Sadržaj

Predgovor ..... **xi**

1. Strukture podataka i algoritmi **1**

1.1. Raspakivanje sekvence u zasebne promenljive 1
1.2. Raspakivanje elemenata iz iterabilne strukture proizvoljne dužine 2
1.3. Zadržavanje poslednjih N stavki 5
1.4. Nalaženje najvećih ili najmanjih N stavki 7
1.5. Implementiranje reda s prioritetom 8
1.6. Mapiranje ključeva na više vrednosti u rečniku 10
1.7. Uređivanje rečnika 12
1.8. Računske operacije nad rečnikom 13
1.9. Nalaženje zajedničkih osobina dva rečnika 14
1.10. Uklanjanje duplih vrednosti iz sekvence uz očuvanje redosleda 15
1.11. Imenovanje isečka 17
1.12. Nalaženje najčešćih stavki u sekvenci 18
1.13. Sortiranje liste rečnika po zajedničkom ključu 20
1.14. Sortiranje objekata koji izvorno ne podržavaju poređenje 21
1.15. Grupisanje zapisa na osnovu polja 22
1.16. Filtriranje elemenata sekvence 24
1.17. Izdvajanje podskupa rečnika 26
1.18. Mapiranje imena na elemente sekvence 27
1.19. Istovremeno transformisanje i redukovanje podataka 30
1.20. Kombinovanje više mapiranja u jedno 31

2. Znakovni nizovi i tekst **35**

2.1. Deljenje znakovnih nizova po različitim graničnicima 35
2.2. Podudaranje teksta na početku ili kraju znakovnog niza 36
2.3. Ispitivanje podudarnosti znakovnih nizova pomoću džokerskih obrazaca ljuski 38
2.4. Podudaranje i traženje tekstualnih obrazaca 39
2.5. Pretraživanje i zamena teksta 42
2.6. Pretraživanje i zamena teksta bez obzira na veličinu slova 44
2.7. Definisanje regularnih izraza za najkraće podudaranje 45
2.8. Pisanje regularnog izraza za višeredne obrasce 46
2.9. Normalizovanje Unicode teksta za standardan način predstavljanja 47

2.10. Rad sa Unicode znakovima u regularnim izrazima 48
2.11. Izuzimanje neželjenih znakova iz znakovnih nizova 49
2.12. Pročišćavanje teksta 51
2.13. Poravnavanje tekstualnih znakova 53
2.14. Kombinovanje i nadovezivanje znakovnih nizova 55
2.15. Interpoliranje promenljivih u znakovnim nizovima 57
2.16. Reformatiranje teksta radi dobijanja fiksnog broja kolona 60
2.17. Rad sa HTML i XML entitetima u tekstu 61
2.18. Deljenje teksta u tokene 62
2.19. Pisanje jednostavnog raščlanjivača rekurzivnog svođenja 65
2.20. Obavljanje tekstualnih operacija nad bajtnim znakovnim nizovima 73

3. Brojevi, datumi i vremena 77
3.1. Zaokruživanje numeričkih vrednosti 77
3.3. Formatiranje brojeva za ispisivanje 80
3.4. Rad s binarnim, oktalnim i heksadecimalnim celim brojevima 82
3.5. Pakovanje i raspakivanje velikih celih brojeva iz bajtova 84
3.6. Obavljanje matematičkih operacija nad kompleksnim brojevima 85
3.7. Rad s beskonačnom i NaN vrednošću 87
3.8. Matematičke operacije s razlomcima 89
3.9. Računske operacije s velikim numeričkim nizovima 90
3.10. Obavljanje računskih operacija nad matricama i operacija linearne algebre 93
3.11. Nasumično biranje stavki 94
3.12. Pretvaranje dana u sekunde i druge osnovne konverzije vremena 96
3.13. Određivanje datuma poslednjeg petka 98
3.14. Nalaženje opsega datuma za tekući mesec 100
3.15. Konverzija znakovnih nizova u objekte tipa datetime 101
3.16. Upravljanje datumima pri radu s vremenskim zonama 102

4. Iteratori i generatori 105
4.1. Ručno konzumiranje iteratora 105
4.2. Delegiranje iteracija 106
4.3. Pravljenje novih obrazaca iteracije pomoću generatora 107
4.4. Implementiranje protokola iteracije 109
4.5. Obrnuta iteracija 111
4.6 Definisanje generatorskih funkcija s dodatnim stanjem 112
4.7. Uzimanje isečka iteratora 113
4.8. Preskakanje prvog dela iterabilne strukture 114
4.9. Iteracije nad svim mogućim kombinacijama ili permutacijama 116
4.10. Iteracije nad parovima indeks/vrednost u sekvenci 118
4.11. Istovremene iteracije nad više sekvenci 120
4.12. Iteracije nad stavkama u zasebnim kontejnerima 122
4.13. Pravljenje kanala za obradu podataka 123

4.14. Poravnavanje ugnežđene sekvence 126
4.15. Iteracije nad spojenim uređenim iterabilnim strukturama 127
4.16. Zamena beskonačnih petlji while iteratorom 128

5. Datoteke i I/O operacije 131
5.1. Čitanje i pisanje tekstualnih podataka 131
5.2. Upisivanje u datoteku 134
5.3. Ispisivanje s drugačijim separatorom ili krajem reda 134
5.4. Čitanje i upisivanje binarnih podataka 135
5.5. Upisivanje u datoteku koja još uvek ne postoji 137
5.6. Izvođenje I/O operacija nad znakovnim nizovima 138
5.7. Čitanje i upisivanje u komprimovane datoteke s podacima 139
5.8. Iteracije nad zapisima fiksne veličine 140
5.9. Učitavanje binarnih podataka u promenljivi bafer 141
5.10. Memorijsko mapiranje binarnih datoteka 143
5.11. Upravljanje imenima putanja 145
5.12. Ispitivanje postojanja datoteke 146
5.13. Dobavljanje listinga sadržaja direktorijuma 147
5.14. Premošćavanje kodiranja imena datoteke 148
5.15. Ispisivanje neispravnih imena datoteka 150
5.16. Dodavanje ili menjanje kodiranja već otvorene datoteke 152
5.17. Upisivanje bajtova u tekstualnu datoteku 154
5.18. Omotavanje postojećeg deskriptora datoteke objektom datoteke 154
5.19. Pravljenje privremenih datoteka i direktorijuma 156
5.20. Komunikacija putem serijskih priključaka 158
5.21. Serijalizovanje Pythonovih objekata 159

6. Kodiranje i obrada podataka 163
6.1. Čitanje i upisivanje CSV podataka 163
6.2. Čitanje i upisivanje JSON podataka 166
6.3. Raščlanjivanje jednostavnih XML podataka 171
6.4. Postupno raščlanjivanje velikih XML datoteka 173
6.5. Pretvaranje rečnika u XML dokument 176
6.6. Raščlanjivanje, izmena i ponovno upisivanje XML podataka 178
6.7. Raščlanjivanje XML dokumenata sa imenskim prostorima 180
6.8. Interakcija sa relacionim bazama podataka 182
6.9. Dekodiranje i kodiranje heksadecimalnih cifara 184
6.10. Dekodiranje i kodiranje putem sistema Base64 185
6.11. Čitanje i upisivanje binarnih nizova struktura 186
6.12. Čitanje ugnežđenih binarnih struktura i binarnih struktura promenljive veličine 190
6.13. Svođenje podataka i statistička analiza 199

7. Funkcije 203
7.1. Pisanje funkcija s proizvoljnim brojem argumenata 203
7.2. Pisanje funkcija koje prihvataju samo argumente rezervisanih reči 204
7.3. Pridruživanje informativnih metapodataka argumentima funkcije 205
7.4. Funkcija s više povratnih vrednosti 206
7.5. Definisanje funkcija s podrazumevanim argumentima 207
7.6. Definisanje anonimnih ili ugrađenih funkcija 210
7.8. Kako da pozivljivi entitet sa N argumenata funkcioniše kao pozivljivi entitet s manje argumenata 212
7.9. Zamena klasa s jednom metodom funkcijama 215
7.10. Povratne funkcije sa dodatnim stanjem 216
7.11. Ugrađivanje povratnih funkcija 219
7.12. Pristupanje promenljivama definisanim unutar zatvaranja 222

8. Klase i objekti 227
8.1. Izmena formata instanci predstavljenih u vidu znakovnih nizova 227
8.2. Prilagođavanje formatiranja znakovnog niza 228
8.3. Uspostavljanje podrške objekata za protokol upravljanja kontekstom 230
8.4. Ušteda u memoriji prilikom pravljenja velikog broja instanci 232
8.5. Kapsuliranje imena u klasi 233
8.6. Pravljenje upravljanih atributa 234
8.7. Pozivanje metode roditeljske klase 238
8.8. Proširivanje svojstva u potklasi 242
8.9. Pravljenje nove vrste atributa klase ili instance 246
8.10. Primena lenjo izračunatih svojstava 249
8.11. Pojednostavljivanje inicijalizacije struktura podataka 252
8.12. Definisanje interfejsa ili apstraktne osnovne klase 255
8.13. Implementiranje modela podataka ili sistema tipa 258
8.14. Implementiranje namenskih kontejnera 264
8.15. Delegiranje pristupa atributu 267
8.16. Definisanje više konstruktora u klasi 271
8.17. Pravljenje instance bez pozivanja metode *init* 273
8.18. Proširivanje klasa pomoću miksin klasa 274
8.19. Implementiranje objekata sa informacijama o stanju ili mašina stanja 279
8.20. Pozivanje metode nad objektom s prosleđenim imenom u vidu znakovnog niza 284
8.21. Implementiranje obrasca posetioca 285
8.22. Implementiranje obrasca posetioca bez rekurzije 290
8.23. Upravljanje memorijom u cikličnim strukturama podataka 296
8.24. Uspostavljanje podrške klasa operacijama poređenja 299
8.25. Pravljenje keširanih instanci 301

9. Metaprogramiranje.....305
9.1. Omotavanje funkcije 305
9.2. Očuvanje metapodataka funkcije prilikom pisanja dekoratora 307
9.3. Odmotavanje dekoratora 308
9.4. Definisanje dekoratora sa argumentima 310
9.5. Definisanje dekoratora s prilagodljivim atributima 311
9.6. Definisanje dekoratora sa opcionim argumentom 314
9.7. Nametanje provere tipa funkciji pomoću dekoratora 316
9.8. Definisanje dekoratora kao dela klase 319
9.9. Definisanje dekoratora kao klasa 321
9.10. Primena dekoratora na metode klase i statičke metode 324
9.11. Pisanje dekoratora koji dodaju argumente omotanim funkcijama 326
9.12. Primena dekoratora za krpljenje definicija klasa 328
9.13. Primena metaklase za kontrolu pravljenja instance 330
9.14. Pamćenje redosleda definisanja atributa klase 332
9.15. Definisanje metaklase sa opcionim argumentima 335
9.16. Nametanje potpisa argumenata sintakse *args/**kwargs 337
9.17. Nametanje konvencija kodiranja u klasama 340
9.18. Programsko definisanje klasa 343
9.19. Inicijalizovanje članova klase prilikom definisanja 346
9.20. Preklapanje metoda pomoću anotacija funkcija 348
9.21. Izbegavanje repetitivnih metoda svojstava 354
9.22. Definisanje upravljača kontekstom na lak način 355
9.23. Izvršavanje koda s lokalnim sporednim efektima 357
9.24. Raščlanjivanje i analiziranje Pythonovog izvornog koda 360
9.25. Rastavljanje Pythonovog bajtnog koda 364

10. Moduli i paketi.....367
10.1. Pravljenje paketa modula s hijerarhijom 367
10.2. Kontrolisanje uvoza svega 368
10.3. Uvoz podmodula paketa pomoću relativnih imena 369
10.4. Deljenje modula u više datoteka 370
10.5. Uvoz zasebnih direktorijuma koda pod zajedničkim imenskim prostorom 373
10.6. Ponovno učitavanje modula 375
10.7. Omogućavanje izvršavanja direktorijuma ili .zip datoteke kao glavnog skripta 376
10.8. Čitanje datoteka s podacima u okviru paketa 377
10.9. Dodavanje direktorijuma promenljivoj sys.path 378
10.10. Uvoženje modula pomoću imena navedenog u znakovnom nizu 380
10.11. Učitavanje modula iz udaljenih mašina pomoću udica za uvoz 381
10.12. Krpljenje modula prilikom uvoza 396
10.13. Instaliranje paketa samo za sebe 399
10.14. Pravljenje novog Pythonovog okruženja 399
10.15. Distribuiranje paketa 401

11. Mrežno i veb programiranje 403
11.1. Interakcija sa HTTP servisima iz pozicije klijenta 403
11.2. Pravljenje TCP servera 407
11.3. Pravljenje UDP servera 410
11.4. Generisanje opsega IP adresa iz CIDR adrese 412
11.5. Pravljenje jednostavnog REST interfejsa 414
11.6. Implementiranje jednostavnog daljinskog pozivanja procedura pomoću protokola XML-RPC 418
11.7. Komuniciranje između interpretera 421
11.8. Implementiranje daljinskih poziva procedura 422
11.9. Jednostavna provera identiteta klijenata 426
11.10. Dodavanje SSL sloja mrežnim servisima 428
11.11. Razmena deskriptora datoteke utičnice između procesa 434
11.12. I/O operacije upravljane događajima 439
11.13. Slanje i primanje velikih nizova 444

12. Paralelan rad 447
12.1. Pokretanje i zaustavljanje niti 447
12.2. Određivanje da li je nit počela da se izvršava 450
12.3. Komunikacija između niti 453
12.4. Zaključavanje kritičnih delova 458
12.5. Zaključavanje sa izbegavanjem kružne blokade 461
12.6. Skladištenje stanja niti 464
12.7. Pravljenje skladišta niti 466
12.8. Izvođenje jednostavnog paralelnog programiranja 469
12.9. Rad sa GIL bravom (i kako da prestanete da brinete o njoj) 473
12.10. Definisanje zadatka učesnika 476
12.11. Implementiranje mehanizma slanja poruka sa objavljivanjem/pretplatom 480
12.12. Primena generatora kao alternativa nitima 483
12.13. Anketiranje većeg broja redova niti 490
12.14. Pokretanje sistemskog procesa na Unixu 493

13. Pomoćni skriptovi i administriranje sistema 497
13.1. Prihvatanje ulaznih podataka za skript putem preusmeravanja, cevi ili ulaznih datoteka 497
13.2. Završavanje programa s porukom o grešci 498
13.3. Raščlanjivanje opcija komandne linije 499
13.4. Traženje lozinke prilikom izvršavanja 501
13.5. Određivanje veličine terminala 502
13.7. Kopiranje ili prenošenje datoteka i direktorijuma 504
13.8. Pravljenje i raspakivanje arhiva 506
13.9. Nalaženje datoteka po imenu 507
13.10. Čitanje konfiguracionih datoteka 509

13.12. Dodavanje evidentiranja bibliotekama 514
13.13. Pravljenje štoperice 516
13.14. Ograničavanje korišćenja memorije i procesora 518
13.15. Pokretanje veb čitača 519

14. Testiranje, otklanjanje grešaka i izuzeci 521
14.1. Testiranje izlaznih rezultata poslatih u stdout 521
14.2. Krpljenje objekata u jediničnim testovima 522
14.3. Ispitivanje mogućnosti pojave izuzetaka u jediničnim testovima 525
14.4. Evidentiranje rezultata testa u datoteku 527
14.5. Izbegavanje ili predviđanje neuspeha testova 528
14.6. Obrada većeg broja izuzetaka 530
14.7. Obrada svih izuzetaka 531
14.8. Pravljenje namenskih izuzetaka 533
14.9. Generisanje izuzetka kao reakcije na drugi izuzetak 534
14.10. Ponovno generisanje poslednjeg izuzetka 537
14.11. Izdavanje upozoravajućih poruka 537
14.12. Otklanjanje osnovnih uzroka otkazivanja programa 539
14.13. Profilisanje i merenje vremena izvršavanja programa 541
14.14. Ubrzavanje izvršavanja programa 544

15. Proširenja na C-u 549
15.1. Pristupanje kodu na C-u pomoću modula ctypes 550
15.2. Pisanje jednostavnog modula proširenja na C-u 556
15.3. Pisanje funkcije proširenja koja radi s nizovima 560
15.4. Rad sa neprovidnim pokazivačima u modulima proširenja na C-u 563
15.5. Definisanje i izvoz API interfejsa C-a iz modula proširenja 565
15.6. Pozivanje Pythona iz C-a 569
15.7. Oslobađanje GIL brave u proširenjima na C-u 574
15.8. Mešanje niti iz C-a i Pythona 575
15.9. Omotavanje koda na C-u pomoću Swiga 576
15.10. Omotavanje postojećeg koda na C-u pomoću Cythona 581
15.11. Korišćenje Cythona za pisanje operacija visokih performansi nad nizovima 587
15.12. Pretvaranje pokazivača na funkciju u pozivljivi entitet 591
15.13. Prosleđivanje znakovnih nizova s NULL završetkom C bibliotekama 593
15.14. Prosleđivanje Unicode znakovnih nizova bibliotekama jezika C 597
15.15. Prevođenje C-ovih znakovnih nizova na Python 601
15.16. Rad sa C-ovim znakovnim nizovima s nepoznatim kodiranjem 602
15.17. Prosleđivanje imena datoteka proširenjima na C-u 605
15.18. Prosleđivanje otvorenih datoteka proširenjima na C-u 606
15.19. Čitanje objekata sličnih datoteci iz C-a 607
15.20. Konzumiranje iterabilne strukture iz C-a 610
15.21. Dijagnostika grešaka u segmentaciji 611

A. Dodatni izvori 613

Spisak termina korišćenih u knjizi 615

Indeks 617

POGLAVLJE 1

Strukture podataka i algoritmi

Python nudi raznovrsne korisne ugrađene strukture podataka kao što su liste, skupovi i rečnici. U najvećem broju slučajeva, ove strukture se jednostavno koriste. Međutim, često se javljaju nedoumice u vezi sa pretraživanjem, sortiranjem, uređivanjem redosleda i filtriranjem. Zato su tema ovog poglavlja uobičajene strukture podataka i algoritmi za rad s podacima. Pored toga, osvrnućemo se na razne strukture podataka u modulu `collections`.

1.1. Raspakivanje sekvence u zasebne promenljive

Problem

Želite da razložite n-torku (engl. *tuple*) ili sekvencu na kolekciju N promenljivih.

Rešenje

Svaka sekvenca (ili iterabilna struktura) može se raspakovati na promenljive putem jednostavne operacije dodeljivanja. Neophodno je jedino da broj promenljivih i struktura odgovaraju sekvenci. Na primer:

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>

>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, date = data
>>> name
'ACME'
>>> date
(2012, 12, 21)

>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
```

```
2012
>>> mon
12
>>> day
21
>>>
```

Ako se broj elemenata ne poklapa, javlja se greška. Na primer:

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>
```

Objašnjenje

Svaki iterabilan objekat može se raspakovati, ne samo n-torke ili liste. Dakle, to važi za znakovne nizove (engl. *strings*), datoteke, iteratore i generatore. Na primer:

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>
```

Prilikom raspakivanja, možda ćete hteti da odbacite neke vrednosti. Python nema posebnu sintaksu za to, ali često je dovoljno da u tu svrhu definišete promenljivu za bacanje. Na primer:

```
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

Ipak, morate se postarati da se odabrano ime promenljive ne koristi već za nešto drugo.

1.2. Raspakivanje elemenata iz iterabilne strukture proizvoljne dužine

Problem

Treba da raspakujete N elemenata iz iterabilne strukture, ali je moguće da ona ima više od N elemenata. U tom slučaju generiše se izuzetak „too many values to unpack“ „previše vrednosti za raspakivanje“).

Rešenje

Za rešenje ovog problema možete iskoristiti Pythonove „izraze sa zvezdicama“. Recimo da držite neki kurs, i da ste odlučili da prilikom zaključivanja ocene računate prosek ocena zadataka bez prvog i poslednjeg zadatka. Ako su bila četiri zadatka, lako ćete raspakovati sva četiri, ali šta ako ih je 24? Izraz sa zvezdicama olakšaće vam posao:

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

Evo drugog primera: pretpostavimo da radite s korisničkim zapisima (engl. *records*) koji se sastoje od imena i e-adrese, praćenim proizvoljnim brojem telefonskih brojeva. Zapise biste mogli da raspakujete na sledeći način:

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
>>> name, email, *phone_numbers = user_record
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```

Treba istaći da će promenljiva `phone_numbers` uvek biti lista, koliko god telefonskih brojeva raspakivali (pa i nijedan). Zato ni za jedan kôd u kome se koristi `phone_numbers` ne morate da brinete hoće li to biti lista niti da obavljate bilo kakvu dodatnu proveru tipa.

Promenljiva sa zvezdicom može biti i prva u listi. Uzmimo kao primer sekvencu numeričkih vrednosti koje izražavaju ostvarenu prodaju vaše kompanije za poslednjih osam kvartala. Ako želite da proverite kako stoji poslednji kvartal spram proseka prethodnih sedam kvartala, mogli biste da uradite nešto slično ovome:

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

Evo te operacije iz Pythonovog interpretera:

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

Objašnjenje

Prošireno raspakivanje je kao stvoreno za razlaganje iterabilnih struktura nepoznate ili proizvoljne veličine. Takve strukture često imaju neku poznatu komponentu ili obrazac (engl. *pattern*) u konstrukciji (na primer, „sve posle prvog elementa je telefonski broj“), i raspakivanje sa zvezdicom omogućava programeru da iskoristi te obrasce, umesto da se dovija kako bi izdvojio relevantne elemente iterabilne strukture.

Skrećemo pažnju i na to da sintaksa sa zvezdicom može da bude naročito korisna prilikom iteracije nad sekvencom n-torki različite veličine. Evo primera sa sekvencom označenih n-torki:

```
records = [
     ('foo', 1, 2),
     ('bar', 'hello'),
     ('foo', 3, 4),
]

def do_foo(x, y):
    print('foo', x, y)

def do_bar(s):
    print('bar', s)

for tag, *args in records:
    if tag == 'foo':
        do_foo(*args)
    elif tag == 'bar':
        do_bar(*args)
```

Raspakivanje sa zvezdicom može biti od koristi i u kombinaciji sa određenim operacijama obrade znakovnih nizova, kao što je njihova podela (engl. *splitting*). Na primer:

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

U nekim slučajevima, biće potrebno da raspakujete vrednosti i da ih onda odbacite. Ne možete samo da navedete * prilikom raspakivanja, ali biste mogli da upotrebite uobičajeno ime promenljive za odbacivanje, poput _ ili `ign` („ignorisati“). Na primer:

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_, (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

Postoje određene sličnosti između raspakivanja sa zvezdicom i mehanizama za obradu lista različitih funkcionalnih jezika. Na primer, ukoliko radite sa listom, lako je možete razložiti (podeliti) na početne (engl. *head*) i prateće (engl. *tail*) komponente:

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
```

```
[10, 7, 4, 5, 9]
>>>
```

Moguće je i pisati funkcije koje obavljaju takvo rastavljanje da biste dobili koristan rekurzivni algoritam. Na primer:

```
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

Međutim, imajte na umu da rekurzija i nije jača Pythonova strana zbog ugrađenog ograničenja za rekurziju. Zato je poslednji primer možda tek teorijski kuriozitet primenjen u praksi.

1.3. Zadržavanje poslednjih N stavki

Problem

Želite ograničenu istoriju poslednjih nekoliko stavki viđenih prilikom iteracije ili nekog drugog postupka obrade.

Rešenje

Ograničeno praćenje istorije je savršena prilika da se upotrebi kontejner `collections.deque`. Na primer, naredni kôd ispituje podudaranje jednostavnog teksta sa sekvencom redova, i vraća podudaran red zajedno s prethodnih N redova konteksta:

```
from collections import deque

def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Primer primene na datoteku
if __name__ == '__main__':
    with open('somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
            print(line, end='')
            print('-'*20)
```

Objašnjenje

Prilikom pisanja koda za traženje stavki, uobičajeno je da se koristi generatorska funkcija `yield`, kao što je pokazano u rešenju ovog recepta. Na ovaj način, razdvaja se proces pretraživanja od koda koji koristi rezultate. Ako se do sada niste sretali s generatorskim funkcijama, pogledajte Recept 4.3.

Izraz `deque(maxlen=N)` pravi red (engl. *queue*) fiksne veličine. Kada se nove stavke dodaju redu koji je pun, automatski se uklanja najstarija stavka. Na primer:

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

Iako biste takve operacije (recimo, dodavanje, brisanje itd.) mogli ručno da izvodite nad listom, rešenje s redom je mnogo elegantnije i brže.

Uopšte uzev, `deque` se može upotrebiti kad god vam je potrebna jednostavna struktura reda. Ako ne zadate maksimalnu veličinu, dobićete neograničen red koji dozvoljava da dodajete i uklanjate stavke na početku i na kraju. Na primer:

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4
```

Dodavanje ili uklanjanje stavki s bilo kog kraja reda jeste operacija složenosti O(1). Umetanje stavke na početak ili na kraj liste, ili njeno uklanjanje odatle, ima složenost O(N).

1.4. Nalaženje najvećih ili najmanjih N stavki

Problem

Hoćete da napravite listu najvećih ili najmanjih N stavki u kolekciji.

Rešenje

Modul `heapq` ima dve funkcije – `nlargest()` i `nsmallest()` – koje rade upravo ono što hoćete. Na primer:

```
import heapq

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums))  # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]
```

Obe funkcije prihvataju i ključan (engl. *key*) parametar koji omogućava da se koriste sa složenijim strukturama podataka. Evo primera:

```
portfolio = [
   {'name': 'IBM', 'shares': 100, 'price': 91.1},
   {'name': 'AAPL', 'shares': 50, 'price': 543.22},
   {'name': 'FB', 'shares': 200, 'price': 21.09},
   {'name': 'HPQ', 'shares': 35, 'price': 31.75},
   {'name': 'YHOO', 'shares': 45, 'price': 16.35},
   {'name': 'ACME', 'shares': 75, 'price': 115.65}
]

cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
```

Objašnjenje

Ako tražite N najmanjih ili najvećih stavki, a vrednost N je mala u poređenju s veličinom kolekcije, ove funkcije omogućavaju vrhunske performanse. Prvi korak je da konvertuju podatke u listu čiji su elementi uređeni u vidu hipa (engl. *heap*). Na primer:

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

Najvažnija odlika hipa je da je `heap[0]` uvek najmanji element. Povrh toga, naredne stavke se lako mogu naći pomoću metode `heapq.heappop()` koja prvi element zamenjuje sledećim najmanjim elementom (što je postupak koji zahteva O(log N) operacija, gde je N veličina hipa). Na primer, tri najmanje stavke našli biste na sledeći način:

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
```

```
1
>>> heapq.heappop(heap)
2
```

Funkcije `nlargest()` i `nsmallest()` najprigodnije su kad vam je cilj da nađete relativno malo stavki. Ukoliko hoćete da otkrijete samo jednu stavku, najveću ili najmanju, (N=1), korišćenjem funkcije `min()` i `max()` ubrzaćete stvari. S druge strane, ako je vrednost N slična veličini kolekcije, obično će biti brže da se kolekcija prvo sortira i da se izdvoji isečak (to jest, da pozovete funkciju `sorted(items)[:N]` ili `sorted(items)[-N:]`). Valja istaći da se u konkretnoj primeni, funkcije `nlargest()` i `nsmallest()` prilagođavaju i izvešće neke od ovih optimizacija umesto vas (na primer, primeniće sortiranje ukoliko je vrednost N slična veličini kolekcije).

Iako nije neophodno da koristite ovaj recept, primena hipa je zanimljiva i vredi je proučiti – objašnjenja ćete naći u svakoj valjanoj knjizi o algoritmima i strukturama podataka. I u dokumentaciji za modul `heapq` razmatraju se detalji primen.

1.5. Implementiranje reda s prioritetom

Problem

Želite da implementirate red koji stavke sortira po zadatom prioritetu i uvek vraća stavku najvišeg prioriteta pri svakoj operaciji uklanjanja (engl. *pop*).

Rešenje

Naredna klasa koristi modul `heapq` da bi realizovala jednostavan red s prioritetom (engl. *priority queue*):

```
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]
```

Evo primera kako bi se to moglo primeniti:

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
```

```
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

Prva operacija `pop()` vratila je element najvišeg prioriteta. Pored toga, dve stavke istog prioriteta (`foo` i `grok`) vraćene su redom kojim su umetnute u red.

Objašnjenje

Srž ovog recepta je primena modula `heapq`. Funkcije `heapq.heappush()` i `heapq.heappop()` umeću odnosno uklanjaju elemente liste `_queue` tako da prvi element u listi ima najmanji prioritet (kao što je objašnjeno u Receptu 1.4). Rezultat metode `heappop()` uvek je „najmanja" stavka, i to je najbitnije da bi se iz reda ispravno izdvajale tražene stavke. Povrh toga, kako je složenost operacija `push()` i `pop()` O(log N), pri čemu je N broj elemenata u hipu, prilično su efikasne i za veće vrednosti N.

U ovom receptu, red se sastoji od n-torki oblika `(-priority, index, item)`. Nad vrednošću `priority` izvršava se negacija da bi red sortirao stavke od najvišeg do najnižeg prioriteta. To je suprotno uobičajenom uređenju hipa po kome se elementiređaju od najniže do najviše vrednosti.

Uloga promenljive `index` je da omogući ispravno uređenje stavki istog prioriteta. Indeks se neprestano uvećava, pa će se elementi sortirati redom kojim su umetnuti. Međutim, indeks ima važnu ulogu i pri poređenju stavki istog nivoa prioriteta.

Da bi vam bilo jasnije, pogledajte naredni primer u kome se instance klase `Item` ne mogu urediti:

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

Ako napravite dvojke `(priority, item)`, one se mogu porediti dok god imaju različite prioritete. Međutim, ukoliko se porede dvojke istog prioriteta, poređenje će opet biti neuspešno. Na primer:

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
```

```
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

Ovaj problem ćete potpuno izbeći uvođenjem dodatnog indeksa i trojki `(priority, index, item)`, jer dve trojke nikada neće imati istu vrednost `index` (a Python se nikada ne upušta u poređenje preostalih vrednosti n-torke kad se već može utvrditi rezultat poređenja):

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
True
>>>
```

Ako ovakav red hoćete da upotrebite za komunikaciju između niti (engl. *threads*), morate dodati odgovarajuće zaključavanje i signaliziranje. Primer za to naći ćete u Receptu 12.3.

Dokumentacija za modul `heapq` sadrži dodatne primere i objašnjenja u vezi sa teorijom i primenom hipa.

1.6. Mapiranje ključeva na više vrednosti u rečniku

Problem

Hoćete da napravite rečnik koji mapira (preslikava) ključeve na više vrednosti (takozvani „multirečnik“).

Rešenje

U osnovi rečnika je mapiranje pri kome se svaki ključ mapira na jednu vrednost. Ukoliko želite da ključeve mapirate na više vrednosti, potrebno je da te vrednosti sačuvate u nekom drugom kontejneru – poput liste ili skupa. Na primer, pravite ovakve rečnike:

```
d = {
   'a' : [1, 2, 3],
   'b' : [4, 5]
}

e = {
   'a' : {1, 2, 3},
   'b' : {4, 5}
}
```

POGLAVLJE 8

Klase i objekti

Osnovni fokus ovog poglavlja je predstavljanje recepata za uobičajene šeme programiranja vezane za definicije klasa. Između ostalog, govorimo o podršci objekata za uobičajene Pythonove alatke, primeni specijalnih metoda, tehnikama kapsuliranja, nasleđivanju, upravljanju memorijom i korisnim šemama projektovanja.

8.1. Izmena formata instanci predstavljenih u vidu znakovnih nizova

Problem

Hoćete da izmenite rezultat ispisivanja ili pregledanja instanci u nešto prigodnije.

Rešenje

Da biste instancu, umesto kao znakovni niz, predstavili u drugom obliku, definišite metode `__str__()` i `__repr__()`. Na primer:

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)
    def __str__(self):
        return '({0.x!s}, {0.y!s})'.format(self)
```

Metoda `__repr__()` vraća detalje koda za predstavljanje instance, a to je obično tekst koji biste upisali da ponovo napravite instancu. Ugrađena funkcija `repr()` vraća ovaj tekst, kao i interaktivni interpreter prilikom ispitivanja vrednosti. Metoda `__str__()` konvertuje instancu u znakovni niz, što je rezultat koji vraćaju funkcije `str()` i `print()`. Na primer:

```
>>> p = Pair(3, 4)
>>> p
Pair(3, 4)          # Rezultat metode __repr__()
>>> print(p)
(3, 4)              # Rezultat metode __str__()
>>>
```

Ovaj recept pokazuje i kako se različiti načini predstavljanja znakovnog niza mogu upotrebiti prilikom formatiranja. Konkretno, kôd formata `!r` označava da bi podrazumevano trebalo da koristi rezultat metode `__repr__()`, a ne metode `__str__()`. Mogli biste da napravite eksperiment s prethodnom klasom:

```
>>> p = Pair(3, 4)
>>> print('p is {0!r}'.format(p))
p is Pair(3, 4)
>>> print('p is {0}'.format(p))
p is (3, 4)
>>>
```

Objašnjenje

Definisanje metoda `__repr__()` i `__str__()` često je dobra praksa, jer može da pojednostavi otkrivanje i otklanjanje grešaka i prikazivanje instance. Na primer, dovoljno je da se instanca ispiše ili evidentira, pa da programer dobije dodatne korisne informacije o sadržaju instance.

Uobičajeno je da metoda `__repr__()` generiše tekst kao što je `eval(repr(x)) == x`. Ukoliko to nije moguće ili nije željeno ponašanje, obično se umesto toga pravi korisna tekstualna reprezentacija između oznaka `<` i `>`. Na primer:

```
>>> f = open('file.dat')
>>> f
<_io.TextIOWrapper name='file.dat' mode='r' encoding='UTF-8'>
>>>
```

Ukoliko metoda `__str__()` nije definisana, kao zamena se koristi rezultat metode `__repr__()`.

Primena funkcije `format()` u rešenju možda je pomalo neočekivana, ali kôd formata `{0.x}` zadaje atribut `x` argumenta 0. Zato je, u narednoj funkciji, `0` zapravo instanca `self`:

```
def __repr__(self):
    return 'Pair({0.x!r}, {0.y!r})'.format(self)
```

Umesto ovakve implementacije, mogli ste da primenite operator `%` i naredni kôd:

```
def __repr__(self):
    return 'Pair(%r, %r)' % (self.x, self.y)
```

8.2. Prilagođavanje formatiranja znakovnog niza

Problem

Želite da objekat podržava prilagođeno formatiranje pomoću funkcije i metode znakovnih nizova `format()`.

Rešenje

Da biste prilagodili formatiranje znakovnih nizova, definišite metodu `__format__()` u klasi. Na primer:

```
_formats = {
    'ymd' : '{d.year}-{d.month}-{d.day}',
    'mdy' : '{d.month}/{d.day}/{d.year}',
    'dmy' : '{d.day}/{d.month}/{d.year}'
    }

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __format__(self, code):
        if code == '':
            code = 'ymd'
        fmt = _formats[code]
        return fmt.format(d=self)
```

Instance klase `Date` sada podržavaju operacije formatiranja poput narednih:

```
>>> d = Date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, 'mdy')
'12/21/2012'
>>> 'The date is {:ymd}'.format(d)
'The date is 2012-12-21'
>>> 'The date is {:mdy}'.format(d)
'The date is 12/21/2012'
>>>
```

Objašnjenje

Metoda `__format__()` je spona sa Pythonovom funkcionalnošću za formatiranje znakovnih nizova. Važno je znati da tumačenje kodova formatiranja potpuno zavisi od same klase. Dakle, kodovi mogu biti gotovo bilo šta. Na primer, razmotrimo naredni kôd s modulom `datetime`:

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d,'%A, %B %d, %Y')
'Friday, December 21, 2012'
>>> 'The end is {:%d %b %Y}. Goodbye'.format(d)
'The end is 21 Dec 2012. Goodbye'
>>>
```

Postoje neke standardne konvencije za formatiranje ugrađenih tipova. Formalnu specifikaciju naći ćete u dokumentaciji za modul `string` (*http://docs.python.org/3/library/string.html*).

8.3. Uspostavljanje podrške objekata za protokol upravljanja kontekstom

Problem

Želite da vaši objekti podržavaju protokol upravljanja kontekstom (naredba `with`).

Rešenje

Da bi objekat bio kompatibilan s naredbom `with`, morate implementirati metode `__enter__()` i `__exit__()`. Na primer, razmotrimo narednu klasu koja omogućava mrežnu vezu:

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.sock = None

    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
```

Ključna odlika ove klase jeste to što predstavlja mrežnu vezu, ali ne radi inicijalno ništa sama (recimo, ne uspostavlja vezu). Umesto toga, veza se uspostavlja i zatvara pomoću naredbe `with` (u suštini, po zahtevu). Na primer:

```
from functools import partial

conn = LazyConnection(('www.python.org', 80))
# Veza je zatvorena
with conn as s:
    # conn.__enter__() se izvršava: veza otvorena
    s.send(b'GET /index.html HTTP/1.0rn')
    s.send(b'Host: www.python.orgrn')
    s.send(b'rn')
    resp = b''.join(iter(partial(s.recv, 8192), b''))
    # conn.__exit__() se izvršava: veza zatvorena
```

Objašnjenje

Kada pišete upravljač kontekstom, imajte na umu da pišete kôd koji treba da okruži blok naredaba definisanih putem naredbe `with`. Kada se u izvršavanju dođe do naredbe `with`,

poziva se metoda `__enter__()`. Povratna vrednost metode `__enter__()` (ako je ima) smešta se u promenljivu na koju ukazuje kvalifikator `as`. Posle toga, izvršavaju se naredbe u telu naredbe `with`. Na kraju se poziva metoda `__exit__()` da počisti.

Ovakav tok izvršavanja važi bez obzira na to šta se dešava u telu naredbe `with`, pa i ako dođe do izuzetaka. Zapravo, tri argumenta metode `__exit__()` sadrže tip izuzetka, vrednost i izveštaj o praćenju steka (engl. *traceback*) za izuzetke na čekanju (ako ih ima). Metoda `__exit__()` može da odabere da li da iskoristi informacije o steku na neki način ili da ih zanemari tako što neće ništa uraditi sem što će vratiti `None` kao rezultat. Ako metoda `__exit__()` vrati `True`, izuzetak se briše kao da se ništa nije desilo i program nastavlja da izvršava naredbe neposredno iza bloka `with`.

Jedno od manje očiglednih pitanja u vezi sa ovim receptom jeste da li klasa `LazyConnection` dozvoljava ugnežđenu primenu veze s većim brojem `with` naredaba. Kako je prikazano, samo jedna veza utičnice dozvoljena je u datom trenutku, a ukoliko se naredba `with` i dalje poziva dok je utičnica u upotrebi, javlja se izuzetak. Ovo ograničenje možete premostiti uz male izmene u implementaciji:

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.connections = []

    def __enter__(self):
        sock = socket(self.family, self.type)
        sock.connect(self.address)
        self.connections.append(sock)
        return sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.connections.pop().close()

# Primer primene
from functools import partial

conn = LazyConnection(('www.python.org', 80))
with conn as s1:
     ...
     with conn as s2:
          ...
          # s1 i s2 su nezavisne utičnice
```

U drugoj verziji, klasa `LazyConnection` je svojevrsna fabrika veza. Interno, za stek se koristi lista. Kad god se `__enter__()` izvršava, uspostavlja se nova veza i dodaje se steku. Metoda `__exit__()` skida poslednju vezu sa steka i zatvara je. Ovo rešenje omogućava da se napravi više veza odjednom pomoću ugnežđenih naredaba `with`, kao što je prikazano.

Upravljači kontekstom se često koriste u programima koji upravljaju resursima kao što su datoteke, mrežne veze i brave. Najvažniji aspekt je to što se takvi resursi moraju eksplicitno zatvoriti ili osloboditi da bi se program valjano izvršavao. Na primer, ako napravite bravu, morate je i otpustiti, tj. osloboditi memoriju, inače rizikujete kružnu blokadu (engl. *deadlock*). Ako implementirate metode `__enter__()`, `__exit__()` i primenite naredbu `with`, mnogo lakše ćete izbeći takve probleme, pošto će se kôd za čišćenje u metodi `__exit__()` u svakom slučaju izvršiti.

Alternativna formulacija upravljača kontekstom može se naći u modulu `contextmanager`. Videti Recept 9.22. Verzija ovog recepta bezbedna za višenitni rad dostupna je u Receptu 12.6.

8.4. Ušteda u memoriji prilikom pravljenja velikog broja instanci

Problem

Vaš program pravi mnogo (na primer, milione) instanci, i pritom koristi ogromnu količinu memorije.

Rešenje

Za klase koje primarno služe kao jednostavne strukture podataka, često je moguće ostvariti znatnu uštedu u memoriji dodavanjem deklaracije `__slots__`. Na primer:

```
class Date:
    __slots__ = ['year', 'month', 'day']
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

Kada definišete `__slots__`, Python koristi mnogo sažetiju internu prezentaciju instanci. Umesto od rečnika, instance se, u osnovi, prave od malog niza fiksne veličine, nalik n-torci ili listi. Imena atributa navedena u specifikatoru `__slots__` interno se mapiraju na određene indekse u okviru tog niza. Sporedna posledica ovakvog rešenja je to što više nije moguće dodavati nove atribute instanci – ograničeni ste samo na imena atributa navedena u specifikatoru `__slots__`.

Objašnjenje

Memorija koja se uštedi zahvaljujući slotovima zavisi od broja i tipa sačuvanih atributa. Međutim, načelno, upotrebljena memorija je porediva s memorijom za skladištenje podataka u n-torci. Da biste stekli predstavu, pomenimo da čuvanje jedne instance tipa `Date` bez slotova zahteva 428 bajta memorije na 64-bitne verzije Pythona. Kada se definišu slotovi, zauzeta memorija se smanjuje na 156 bajtova. U programu koji istovremeno upravlja velikim brojem datuma, na ovaj način bi se ostvarila značajna ušteda u ukupnoj upotrebljenoj memoriji.

Iako se slotovi možda čine kao nešto što bi uvek moglo biti korisno, oduprite se iskušenju da im prečesto pribegavate. Mnogi delovi Pythona se oslanjaju na normalnu implementaciju zasnovanu na rečniku. Pored toga, klase koje ne definišu slotove ne podržavaju određene mehanizme (na primer, višestruko nasleđivanje). U većini slučajeva, slotove bi trebalo da koristite samo za klase koje će se često koristiti u programu kao strukture podataka (to jest, ako vaš program pravi milione instanci određene klase).

Često se za specifikator `__slots__` pogrešno misli da predstavlja alatku kapsuliranja koja sprečava korisnike da dodaju nove atribute instancama. Iako to jeste sporedna posledica primene slotova, to nikada nije bila prvobitna namera. Umesto toga, zamisao je bila da `__slots__` bude alatka za optimizaciju.

8.5. Kapsuliranje imena u klasi

Problem

Želite da kapsulirate „privatne“ podatke instance klase, ali brine vas Pythonov nedostatak kontrole pristupa.

Rešenje

Umesto da se oslanjaju na elemente jezika za kapsuliranje podataka, Pythonovi programeri bi trebalo da imaju na umu određene konvencije imenovanja u vezi sa željenom upotrebom podataka i metoda. Prva konvencija je da za svako ime koje počinje jednom donjom crtom (_) uvek treba da pretpostave da označava internu implementaciju. Na primer:

```
class A:
    def __init__(self):
        self._internal = 0    # Interni atribut
        self.public = 1       # Javni atribut

    def public_method(self):
        '''
        A public method
        '''
        ...

    def _internal_method(self):
        ...
```

Python vas, zapravo, ne sprečava da pristupate internim imenima. Ipak, to se smatra nepristojnim, i moglo bi da dovede to nestabilnog koda. Valjalo bi istaći i da se vodeća donja crta koristi i za imena modula i funkcije na nivou modula. Na primer, ako naiđete na ime modula koje počinje donjom crtom (kao što je `_socket`), znajte da je reč o internoj implementaciji. Slično tome, funkcije na nivou modula kao što je `sys._getframe()` koristite vrlo oprezno.

Možda ćete se susresti i s dve početne donje crte (__) u imenima u okviru definicija klase. Na primer:

```
class B:
    def __init__(self):
        self.__private = 0
    def __private_method(self):
        ...
    def public_method(self):
        ...
        self.__private_method()
        ...
```

Zbog prefiksa u vidu duple donje crte, ime je iseckano. Konkretno, privatni atributi u prethodnoj klasi preimenuju se u `_B__private` i `_B__private_method`. Možda se sada pitate čemu takvo seckanje imena. Odgovor je u nasleđivanju – ovakvi atributi se ne mogu redefinisati putem nasleđivanja. Na primer:

```
class C(B):
    def __init__(self):
        super().__init__()
        self.__private = 1      # Ne redefiniše B.__private
    # Ne redefiniše B.__private_method()
    def __private_method(self):
        ...
```

U ovom primeru, privatna imena `__private` i `__private_method` izmenjena su u `_C__private` i `_C__private_method` – dakle, drugačija su od imena u osnovnoj klasi `B`.

Objašnjenje

Činjenica da postoje dve konvencije (s jednom, odnosno dve donje crte) za privatne atribute, navodi da se zapitate koji stil bi trebalo da primenite. U većini slučajeva, verovatno bi imena koja nisu javna trebalo da počinjete jednom donjom crtom. Međutim, ako znate da će vaš kôd da dovede do potklasa i postoje interni atributi koji bi trebalo da budu sakriveni od potklasa, primenite umesto toga duplu donju crtu.

Valjalo bi istaći i da ćete ponekad možda hteti da definišete promenljivu istog imena kao rezervisana reč. U tom slučaju, trebalo bi da stavite jednu prateću donju crtu. Na primer:

```
lambda_ = 2.0     # Prateća _ da bi se izbeglo poklapanje s rezervisanom rečju lambda
```

U ovom slučaju, početnu donju crtu nismo primenili da bismo izbegli zabunu u vezi s namerom (to jest, vodeća donja crta mogla bi se protumačiti kao način da se predupredi konflikt imena, umesto kao indikacija da je vrednost privatna). Prateća donja crta rešava taj problem.

8.6. Pravljenje upravljanih atributa

Problem

Hoćete da proširite obradu (na primer, da dodate proveru tipa ili ispravnosti) u okviru čitanja ili zadavanja atributa instance.

POGLAVLJE 11

Mrežno i veb programiranje

Ovo poglavlje pokriva razne teme vezane za korišćenje Pythona u mrežnim i distribuiranim aplikacijama. S jedne strane, govorimo o korišćenju Pythona kao klijenta za pristup raznim servisima, a s druge, o implementiranju mrežnih servisa u vidu servera pomoću Pythona. Predstavljamo i uobičajene tehnike za pisanje koda za rad ili komuniciranje sa interpreterima.

11.1. Interakcija sa HTTP servisima iz pozicije klijenta

Problem

Treba da pristupate raznim servisima putem HTTP protokola kao klijent. Na primer, da preuzimate podatke ili da radite sa API interfejsom REST.

Rešenje

U jednostavnim slučajevima, obično je najlakše primeniti modul `urllib.request`. Na primer, da biste poslali jednostavan HTTP zahtev `GET` udaljenom servisu, postupićete na sledeći način:

```
from urllib import request, parse

# Pristupa se osnovnoj URL adresi
url = 'http://httpbin.org/get'

# Rečnik parametara upita (ako ih ima)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Kodira znakovni niz upita
querystring = parse.urlencode(parms)

# Pravi GET zahtev i čita odgovor
u = request.urlopen(url+'?' + querystring)
resp = u.read()
```

Ukoliko je potrebno da pošaljete parametre upita u telu zahteva metodom `POST`, kodiraćete ih i poslati kao opcioni argument funkcije `urlopen()` na sledeći način:

```
from urllib import request, parse

# Pristupa se osnovnoj URL adresi
url = 'http://httpbin.org/post'

# Rečnik parametara upita (ako ih ima)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Kodira znakovni niz upita
querystring = parse.urlencode(parms)

# Pravi POST zahtev i čita odgovor
u = request.urlopen(url, querystring.encode('ascii'))
resp = u.read()
```

Ako u zahtev treba da uvrstite neka namenska HTTP zaglavlja, kao što je promena polja korisničkog agenta, napravite rečnik s njihovom vrednošću, potom i instancu tipa `Request` i prosledite je funkciji `urlopen()` na sledeći način:

```
from urllib import request, parse
...

# Dodatna zaglavlja
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

req = request.Request(url, querystring.encode('ascii'), headers=headers)

# Pravi zahtev i čita odgovor
u = request.urlopen(req)
resp = u.read()
```

Ukoliko je interakcija komplikovanija, verovatno bi trebalo da razmislite o biblioteci `requests` (*http://pypi.python.org/pypi/requests*). Na primer, evo koda ekvivalentnog prethodnim operacijama, ali s bibliotekom `requests`:

```
import requests

# Pristupa se osnovnoj URL adresi
url = 'http://httpbin.org/post'

# Rečnik parametara upita (ako ih ima)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}
```

```
# Dodatna zaglavlja
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

resp = requests.post(url, data=parms, headers=headers)

# Dekodirani tekst koji zahtev vraća
text = resp.text
```

Važna karakteristika biblioteke `requests` jeste način na koji vraća sadržaj dobijenog odgovora na zahtev. Kao što je prikazano, atribut `resp.text` sadrži dekodirani Unicode tekst zahteva. Međutim, ako pristupite atributu `resp.content`, umesto toga dobijate binarni sadržaj. S druge strane, vrednost u atributu `resp.json` jeste sadržaj odgovora u formatu JSON.

Evo kako biste upotrebili `requests` da napravite `HEAD` zahtev i izdvojite nekoliko polja podataka zaglavlja iz zahteva:

```
import requests

resp = requests.head('http://www.python.org/index.html')

status = resp.status_code
last_modified = resp.headers['last-modified']
content_type = resp.headers['content-type']
content_length = resp.headers['content-length']
```

U narednom primeru, biblioteka `requests` se koristi za prijavljivanje na portal Python Package Index putem osnovne provere identiteta:

```
import requests

resp = requests.get('http://pypi.python.org/pypi?:action=login',
                    auth=('user','password'))
```

Evo primera u kome se pomoću biblioteke `requests` prosleđuju HTTP kolačići od jednog do drugog zahteva:

```
import requests

# Prvi zahtev
resp1 = requests.get(url)
...
# Drugi zahtevi dobijeni na prve zahteve
resp2 = requests.get(url, cookies=resp1.cookies)
```

Na kraju, evo primera korišćenja biblioteke `requests` za otpremanje sadržaja:

```
import requests
url = 'http://httpbin.org/post'
files = { 'file': ('data.csv', open('data.csv', 'rb')) }

r = requests.post(url, files=files)
```

Objašnjenje

Ugrađeni modul `urllib` obično je dovoljan za jednostavan kôd HTTP klijenta. Međutim, za bilo šta van jednostavnih `GET` ili `POST` zahteva, ne možete računati na njegovu funkcionalnost. Tu na scenu stupaju namenski moduli kao što je `requests`.

Recimo, ako odlučite da se dosledno držite standardne biblioteke, izbegavajući biblioteke kakva je `requests`, možda ćete morati da implementirate kôd pomoću modula niskog nivoa `http.client`. Na primer, evo kako se izvršava `HEAD` zahtev:

```
from http.client import HTTPConnection
from urllib import parse

c = HTTPConnection('www.python.org', 80)
c.request('HEAD', '/index.html')
resp = c.getresponse()

print('Status', resp.status)
for name, value in resp.getheaders():
    print(name, value)
```

Isto tako, ako treba da pišete kôd s namenskim klasama, proverom identiteta, kolačićima i drugim detaljima, modul `urllib` je nepodesan i preopširan. Kao primer, navešćemo jednostavan segment koda za proveru identiteta prilikom upisa na portal Python Package Index:

```
import urllib.request

auth = urllib.request.HTTPBasicAuthHandler()
auth.add_password('pypi','http://pypi.python.org','username','password')
opener = urllib.request.build_opener(auth)

r = urllib.request.Request('http://pypi.python.org/pypi?:action=login')
u = opener.open(r)
resp = u.read()

# Odavde možete da pristupite stranicama koristeći opener
...
```

Iskreno govoreći, sve ovo je mnogo lakše uraditi pomoću biblioteke `requests`.

Testiranje koda HTTP klijenta u razvojnoj fazi često može biti obeshrabrujuće zbog raznih nezgodnih detalja o kojima morate da vodite računa (na primer, o kolačićima, proveri identiteta, zaglavljima, kodiranju itd.). U tome bi vam mogao pomoći servis httpbin (*http://httpbin.org*). Ova veb lokacija prima zahteve, a potom vam vraća informacije u obliku JSON odgovora. Evo interaktivnog primera:

```
>>> import requests
>>> r = requests.get('http://httpbin.org/get?name=Dave&n=37',
...     headers = { 'User-agent': 'goaway/1.0' })
>>> resp = r.json
>>> resp['headers']
{'User-Agent': 'goaway/1.0', 'Content-Length': '', 'Content-Type': '',
'Accept-Encoding': 'gzip, deflate, compress', 'Connection':
```

```
'keep-alive', 'Host': 'httpbin.org', 'Accept': '*/*'}
>>> resp['args']
{'name': 'Dave', 'n': '37'}
>>>
```

Često je bolje raditi s veb lokacijom *httpbin.org* nego eksperimentisati s pravom veb prezentacijom, naročito ako postoji rizik da vam ukinu nalog posle tri neuspešna pokušaja prijavljivanja (tj., ne pokušavajte da naučite kako se piše HTTP klijent za proveru identiteta tako što ćete se prijavljivati na portal vaše banke).

Iako ovde ne razgovaramo o tome, modul `requests` ima podršku za brojne naprednije protokole HTTP klijenata, kao što je OAuth. Dokumentacija za modul `requests` (*http://docs.python-requests.org*) izuzetna je (i, iskreno govoreći, bolja od bilo čega što bi moglo da stane u ovako ograničen prostor). Tamo ćete naći podrobnije informacije.

11.2. Pravljenje TCP servera

Problem

Hoćete da implementirate server koji komunicira s klijentima putem TCP Internet protokola.

Rešenje

Biblioteka `socketserver` omogućava da lako napravite TCP server. Na primer, evo jednostavnog eho-servera (engl. *echo server*):

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        while True:
            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

U ovom kodu definisali smo specijalnu upravljačku klasu koja implementira metodu `handle()` za servisiranje klijentskih veza. Atribut `request` predstavlja odgovarajući klijentski priključak, dok `client_address` sadrži adresu klijenta. Da bismo testirali server, pokrenućemo ga, a potom otvoriti poseban Pythonov proces koji se povezuje s njim:

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect(('localhost', 20000))
>>> s.send(b'Hello')
5
```

```
>>> s.recv(8192)
b'Hello'
>>>
```

Često je lakše definisati malo drugačiju vrstu upravljačke klase. U narednom primeru koristimo osnovnu klasu `StreamRequestHandler` da priključku pridružimo interfejs nalik datoteci:

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # self.rfile je objekat nalik datoteci za čitanje
        for line in self.rfile:
            # self.wfile je objekat nalik datoteci za upisivanje
            self.wfile.write(line)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

Objašnjenje

Pomoću modula `socketserver`, relativno je lako napraviti jednostavne TCP servere. Međutim, trebalo bi da znate da su serveri podrazumevano jednonitni i da u jednom trenutku mogu da opslužuju samo jednog klijenta. Ako hoćete da radite s više klijenata, umesto toga napravite instancu tipa `ForkingTCPServer` ili `ThreadingTCPServer`. Na primer:

```
from socketserver import ThreadingTCPServer
...
if __name__ == '__main__':
    serv = ThreadingTCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

Ovakvi serveri, koji rade s više klijenata ili procesa, prave nov proces ili nit za svaku vezu s klijentom. Ne postoji ograničenje u broju dozvoljenih klijenata, tako da bi zlonamerni haker mogao pokrenuti veliki broj veza u isto vreme, s namerom da onesposobi vaš server.

Ukoliko se brinete da bi se to moglo desiti, možete da napravite unapred alocirano skladište radnih niti ili procesa. U tu svrhu, napravili biste instancu normalnog servera bez niti, ali biste potom pokrenuli metodu `serve_forever()` nad skladištem niti. Na primer:

```
...
if __name__ == '__main__':
    from threading import Thread
    NWORKERS = 16
    serv = TCPServer(('', 20000), EchoHandler)
    for n in range(NWORKERS):
        t = Thread(target=serv.serve_forever)
        t.daemon = True
        t.start()
    serv.serve_forever()
```

U normalnim okolnostima, `TCPServer` povezuje i aktivira utičnicu po instanciranju. Međutim, ponekad ćete hteti da podesite datu utičnicu. Da biste to uradili, prosledili argument `bind_and_activate=False` na sledeći način:

```
if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler, bind_and_activate=False)
    # Podešava razne opcije priključka
    serv.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    # Povezuje i aktivira
    serv.server_bind()
    serv.server_activate()
    serv.serve_forever()
```

Prikazana opcija utičnice zapravo je uobičajena postavka koja omogućava serveru da se ponovo poveže s prethodno korišćenim brojem priključaka (engl. *port*). U toj meri je uobičajena, da predstavlja promenljivu klase koja se može zadati za klasu `TCPServer`. Podesićete je pre instanciranja servera, kao što je prikazano u ovom primeru:

```
...
if __name__ == '__main__':
    TCPServer.allow_reuse_address = True
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

U ovom rešenju prikazane su dve upravljačke osnovne klase (`BaseRequestHandler` i `StreamRequestHandler`). Klasa `StreamRequestHandler` malo je prilagodljivija i podržava neke opcije koje se mogu aktivirati putem specifikacije dodatnih promenljivih klase. Na primer:

```
import socket

class EchoHandler(StreamRequestHandler):
    # Opciona podešavanja (podrazumevana u ovom primeru)
    timeout = 5                     # Tajm-aut za sve operacije utičnice
    rbufsize = -1                   # Čita veličinu bafera
    wbufsize = 0                    # Upisuje veličinu bafera
    disable_nagle_algorithm = False  # Zadaje opciju TCP_NODELAY utičnice
    def handle(self):
        print('Got connection from', self.client_address)
        try:
            for line in self.rfile:
                # self.wfile je objekat nalik datoteci za pisanje
                self.wfile.write(line)
        except socket.timeout:
            print('Timed out!')
```

Na kraju, valja primetiti da je većina Pythonovih mrežnih modula visokog nivoa (na primer, HTTP, XML-RPC itd.) napravljena povrh funkcionalnosti `socketserver`. Ipak, servere je jednostavno i direktno implementirati pomoću biblioteke `socket`.

Evo jednostavnog primera direktnog programiranja servera pomoću ove biblioteke:

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_handler(address, client_sock):
    print('Got connection from {}'.format(address))
    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)
    client_sock.close()

def echo_server(address, backlog=5):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(backlog)
    while True:
        client_sock, client_addr = sock.accept()
        echo_handler(client_addr, client_sock)

if __name__ == '__main__':
    echo_server(('', 20000))
```

11.3. Pravljenje UDP servera

Problem

Hoćete da implementirate server koji komunicira s klijentima putem UDP Internet protokola.

Rešenje

Kao i TCP servere, UDP servere ćete lako napraviti pomoću biblioteke `socketserver`. Evo primera jednostavnog servera za vreme:

```
from socketserver import BaseRequestHandler, UDPServer
import time

class TimeHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # Dobavlja poruku i utičnicu klijenta
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    serv = UDPServer(('', 20000), TimeHandler)
    serv.serve_forever()
```

I u ovom slučaju, definišete posebnu upravljačku klasu koja implementira metodu `handle()` za servisiranje klijentskih veza. Atribut `request` je n-torka koja sadrži dolazeći datagram i prateći objekat utičnice za server. `client_address` sadrži adresu klijenta.

Ako želite da testirate server, pokrenite ga, a potom otvorite zaseban Pythonov proces koji mu šalje poruke:

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 20000))
0
>>> s.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 20000))
>>>
```

Objašnjenje

Tipičan UDP server prima datagram (poruku) zajedno sa adresom klijenta. Ako server hoće da odgovori, vratiće datagram klijentu. Datagrame bi trebalo da šaljete metodama `sendto()` i `recvfrom()` utičnice. Iako bi mogle da se upotrebe i tradicionalne metode `send()` i `recv()`, prethodne dve metode su uobičajene kada je reč o UDP komunikaciji.

S obzirom da u osnovi nema veze, UDP server se često mnogo lakše piše nego TCP server. Međutim, UDP server je i nepouzdan (na primer, ne uspostavlja se „veza“ i poruke se mogu zagubiti). Zato je na vama da odredite šta ćete sa izgubljenim porukama. Ta tema prevazilazi opseg ove knjige, ali se pouzdanost obično postiže brojevima sekvenci, ponovnim pokušajima, tajm-autom i drugim mehanizmima, ukoliko je to važno za vašu aplikaciju. UDP se često koristi u slučajevima za koje ne postoje strogi zahtevi za pouzdanu isporuku poruka. Na primer, u aplikacijama koje se izvršavaju u realnom vremenu, poput reprodukovanja multimedijskih sadržaja u realnom vremenu i igrica, kada nema mogućnosti da se vratite unazad i povratite izgubljeni paket (program ga naprosto preskače i nastavlja napred).

Klasa `UDPServer` je jednonitna, što znači da se u jednom trenutku može obrađivati samo jedan zahtev. U praksi, to je manje važno za UDP veze nego za TCP veze. Ipak, ako vas zanimaju istovremene operacije, umesto toga instanciraćete objekat tipa `ForkingUDPServer` ili `ThreadingUDPServer`:

```
from socketserver import ThreadingUDPServer
...
if __name__ == '__main__':
    serv = ThreadingUDPServer(('',20000), TimeHandler)
    serv.serve_forever()
```

Direktno implementiranje UDP servera pomoću utičnica takođe nije teško uraditi. Evo primera:

```
from socket import socket, AF_INET, SOCK_DGRAM
import time

def time_server(address):
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.bind(address)
    while True:
        msg, addr = sock.recvfrom(8192)
        print('Got message from', addr)
```