima značajno vreme. Za to vreme ceo program je blokiran i ne mogu se izvršiti nikakve druge operacije.

- Asinhrona funkcije:

U asinhronim operacijama, zadatak se može pokrenuti, a program može nastaviti sa drugim zadacima bez čekanja da se završi. Asinhroni kod se obično koristi u scenarijima u kojima zadaci uključuju čekanje na spoljne resurse, kao što su U/I datoteke, mrežni zahtevi ili upiti za bazu podataka. Asinhrona operacije mogu značajno poboljšati performanse tako što dozvoljavaju programu da radi drugi posao dok čeka da se sporiji zadaci završe.

```
import requests
import cProfile
import pstats

urls=["https://sourceforge.net/", "https://www.imdb.com/?ref=nv_home",
      "https://nova.rs/", "https://www.youtube.com/",
      "https://matf-software-verification.github.io/",
      "http://www.matf.bg.ac.rs/m/91/osnovne-matematika/", "https://www.google.com/"]

def taking_htmls_from_web_pages():
    htmls= []
    for url in urls:
        htmls=htmls+[requests.get(url).text]
    print(htmls)

def main():
    pr=cProfile.Profile()
    pr.enable()
    taking_htmls_from_web_pages()
    stats=pstats.Stats(pr)
    stats.sort_stats(pstats.SortKey.TIME)
    stats.print_stats()
    stats.dump_stats(filename='prof_results1.prof')
    pr.disable()
if __name__=='__main__':
    main()
```

Listing 7: main4.py

U ovom kodu ispisujemo pronađene html-ove u okviru datih veb stranica. Hoćemo da napravimo listu html-ova i koristimo biblioteku `requests`. To je sinhrona biblioteka i dok se obrađuje

get zahtev ništa drugo se ne izvršava, tj. čeka se na kraj izvršavanja za svaki url.

U narednom kodu napisana je funkcija koja ima istu svrhu kao `taking_htmls_from_web_pages()` ali je asinhrona i koristićemo `httpx` biblioteku koja je asinhrona i onda uporediti rezultate.

```
import cProfile
import pstats
import httpx
import asyncio

urls=["https://sourceforge.net/", "https://www.imdb.com/?ref=nv_home",
      "https://nova.rs/", "https://www.youtube.com/",
      "https://matf-software-verification.github.io/",
      "http://www.matf.bg.ac.rs/m/91/osnovne-matematika/", "https://www.google.com/"]

async def better_taking_htmls_from_web_pages():
    async with httpx.AsyncClient() as client:
        tasks=(client.get(url) for url in urls)
        reqs=await asyncio.gather(*tasks)
        htmls=[req.text for req in reqs]
        print(htmls)

def main():
    pr=cProfile.Profile()
    pr.enable()
    asyncio.run(better_taking_htmls_from_web_pages())
    stats=pstats.Stats(pr)
    stats.sort_stats(pstats.SortKey.TIME)
    stats.print_stats()
    stats.dump_stats(filename='prof_results2.prof')
    pr.disable()
if __name__=='__main__':
    main()
```

Listing 8: main5.py

`tasks=(client.get(url) for url in urls)` - prave se asinhroni HTTP zahtevi za svaki URL na listi URL adresa. Ovi zahtevi još nisu izvršeni, samo su pripremljeni

`reqs=await asyncio.gather(*tasks)` - koristi se `asyncio.gather` za istovremeno izvršavanje svih asinhronih HTTP zahteva. Ključna reč `await` se koristi za asinhrono čekanje da se svi zadaci završe. Rezultat je lista odgovora (`reqs`), gde svaki odgovor odgovara URL adresi.

`htmls=[req.text for req in reqs]` - u ovoj liniji se izdvaja HTML sadržaj (atribut `text`) iz svakog `req` i kreira se lista `html`-ova

6.4.1 Vizualizacija alatom SnakeViz

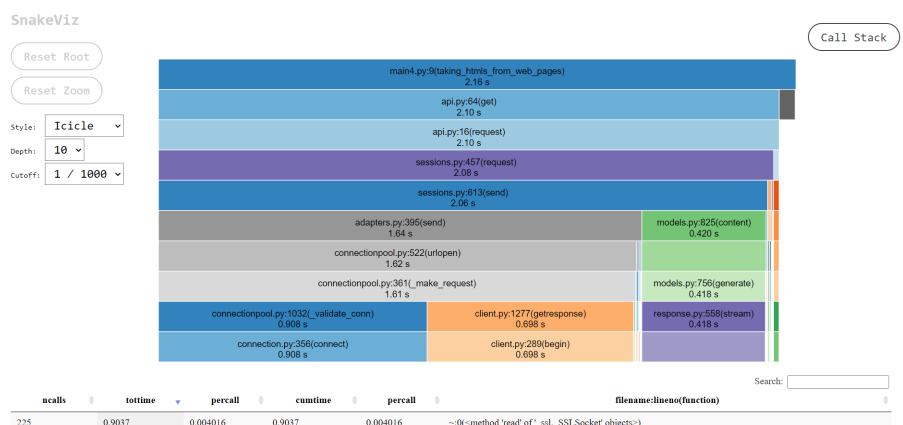
Potrebno je da instaliramo ovaj alat [5]. Za Windows operative sisteme se u terminalu kuca

```
pip install snakeviz.
```

Zatim se pokreće naredba `snakeviz prof_results1.prof` gde smo `prof_results1.prof` dobili putem ugrađene funkcije `dump_stats()`. Na veb pretraživaču se prikazuju blokovi koji pokazuju količinu vremena potrošenog za izvršavanje svake pojedinačne funkcije i to se lako vidi po veličinama blokova.

Ispod njih nalazi se izveštaj koji smo dobili i u našem terminalu.

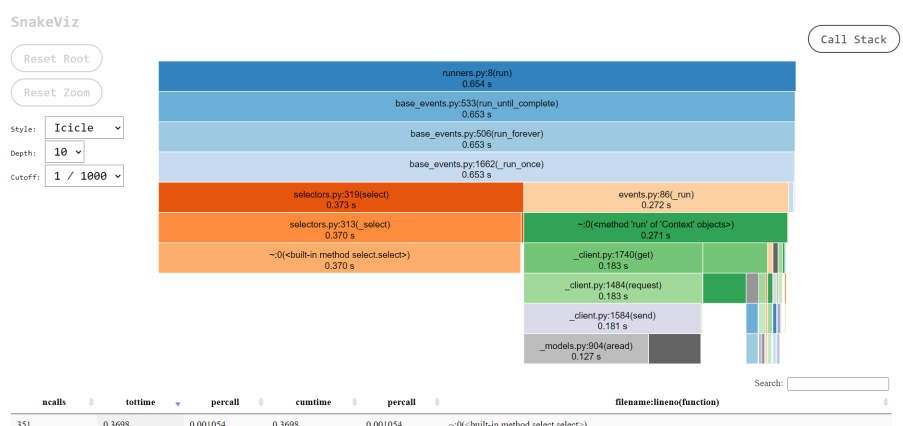
Umesto blokova možemo koristiti i kružni grafikon tako što izaberemo `Style:Sunburst`.



Slika 9: Prikazano na veb pretraživaču

Na slici 9 možemo primetiti da get zahtev oduzima više vremena u odnosu na ostale funkcije u programu.

Kada se pokrene `snakeviz prof_results2.prof` videćemo da je funkcija `better_taking_htmls_from_web_pages()` zaista bolja što se tiče vremenskih performansi jer je asinhrona.



Slika 10: Prikazano na veb pretraživaču

Na slikama 9 i 10 može se primetiti da je `main5.py` više nego duplo brži u odnosu na `main4.py`, tj. funkciji `run` je potrebno 0.654s, a funkciji `get` 2.10s.

Drugi alati za vizualizaciju

- *Py-Spy* pravi flame graphs
- *RunSnakeRun*
- *kcachegrind*

7 Poređenje sa ostalim Python profajlerima

- Razlika između *profile* i *cProfile*:

Modul *profile* je implementiran u čistom Python-u i deo je standardne biblioteke. Pruža osnovni i jednostavan alat za profajliranje. Može u većoj meri uticati na performanse koda koji se profajlira. Pogodan je za osnovne potrebe profajliranja i prenosiviji je pošto je implementiran u čistom Python-u. Modul *cProfile* je, s druge strane, implementiran u C-u. Generalno je brži od modula *profile* i pogodan je za ozbiljnije zadatke profajliranja.

- Razlika između *Py-Spy* i *cProfile*:
Py-Spy je profajler uzorkovanja koji periodično uzorkuje stanje odnosno snima ‘slike’ steka u intervalima, a *cProfile* je zasnovan na instrumentalizaciji. *Py-Spy* je spoljni alat koji se može koristiti iz komandne linije. Povezuje se sa pokrenutim Python procesom i uzorkuje njegovo stanje. Daje pregled gde program provodi svoje vreme, ali ne pruža detaljne informacije o pojedinačnim pozivima funkcija. *cProfile* dodaje neke dodatne troškove izvršavanju programa zbog detaljnog profajliranja koje obavlja, dok *Py-Spy* ima manje troškove jer uzorkuje program u intervalima.
- Razlika između *PyFlame* i *cProfile*:
PyFlame ne koristi instrumentalizaciju već uzorkovanje. To je spoljni alat koji se mora instalirati i koristiti iz komandne linije. Ovaj profajler generise *flame graph* koji omogućavaju lakšu vizualizaciju podataka usled profajliranja. *cProfile* omogućava detaljnije profajliranje od *PyFlame*.

8 Zaključak

Profajleri igraju ključnu ulogu u razumevanju performansi Python programa tako što pružaju uvid u vreme izvršavanja funkcija i identifikuju uska grla. Među različitim dostupnim profajlerima, *cProfile* se ističe kao ugrađena i efikasna opcija za profajliranje Python koda, pogodna za male i velike projekte. [3]

Kada je u pitanju vizualizacija podataka *cProfile*, nekoliko alata nudi pogodne načine za analizu i tumačenje rezultata profajliranja. SnakeViz, Py-Spy, RunSnakeRun i Pycallgraph su značajni kandidati.

Izbor pravog alata za vizualizaciju zavisi od individualnih preferencija, specifičnih zahteva analize i željenog nivoa detalja. Programeri se mogu odlučiti za vizualizacije zasnovane na vebu, interaktivne grafičke interfejse ili alate komandne linije na osnovu njihovog toka posla i poznavanja različitih okruženja.

Meni lično, najviše je odgovarao SnakeViz koji je lak i intuitivan za korišćenje.

Iako *cProfile* pruža osnovne mogućnosti profajliranja, vredi napomenuti da je Python ekosistem evoluirao sa dodatnim profajlerima nezavisnih proizvođača i alatima za vizualizaciju koji nude naprednije funkcije i interaktivne interfejse. Bez obzira na to, *cProfile* ostaje fundamentalni i lako dostupni alat za profajliranje Python koda.

Literatura

- [1] Python libraries. <https://docs.python.org/3/library/profile.html>.
- [2] python profiling. <https://pyshark.com/profiling-python-code-with-cprofile>.
- [3] python-profiling. <https://realpython.com/python-profiling/>.
- [4] Visualisation tool graphviz. <https://graphviz.org/>.
- [5] Visualisation tool snakeviz. <https://jiffyclub.github.io/snakeviz/>.
- [6] Visualisation tool tuna. <https://github.com/nschloe/tuna>.
- [7] Vladimir Filipović. *prezentacije sa predmeta Uvod u veb i internet tehnologije*. Matematički fakultet, Beograd, 2019.
- [8] Milena Vujošević Jančić. *Verifikacija softvera*. Matematički fakultet, Beograd, 2023.