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}
}
```