Optimizacija Python koda

Blažević, Lucija

Undergraduate thesis / Završni rad

2017

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:939901

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-28**

Repository / Repozitorij:

Repository of School of Applied Mathematics and Computer Science

Sveučilište J.J.Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Lucija Blažević

Optimizacija Python koda

Završni rad

Osijek, 2017.

Sveučilište J.J.Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Lucija Blažević

Optimizacija Python koda

Završni rad

Voditelj rada:
izv.prof.dr.sc. Domagoj Matijević

Osijek, 2017.

SAŽETAK

Optimizacija koda (code tuning) je vrlo korisna stvar kada se bavimo ozbiljnijim programiranjem. Zahvaljujući njoj naš kod postaje brži i efikasniji. Upravo time ćemo se u ovome radu baviti. Naučit ćemo kako optimizirati pojedine dijelove programskog koda napisanog u programskom jeziku Python. Kroz primjere ćemo vidjeti koje dijelove je potrebno optimizirati, a koji su, već sami po sebi, dovoljno brzi i nemaju potrebe za optimizacijom. Proučit ćemo optimizaciju regularnih izraza, stringova, lista te rječnika. Na kraju samog rada, donijet ćemo konkretne zaključke i smjernice koje će vam reći čega se trebate pridržavati te što trebate izbjegavati kada je riječ o optimizaciji. Nadam se kako će vam one biti korisne u daljnjem radu ukoliko se budete susretali s optimizacijom.

Ključne riječi: optimizacija, Python kod, regularni izrazi, stringovi, liste, rječnici

ABSTRACT

Code tuning is a very usefull thing when dealing with some serious programming. Thanks to code tuning, our code becomes faster and more efficient. That is exactly what we're gonna deal with in this paper. We will learn how to optimize individual parts of program code written in Python Programming Language. Through examples we will see which parts need to be optimized, and which themselves are already fast enough and do not have any need for optimization. Also, we will study the optimization of regular expressions, strings, lists and dictionaries. At the end of this paperwork, we'll get concrete conclusions and guidelines that will tell you what you should comply with and what you should avoid when it comes to optimization. I hope that they will be useful in further work if you encounter optimization.

Key words: optimization, Python code, regular expressions, strings, lists, dictionaries

Sadržaj

Sažetak **3**

Uvod **6**

1 Optimizacija koda **7**

2 Korištenje `timeit` modula **7**
2.1 Što je to timeit modul? 8
2.2 Detaljnije o timeit modulu uz primjer 8

3 Regularni izrazi **9**
3.1 Optimizacija regularnih izraza 10

4 Stringovi **13**
4.1 Optimizacija stringova 13

5 Liste **15**
5.1 Optimizacija lista 15

6 Rječnici **17**
6.1 Lambda funkcija 18
6.2 Optimizacija rječnika 18

Zaključak **23**

Literatura **24**

Popis slika

1 Funkcija uvecaj() 8
2 timeit modul 8
3 Primjer korištenja repeat() metode 9
4 Primjer korištenja min() metode 9
5 Primjer korištenja re.search 10
6 Primjer korištenja re.compile 10
7 Korištenje regularnih izraza 10
8 Skraćena sintaksa regularnih izraza 11
9 Kompajlirana verzija regularnih izraza 11
10 Korištenje petlje 12
11 Korištenje metode isalpha 12
12 Korištenje stringova 13
13 Korištenje string.replace 14

14 Zamjena while petlje 14
15 Korištenje for petlje 16
16 Optimizacija liste 16
17 range() i join() 17
18 Lambda funkcija 18
19 Korištenje map i lambda funkcije 18
20 Korištenje rječnika 19
21 Korištenje lambda funkcije 20
22 Korištenje list comprehention 21
23 Korištenje funkcije string.maketrans 22

Uvod

U ovom završnom radu, govorit ćemo o optimizaciji programskog koda. Kako smo se u kolegiju Programiranje i programsko inženjerstvo bavili programiranjem u Pythonu, ovdje ćemo se bazirati isključivo na optimizaciji programskih kodova napisanih u Pythonu.

Rad je podijeljen u šest poglavlja. Na početku samog rada, u prvom poglavlju, naučit ćemo nešto osnovno o optimizaciji koda; što je to optimizacija koda te kada je potrebno optimizirati naš kod i zašto. U drugom poglavlju reći ćemo nešto o vremenskim funkcijama koje nam pomažu računati vrijeme potrebno za izvršenje našeg programskog koda. Zatim ćemo u sljedećim poglavljima proučiti optimizaciju određenih struktura podataka na način da ćemo napraviti neki kratki uvod u određenu strukturu pa ćemo kroz praktičan primjer vidjeti njenu optimizaciju. Na taj način probat ćemo objasniti kako optimizirati regularne izraze, stringove, liste te rječnike. Na tim primjerima također ćemo dobiti još jedan lijepi uvid u to što optimizacija koda jest te ćemo također naučiti kako i na koji način je najbolje optimizirati određene dijelove našeg koda.

Nakon što proučimo kako optimizirati određene strukture, u zaključnom dijelu ukratko ćemo ponoviti što smo sve naučili te dati savjete na što se trebamo bazirati kada se susretnemo s optimizacijom.

1 Optimizacija koda

Kako smo već rekli u uvodnom dijelu, u ovom radu bavit ćemo se optimizacijom Python koda, pa je prirodno da na početku kažemo nešto općenito o optimizaciji koda. Što optimizacija koda zapravo jest? Zašto je potrebno optimizirati naš kod? Da li je uopće potrebno optimizirati kod ili to možemo izbjeći? Na ova pitanja pokušat ćemo odmah na početku dati odgovor.

Optimizacija koda *(code tuning)* je izmjena ili uređivaje programskog koda kako bi on bio brži i efikasniji. Temelji se na izmjeni određenih dijelova koda (npr. mijenjanje određenih petlji) te izbacivanju nepotrebnih dijelova (npr. izbacivanje nekoliko linija koda koje ne pridonose važnosti koda). Nadalje, postavlja nam se pitanje zbog čega trebamo optimizarati naš kod? Može se reći kako je optimiziranje programskog koda samo po sebi dosta primamljivo jer sama činjenica da smo uspješno ubrzali naš kod nam govori o tome kako smo postali ozbiljniji programeri. No, prije nego započnemo s optimizacijom, moramo se zapitati da li je ona uopće potrebna. Naravno da je pozitivna stvar da je naš kod brži i efikasniji, ali optimiziranje iziskuje dosta vremena pa bi trebali dobro razmisliti je li naš kod uistinu tako loš. Naime, može nam se dogoditi da dosta vremena posvetimo optimizaciji da bi na kraju uvidjeli kako smo samo malo ubrzali naš kod. Vrijeme potrošeno na to ne možemo vratiti, a mogli smo ga iskoristiti za neke druge, korisnije stvari. Također, pažnju moramo obratiti i na to jesmo li sigurni da je naš kod gotov i da ispravno radi, tj. radi li naš kod ono što od njega želimo i očekujemo. Optimizacija nam neće biti ni od kakve koristi ako sam naš izvorni kod nije u potpunosti gotov i ispravno napisan.
Dakle, dvije najbitinije stvari o kojima moramo razmisliti, prije nego započnemo s optimizacijom programskog koda, su:

- Jesmo li sigurni da smo gotovi s kodiranjem i da naš kod radi ono što želimo?
- Ima li uistinu potrebe za njegovom optimizacijom?

Ukoliko na oba pitanja odgovorimo potvrdno, spremni smo za optimizaciju.

2 Korištenje `timeit` modula

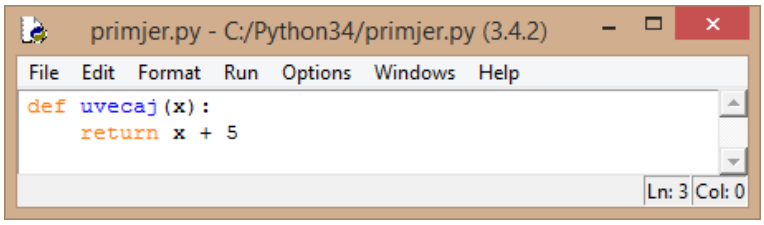
Kada se bavimo optimizacijom koda, potrebno je da naš kod sadrži neku funkciju, takozvanu vremensku funkciju, koja će nam računati vrijeme potrebno za izvršenje tog našeg koda. Ono što je bitno kod optimizacije koda u Pythonu jest to da nikada ne trebamo pisati sami svoju vremensku funkciju. Razlog tomu je vrlo jednostavan. Ne samo da Python u sebi već sadrži vremensku funkciju, već je i pisanje vlastitih previše komplicirano. Zbog tih razloga vlastite vremenske funkcije nećemo pisati, već ćemo koristiti Pythonov `timeit` modul.

2.1 Što je to timeit modul?

Timeit modul pruža nam jednostavno sučelje za određivanje vremena izvršenja dijelova Python koda. On koristi vremensku funkciju kako bi nam osigurao najtočniji izračun vremena potrebnog za izvršenje koda. Pri samom njegovom korištenju, bitno je da znamo da timeit modul funkcionira samo ukoliko znamo koji točno dio koda trebamo i želimo optimizirati.

2.2 Detaljnije o timeit modulu uz primjer

Pretpostavimo da imamo sljedeću, vrlo jednostavnu funkciju, koja prima neki broj i vraća taj broj uvećan za 5.



Slika 1: Funkcija uvecaj()

Tu funkciju nazvali smo uvecaj, dok smo cijeli taj program spremili pod nazivom primjer.py. Pogledajmo sada kako izgleda naš timeit modul:

```
>>> import timeit
>>> t=timeit.Timer("primjer.uvecaj(3)", "import primjer")
>>> t.timeit()
0.5545398519011471
>>>
```

Slika 2: timeit modul

- Timer prima dva argumenta; prvi argument je izjava za koju želimo izračunati vrijeme, dok drugi argument postavlja okruženje za tu izjavu, tj. moramo navesti pod kojim smo nazivom spremili cijeli taj naš program kako bi računalo moglo naći funkciju koju koristimo.
- timeit() pokreće našu izjavu i vraća nam broj sekundi koliko je bilo potrebno da se ona izvrši. Argument koji timeit() prima nam govori koliko puta treba pokrenuti tu izjavu. Ako argument nije zadan (kao u našem primjeru), on po defaultu iznosi 1000000.

Jedna od bitnih metoda koju ćemo koristiti je metoda repeat(). repeat() metoda prima dva argumenta. Prvi argument je broj koji nam govori koliko puta trebamo pozvati timeit(), tj. koliko puta trebamo ponoviti cijeli test, dok nam drugi argument predstavlja broj ponavljanja pojedinog testa. Ova metoda vraća nam listu brojeva koji nam govore koliko je trajao pojedini krug ponavljanja. Ako nisu posebno zadani, argumenti ove metode iznose 3 i 1000000.

```
>>> t.repeat(4, 2000000)
[1.0605198869691748, 1.0943053089419443, 1.0888541568697363, 1.119134186531106]
```

Slika 3: Primjer korištenja repeat() metode

Uočimo kako kod korištenja repeat() metode vremena nisu identična. Ona gotovo nikada neće niti biti ista, ne zbog razlike u kodu, već zbog pozadinskih procesa ili nekih drugih faktora koji se odvijaju izvan Pythona. No oko tih malih razlika ne trebamo se previše brinuti. Kako bismo izbjegli listu koju nam vraća repeat() metoda, u daljnjem radu koristit ćemo metodu min() koja uzima listu tih vremena i vraća nam najmanje vrijeme potrebno za izvršenje našeg koda:

```
>>> min(t.repeat(2, 2000000))
1.1568034920628634
```

Slika 4: Primjer korištenja min() metode

Ove stvari biti će nam jako korisne u daljnjem radu jer ćemo upravo preko njih provjeravati jesmo li uspjeli ubrzati naš kod. Sada, kada smo naučili kako koristiti vremensku funkciju, možemo krenuti s optimizacijom pojedinih struktura podataka.

3 Regularni izrazi

Prije same optimizacije regularnih izraza, reći ćemo nešto o regularnim izrazima. Što su to regularni izrazi?
Regularni izrazi su nizovi znakova koji, u skladu s određenim pravilima, opisuju druge nizove znakova. Prvenstveno nam koriste za pretraživanje, raščlanjivanje i zamjenu dijelova teksta preko re modula. Navest ćemo samo nekoliko metoda koje ćemo koristiti u našem kodu, no njih ima mnogo više.

re.search (pattern, string, flags=0) – prima regularni izraz (pattern) i string u kojem se pokušava pronaći pattern definiran regularnim izrazom. Ukoliko je pronađen, search vraća odgovarajuću instancu, u suprotnom vraća None.

```
>>> import re
>>> re.search('^[A-Za-z]+$', 'sun')
<_sre.SRE_Match object; span=(0, 3), match='sun'>
>>> re.search('^[A-Za-z]+$', '123')
>>>
```

Slika 5: Primjer korištenja re.search

Znak '^' označava početak stringa, a '$' kraj stringa. U uglatim zagradama navodimo skup znakova, dok nam znak '+' omogućava ponavljanje prethodnog regularnog izraza, tj. u ovom primjeru, [A-Za-z]+ će odgovarati bilo kojem nizu znakova iz skupa [A-Za-z] (ne mora biti samo jedan znak iz tog skupa).

re.compile (pattern, flags=0) – pretvara pattern u objekt koji se zatim može, preko metode search(), koristiti za sparivanje s nekim stringom.

```
>>> import re
>>> sva_slova = re.compile('[A-Za-z]+$').search
>>> sva_slova('abc')
<_sre.SRE_Match object; span=(0, 3), match='abc'>
```

Slika 6: Primjer korištenja re.compile

Proučimo sada optimizaciju regularnih izraza kroz sljedeći primjer.

3.1 Optimizacija regularnih izraza

Pretpostavimo da imamo funkciju koja radi sljedeće: prima jednu riječ i provjerava je li ta riječ neprazan string sastavljen isključivo od slova. Ukoliko je, funkcija vraća tu istu riječ, u suprotnom vraća "000". Za početak, mi smo tu provjeru napravili pomoću regularnih izraza, no, kada krenemo optimizirati taj naš kod, vidjet ćemo da bi regularne izraze trebali izbjegavati. Prije svega, pogledajmo naš početni kod:

```
import string, re

def reg_izrazi(rijec):
    sva_slova = string.ascii_uppercase + string.ascii_lowercase
    if not re.search('^[%s]+$' % sva_slova, rijec):
        return "000"
    return rijec

if __name__ == '__main__':
    from timeit import Timer
    word = 'proljece'
    statement = "reg_izrazi('%s')" % word
    t = Timer(statement, "from __main__ import reg_izrazi")
    print(word.ljust(15), reg_izrazi(word), min(t.repeat()))
```

```
proljece        proljece 9.538276673012321
```

Slika 7: Korištenje regularnih izraza

Vidimo kako u main sekciji, postavljamo vremenski test za riječ 'proljece', testiramo tu riječ tri puta i vraćamo minimalno vrijeme, zatim ispisujemo riječ koju provjeravamo, rezultat funkcije te vrijeme potrebno za izvršenje funkcije. Možemo vidjeti kako je u ovom slučaju našem programu trebalo 9.5 sekundi da provede ovaj test pomoću regularnih izraza. No, možemo li bolje?

Za početak, mogli bi koristiti skraćenu sintaksu regularnih izraza te, umjesto da posebno definiramo sva slova preko stringova, jednostavno to napravimo pomoću regularnih izraza.

Također, kako smo vidjeli u primjeru korištenja `re`.compile, regularne izraze možemo i kompajlirati (prevesti u jezik poznat našem računalu) i koristiti kompajliranu verziju što nam daje već puno bolji rezultat. Skraćenu sintaksu regularnih izraza i kompajliranu verziju te njihove rezultate možemo vidjeti na sljedećim primjerima:

```
import re

def reg_izrazi(rijec):
    if not re.search('^[A_Za-z]+$', rijec):
        return "000"
    return rijec

if __name__ == '__main__':
    from timeit import Timer
    word = 'proljece'
    statement = "reg_izrazi('%s')" % word
    t = Timer(statement, "from __main__ import reg_izrazi")
    print(word.ljust(15), reg_izrazi(word), min(t.repeat()))
```

```
proljece        proljece 5.456603008482507
```

Slika 8: Skraćena sintaksa regularnih izraza

```
import re

samo_slova = re.compile('^[A-Za-z]+$').search
def reg_izrazi(rijec):
    if not samo_slova(rijec):
        return "000"
    return rijec

if __name__ == '__main__':
    from timeit import Timer
    word = 'proljece'
    statement = "reg_izrazi('%s')" % word
    t = Timer(statement, "from __main__ import reg_izrazi")
    print(word.ljust(15), reg_izrazi(word), min(t.repeat()))
```

```
proljece        proljece 1.6543907889090326
```

Slika 9: Kompajlirana verzija regularnih izraza

Primjetimo kako regularne izraze lako možemo zamijeniti s petljom. No, hoće li nam to biti brže rješenje? Odgovor na to pitanje lako možemo dobiti ako testiramo naš program. Pa probajmo umjesto regularnog izraza ubaciti petlju i pogledajmo što ćemo dobiti:

```
def reg_izrazi(rijec):
    if not rijec:
        return "000"
    for c in rijec:
        if not ('A'<=c<='Z') and not ('a'<=c<='z'):
            return "000"
    return rijec

if __name__ == '__main__':
    from timeit import Timer
    word = 'proljece'
    statement = "reg_izrazi('%s')" % word
    t = Timer(statement, "from __main__ import reg_izrazi")
    print(word.ljust(15), reg_izrazi(word), min(t.repeat()))
```

```
proljece        proljece 5.429213510759569
```

Slika 10: Korištenje petlje

Uočimo kako petlja možda jest malo brža od regularnih izraza, ali nije brža od kompajlirane verzije regularnih izraza. Razlog tomu je to što se kompajlirana verzija prirodno pokreće na računalu, dok su petlje napisane u Pythonu i pokreću se kroz Python. Za sada nam se kao najbolje rješenje čini kompajlirana verzija regularnih izraza.

No, Python nam nudi još jednu stvar. Nudi nam string metodu `isalpha()` koja provjerava sadrži li string samo slova. Čini nam se kako bi nam korištenje te metode dosta olakšalo stvar jer upravo to naš program i treba napraviti, provjeriti je li string sastavljen isključivo od slova. Na sljedećoj slici vidjet ćemo kako je korištenje ove metode, ne samo jednostavnije, već i dosta brže od svih prethodnih!

```
def reg_izrazi(rijec):
    if (not rijec) and (not rijec.isalpha()):
        return "000"
    return rijec

if __name__ == '__main__':
    from timeit import Timer
    word = 'proljece'
    statement = "reg_izrazi('%s')" % word
    t = Timer(statement, "from __main__ import reg_izrazi")
    print(word.ljust(15), reg_izrazi(word), min(t.repeat()))
```

```
proljece        proljece 0.35162655419673294
```

Slika 11: Korištenje metode `isalpha`

Sve u svemu, zaključak ovoga poglavlja jest da, ukoliko znamo neku string metodu kojom bi mogli zamijeniti regularni izraz u našem kodu, iskoristimo ju, jer ona će nam donijeti brže rješenje, a i naš kod će biti jednostavniji i razumljiviji. No, ukoliko biramo između regularnih izraza i petlji, bolje nam je koristiti regularne izraze.

4 Stringovi

U ovome uvodu nećemo ništa posebno govoriti o stringovima. Podrazumjevamo kako su nam stvari koje koristimo u ovom poglavlju, vezane za stringove, već opće poznate, pa bi najbolje bilo da odmah krenemo s njihovom optimizacijom.

4.1 Optimizacija stringova

Kod optimizacije stringova uzet ćemo za primjer funkciju koja prima neki string sastavljen od brojeva (pretpostavimo da je string ili prazan ili je sastavljen isključivo od brojeva), uklanja sve jedinice iz tog stringa (ako ih ima), nadopunjava string s nulama (ukoliko je potrebno) te vraća prvih 5 znamenaka našeg stringa. Mi smo to napravili tako da smo pomoću `re.sub` izbacili sve jedinice iz početnog stringa i spremili ga u neki novi string. Zatim smo taj novi string nadopunili s nulama (ukoliko on ima manje od 5 znamenaka) te vratili prvih 5 znamenki:

```
import re

def stringovi(kod):
    novi_kod = re.sub('1', '', kod)
    while len(novi_kod) < 5:
        novi_kod += "0"
    return novi_kod[:5]

if __name__ == '__main__':
    from timeit import Timer
    codes = ("314513152", "13121")
    for code in codes:
        statement = "stringovi('%s')" % code
        t = Timer(statement, "from __main__ import stringovi")
        print(code.ljust(15), stringovi(code), min(t.repeat()))
```

```
314513152       34535 9.66221352399281
13121           32000 10.625240797699686
```

Slika 12: Korištenje stringova

Krenimo sada s optimizacijom tog koda. Za početak, sjetimo se kako smo u prethodnom poglavlju optimizirali regularne izraze. Vidjeli smo na primjerima da nam regularni izrazi nisu baš najbolje rješenje te da bi ih trebali zamijeniti s nečim boljim.

Također smo zaključili kako nije pametno regularne izraze zamijeniti s petljom, već s nekom jednostavnijom string metodom. Uočimo kako Python sadrži posebnu string metodu koja zapravo radi isto što i re.sub. Ta metoda naziva se string.replace a funkcionira na način da prima dva argumenta te u našem stringu zamjenjuje prvi argument s drugim. Probajmo sada navedeni regularni izraz zamijeniti s tom string metodom da vidimo hoće li nam se vrijeme poboljšati.

```
def stringovi(kod):
    novi_kod = kod.replace('1', '')
    while len(novi_kod) <5:
        novi_kod += "0"
    return novi_kod[:5]

if __name__ == '__main__':
    from timeit import Timer
    codes = ("314513152", "13121")
    for code in codes:
        statement = "stringovi('%s')" % code
        t = Timer(statement, "from __main__ import stringovi")
        print(code.ljust(15), stringovi(code), min(t.repeat()))
```

```
314513152       34535 2.629250469307391
13121           32000 3.854439090233681
```

Slika 13: Korištenje string.replace

Vidimo da nam uistinu ova izmjena daje puno bolje rezultate. Iz tog razloga, ostavit ćemo taj dio koda sa string metodom i prebaciti se na sljedeći dio u kojem prekratke rezultate nadopunjavamo nulama, a predugačke skraćujemo na 5 znamenaka. Primjetimo kako naš novi_kod odmah možemo nadopuniti s 5 nula. U tom slučaju znamo da novi_kod sadrži minimalno 5 znamenaka pa čak i u slučaju da je naš početni string prazan. Sada, na jednostavan način, možemo vratiti prvih 5 znamenaka našeg novog stringa:

```
def stringovi(kod):
    novi_kod = kod.replace('1', '')
    novi_kod += "00000"
    return novi_kod[:5]

if __name__ == '__main__':
    from timeit import Timer
    codes = ("314513152", "13121")
    for code in codes:
        statement = "stringovi('%s')" % code
        t = Timer(statement, "from __main__ import stringovi")
        print(code.ljust(15), stringovi(code), min(t.repeat()))
```

```
314513152       34535 2.4778508651703026
13121           32000 2.1741754273863787
```

Slika 14: Zamjena while petlje

Taj postupak nam daje puno brže i povoljnije rezultate. A možemo i primjetiti kako je naš kod uredniji i lakši za čitati i razumjeti.

Dakle, kada je riječ o stringovima, uvijek bismo trebali tražiti neke posebne string metode koje možemo koristiti u našem kodu. Također, probajmo izbjegavati petlje i na što jednostavniji način pristupiti našem kodu. Ponekad je najočitiji i najjednostavniji način ujedno i najbrži.

5 Liste

Korištenje lista u Pythonu je vrlo jednostavna stvar pa nećemo o njima previše govoriti, samo ćemo navesti par osnovnih stvari da se prisjetimo što su to liste i kako ih koristimo.

Lista je skup elemenata definiran unutar uglatih zagrada. Indeksi liste su od 0 do n-1 pa ukoliko na primjer imamo listu: lista = ['a', 'b', 'c', 'd'], pri pozivu lista[0], dobit ćemo prvi element liste, tj. u ovom primjeru 'a'. Kako smo već rekli, indeksi liste nam idu do n-1, pa ćemo zadnjem elementu liste pristupiti tako da pozovemo lista[3], što će nam dati zadnji element 'd'. Također, zadnjem elementu možemo pristupiti i na način da pozovemo lista[-1] što nam isto za rezultat daje element 'd'.

Pri korištenju lista, često uzimamo određene intervale lista koji su nam potrebni.
Uzmimo ponovno za primjer listu: lista = ['a', 'b', 'c', 'd']

- lista[1:3] nam zapravo znači da uzimamo elemente od indeksa 1 do indeksa 3 (ali bez indeksa 3), pa ćemo u ovom primjeru dobiti elemente: 'b' i 'c'
- lista[:3] = lista[0:3] → 'a', 'b', 'c'
- lista[1:] = lista[1:len(lista)] → 'b', 'c', 'd'

Sada, nakon ovog kratkog uvoda, možemo krenuti s optimizacijom lista.

5.1 Optimizacija lista

Uzmimo primjer funkcije koja prima neki string (pretpostavimo da je on sadržan isključivo od znamenaka) te iz tog stringa uklanja uzastopne duplikate. Mi smo to učinili na vrlo jednostavan način, koristeći for petlju. Nekom novom stringu pridružili smo prvi element početnog stringa, dok smo u petlji za svaki sljedeći element provjeravali je li on isti kao prethodni. Ukoliko su različiti, dodali smo ga novom stringu, u suprotnom nastavljali smo dalje provjeravati za sljedeći element.

```python
def liste(kod):
    novi_kod = kod[0]
    for br in kod[1:]:
        if novi_kod[-1] != br:
            novi_kod += br
    return novi_kod

if __name__ == '__main__':
    from timeit import Timer
    code = "3445733223"
    statement = "liste('%s')" % code
    t = Timer(statement, "from __main__ import liste")
    print(code.ljust(15), liste(code), min(t.repeat()))
```

```
3445733223      3457323 4.583826901663795
```

Slika 15: Korištenje `for` petlje

Možemo pomisliti kako nam program usporava to, što svaki put kada prolazimo kroz petlju, provjeravamo novi_kod[-1] pa probajmo taj dio izmjeniti tako da zadnju znamenku spremimo u zasebnu varijablu pa provjeravamo nju:

```python
def liste(kod):
    novi_kod = ''
    zadnji_br = ''
    for br in kod:
        if br != zadnji_br:
            novi_kod += br
            zadnji_br = br
    return novi_kod

if __name__ == '__main__':
    from timeit import Timer
    code = "3445733223"
    statement = "liste('%s')" % code
    t = Timer(statement, "from __main__ import liste")
    print(code.ljust(15), liste(code), min(t.repeat()))
```

```
3445733223      3457323 4.115162267874798
```

Slika 16: Optimizacija liste

Kako nam je vrijeme podjednako i nije se previše poboljšalo, možemo zaključiti da stalno pristupanje novi_kod[-1] uopće zapravo ne predstavlja problem.
Stoga, probajmo pristupiti optimizaciji ovog dijela koda na drugačiji način. Znamo da string možemo tretirati kao listu znakova (kako smo dosada i činili) pa pomoću funkcije range() pretražimo tu listu i izvucimo svaki znak koji je drugačiji od prethodnog. Time ćemo dobiti novu listu znakova (međusobno različitih) koje metodom join() spojimo u string.
Pa pogledajmo na sljedećoj stranici kako to izgleda.

```
def liste(kod):
    novi_kod = "".join([kod[i] for i in range(len(kod)) if i==0 or kod[i-1]!=kod[i]])
    return novi_kod

if __name__ == '__main__':
    from timeit import Timer
    code = "3445733223"
    statement = "liste('%s')" % code
    t = Timer(statement, "from __main__ import liste")
    print(code.ljust(15), liste(code), min(t.repeat()))
```

```
3445733223      3457323 11.908574855710427
```

Slika 17: `range()` i `join()`

Vidimo kako nam se vrijeme i dalje nije poboljšalo. Naš kod je sada čak i dosta sporiji nego onaj kojeg smo imali na početku.

Dakle, promatrajući sve promjene koje smo radili, možemo vidjeti da nam ništa pretjerano ne pomaže u optimizaciji koda pa je pametnije da taj kod jednostavno ostavimo ovako kako je jer nam je to još i najbolje rješenje.

6 Rječnici

Na kolegiju Programiranje i programsko inženjerstvo već smo naučili što su rječnici u Pythonu i čemu nam sve oni služe pa ćemo se samo ukratko toga prisjetiti kako bismo lakše shvatili naš primjer optimizacije rječnika.

Rječnici su tipovi podataka oblika ključ:vrijednost(i). Rječnik uvijek ima vitičaste zagrade i po tome ga razlikujemo od ostalih tipova podataka, npr. lista. Elemente rječnika odvajamo zarezom, a svaki element se sastoji od ključa i vrijednosti koje odvajamo dvotočkom. npr.:

```
rjecnik = {"A":"1", "B":"2"}
```

nam zapravo znači da je ključ prvog elementa "A" s vrijednošću "1", dok je ključ drugog elementa "B" s vrijednošću "2".
Elementima rječnika pristupa se isključivo preko ključa te ako bismo npr. željeli pristupiti 1. elementu ovog rječnika, učinili bismo to na način: rjecnik["A"], što bi nam kao rezultat dalo njegovu vrijednost '1'.

6.1 Lambda funkcija

Prije nego započnemo s optimizacijom, reći ćemo nešto i o `lambda` funkciji koju ćemo kasnije nakratko koristiti.

Lambda funkcija u Pythonu zapravo radi isto što i obična funkcija, samo što ona, za razliku od funkcija, nema svoje ime i malo je drugačije konstruirana.
Promotrimo sljedeći primjer koji će nam dati bolji uvid u lambda funkcije:

```
>>> def a(x): return x + 5

>>> print(a(4))
9
>>> b = lambda x: x + 5
>>> print(b(4))
9
```

Slika 18: Lambda funkcija

Vidimo da a() i b() rade istu stvar. Obje funkcije nam za uneseni x vraćaju x + 5, jedino im je sintaksa malo drugačija.

Lambda funkciju možemo koristiti na sljedeći način, a upravo on će nam koristiti u daljnjem radu:

```
a = [2, 3, 4, 5]
print(map(lambda x: x*2, a))
```

Slika 19: Korištenje `map` i `lambda` funkcije

Prethodni primjer radi sljedeće: `lambda` funkcija svaki element iz liste a množi s 2, dok nam `map()` vraća novu listu s elementima koje nam je `lambda` vratila, tj. u ovom primjeru dobit ćemo listu [4, 6, 8, 10].

Sada, kada smo ukratko objasnili što su nam rječnici i lambda funkcija, krenimo s optimizacijom rječnika.

6.2 Optimizacija rječnika

Za primjer optimizacije rječnika, uzet ćemo funkciju koja prima neku riječ, prvo slovo te riječi pretvara u veliko slovo, a ostala slova zamjenjuje s odgovarajućim znamenkama. U našem primjeru to smo učinili na način da smo definirali rječnik pomoću kojeg smo svakom slovu pridružili odgovarajuću znamenku.

```python
slovo_u_broj = {"A":"1",
                "B":"2",
                "C":"3",
                "D":"4",
                "E":"1",
                "F":"5",
                "G":"3",
                "H":"2",
                "I":"1",
                "J":"2",
                "K":"5",
                "L":"6",
                "M":"4",
                "N":"3",
                "O":"1",
                "P":"2",
                "Q":"1",
                "R":"3",
                "S":"3",
                "T":"4",
                "U":"1",
                "V":"6",
                "W":"4",
                "X":"3",
                "Y":"1",
                "Z":"3"}

def rjecnici(rijec):
    rijec = rijec[0].upper() + rijec[1:]
    kod = rijec[0]
    for s in rijec[1:]:
        s = s.upper()
        kod += slovo_u_broj[s]
    return kod

if __name__ == '__main__':
    from timeit import Timer
    word = "osijek"
    statement = "rjecnici('%s')" % word
    t = Timer(statement, "from __main__ import rjecnici")
    print(word.ljust(15), rjecnici(word), min(t.repeat()))
```

```
osijek          031215 6.972532048070688
```

Slika 20: Korištenje rječnika

Promatramo li kod, možemo primjetiti kako pozivamo metodu `upper()` na svako pojedino slovo te gradimo naš kod postupno, znamenku po znamenku, tj. ulazeći u petlju, svaki put se nadovezujemo na stari string. To nam se baš i ne čini kao najbolje rješenje pa probajmo taj dio poboljšati tako da pozovemo funkciju `upper()` samo jednom, na cijeli string, a problem s petljom riješimo pomoću ranije spomenute, `l`ambda funkcije.

```
slovo_u_broj = {"A":"1",
                "B":"2",
                "C":"3",
                "D":"4",
                "E":"1",
                "F":"5",
                "G":"3",
                "H":"2",
                "I":"1",
                "J":"2",
                "K":"5",
                "L":"6",
                "M":"4",
                "N":"3",
                "O":"1",
                "P":"2",
                "Q":"1",
                "R":"3",
                "S":"3",
                "T":"4",
                "U":"1",
                "V":"6",
                "W":"4",
                "X":"3",
                "Y":"1",
                "Z":"3"}

def rjecnici(rijec):
    rijec = rijec.upper()
    kod = rijec[0]+"".join(map(lambda c: slovo_u_broj[c], rijec[1:]))
    return kod

if __name__ == '__main__':
    from timeit import Timer
    word = "osijek"
    statement = "rjecnici('%s')" % word
    t = Timer(statement, "from __main__ import rjecnici")
    print(word.ljust(15), rjecnici(word), min(t.repeat()))
```

```
osijek          O31215 8.490964060411631
```

Slika 21: Korištenje lambda funkcije

Iako smo možda očekivali kako će nam ovo rješenje biti brže, vidimo da je ono zapravo i sporije od onoga kojeg smo na početku imali (preko rječnika). No, to nam zapravo i jest bit optimizacije. Moramo izmjenjivati naš kod kako bismo dobili što brže rješenje. U nekim situacijama nam naše izmjene neće biti brže od početnog koda, ali to nećemo znati dok ne isprobamo.

Sljedeća ideja je da zamjenimo `lambda` funkciju s list comprehention.

```
slovo_u_broj = {"A":"1",
                "B":"2",
                "C":"3",
                "D":"4",
                "E":"1",
                "F":"5",
                "G":"3",
                "H":"2",
                "I":"1",
                "J":"2",
                "K":"5",
                "L":"6",
                "M":"4",
                "N":"3",
                "O":"1",
                "P":"2",
                "Q":"1",
                "R":"3",
                "S":"3",
                "T":"4",
                "U":"1",
                "V":"6",
                "W":"4",
                "X":"3",
                "Y":"1",
                "Z":"3"}

def rjecnici(rijec):
    rijec = rijec.upper()
    kod = rijec[0]+"".join([slovo_u_broj[c] for c in rijec[1:]])
    return kod

if __name__ == '__main__':
    from timeit import Timer
    word = "osijek"
    statement = "rjecnici('%s')" % word
    t = Timer(statement, "from __main__ import rjecnici")
    print(word.ljust(15), rjecnici(word), min(t.repeat()))
```

```
osijek          O31215 6.320183431605892
```

Slika 22: Korištenje list comprehention

Vidimo kako je list comprehention malo ubrzalo naš kod, ali nije toliko znatna razlika u brzini.

Preostaje nam još isprobati Pythonovu funkciju string.maketrans koja prima dva argumenta i prevodi prvi argument u drugi. Uvjet je da ti argumenti moraju biti iste duljine. Korištenjem te funkcije vidimo kako se i dalje istim slovima pridružuju iste znamenke, ali smo na ovaj način izbjegli korištenje rječnika te koristimo samo string metodu što nam daje još najbolje rezultate dosad, koje možemo pogledati na sljedećoj stranici.

```
import string, re

sva_slova = string.ascii_uppercase + string.ascii_lowercase
slovo_u_broj = str.maketrans(sva_slova, "1234153212564312133416431 3"*2)

def rjecnici(rijec):
    kod = rijec[0].upper() + rijec[1:].translate(slovo_u_broj)
    return kod

if __name__ == '__main__':
    from timeit import Timer
    word = "osijek"
    statement = "rjecnici('%s')" % word
    t = Timer(statement, "from __main__ import rjecnici")
    print(word.ljust(15), rjecnici(word), min(t.repeat()))
```

```
osijek          O31215 3.146422960805034
```

Slika 23: Korištenje funkcije string.maketrans

Možemo zaključiti kako je korištenje rječnika samo po sebi dosta brzo rješenje i ne trebamo ga posebno optimizirati, jedino ukoliko rječnike možemo zamijeniti s nekom posebnom funkcijom, kao što je u ovom slučaju string.maketrans, tada ćemo doći do bržeg rješenja.

Zaključak

Na kraju ovog rada, možemo zaključiti nekoliko bitnih stvari koje smo mogli vidjeti u optimizaciji naših kodova:

- testirajte svoj kod - koje god izmjene radili u kodu, uvijek je bitno da ga testirate, jedino tada ćete znati imaju li te izmjene uopće smisla
- izbjegavajte petlje - kad god je to moguće, izbjegnite pisanje petlji. Petlje probajte zamijeniti s nekim jednostavnijim naredbama
- učinite kod što jednostavnijim - kako i nama, tako će i Pythonu ponekad biti lakše (a time i brže) pročitati jednostavniji kod
- regularne izraze zamijenite sa string metodom - ukoliko znate string metodu kojom možete zamijeniti regularni izraz u kodu, a da kod i dalje radi istu stvar, napravite to! Da se vratimo na prethodnu natuknicu, string metode je jednostavno lakše za razumjeti
- rječnike, ako znate, možete zamijeniti s nekom posebnom funkcijom, ali oni su sami po sebi već dosta brzi

No, ne zaboravite najbitniju stvar od svih! Prije nego što uopće krenete s optimizacijom, dobro odvagnite je li ona uistinu potrebna. Ukoliko jest, nadamo se kako će vam ove smjernice biti od koristi.

Literatura

[1] http://pymotw.com/2/timeit/
[2] http://pypy.org/performance.html#optimization-strategy
[3] http://www.algorithm.co.il/blogs/computer-science/10-python-optimization-tips-and-issues/
[4] http://www.renderx.com/files/demos/examples/diveintopython.pdf
[5] http://www.secnetix.de/olli/Python/lambda_functions.hawk
[6] https://bs.wikipedia.org/wiki/Optimizacija_programskog_koda
[7] https://docs.python.org/2/library/re.html
[8] https://docs.python.org/2/library/string.html
[9] https://docs.python.org/2/library/timeit.html