Univerza v Ljubljani

Fakulteta za računalništvo in informatiko

Janez Demšar

Python za programerje

Ljubljana, 2009

O jeziku in knjigi

Python je skriptni jezik. Programi v njem so lepo berljivi – to je bil eden ciljev pri sestavljanju jezika – zato je primeren za začetnika, ki se mu tako poleg tega, da se mora učiti razmišljati kot programer, ni potrebno ukvarjati še z odvečno sintaktično navlako, za njegovega ubogega učitelja, ki mora slediti učenčevim izkrivljenim mislim in ugibati njegove zgrešene ideje, in za profesionalca, ki mora brati tujo kodo ali razumeti svojo po tem, ko je nekaj let ni pogledal.

Zaradi berljivosti ne trpi hitrost programiranja. Nasprotno, programiranje v Pythonu je zelo hitro, saj so visokonivojske podatkovne strukture tesno vdelane v jezik, svoje pa dodajo tudi elementi funkcijskega programiranja, ki jih je pokradel Haskllu, in že napisani moduli, ki programiranje v Pythonu pogosto spremenijo v komaj kaj več kot lepljenje tuje kode.

Jezik je predmetno usmerjen, brez neobjektnih "primitivov" in podobnih kompromisov. Celo modul, funkcija in tip so objekti. Obenem predmetne usmerjenosti ne vsiljuje: z razredi se zavestno ukvarjamo le, ko to želimo, sicer pa se le tiho sukajo nekje v ozadju.

Slabost Pythona je hitrost izvajanja. Ker se (neopazno) prevaja v kodo za navidezni stroj, je hitrejši od skriptnih jezikov, ki tega ne počno, obenem pa se ne more kosati z Javo, katere navidezni stroj je običajno (a ne vedno) hitrejši, kaj šele s Cjem, ki teče na čisto pravem stroju. Vendar to nikakor ni resna pomanjkljivost, saj moremo vse časovno ali prostorsko zahtevnejše delo sprogramirati v Cju, s katerim se Python lepo povezuje – če tega ni pred nami naredil že kdo drug. Možnosti za slednje so velike, na spletu najdemo knjižnice za vse od linearne algebre do grafike in zvoka. Zajemati sliko s kamere in jo nekoliko obdelano predvajati v oknu zahteva le nekaj vrstic lastne kode, s katero povežemo dva, tri module, ki jih brezplačno dobimo na spletu.

Uporabljati Python za tisto, kar bi radi hitro sprogramirali, in C le za tisti del programa, ki bi bil v Pythonu prepočasen, je bolj smiselno od uporabe jezika, ki poskuša zadostiti obojemu in zato vsakemu zadosti le napol. Da je ideja dobra, kaže širok spekter uporabe Pythona: v njem so pisane mnoge sistemske skripte za

Linux, v njem je vedno več spletnih aplikacij (med njimi je najbrž najopaznejši YouTube), Python je eden treh "uradnih jezikov" Googlea (poleg JavaScripta in Cja), te, ki jih morda le skrbi hitrost izvajanja, pa mora potolažiti, da so bili v njem izdelani vizualni učinki za Vojno zvezd.

Knjiga je namenjena bralcu, ki že zna programirati. Kdor ne ve, kakšna je razlika med spremenljivko in zanko, naj poseže po čem drugem.

Nekdo, ki je vešč vsaj enega imperativnega jezika (se pravi bolj ali manj česarkoli, kar je danes v širši rabi), se lahko naslednjega navidez nauči zelo hitro. Izvedeti mora le, kako se napiše `for` in kako `if`, pa bo jezik približno znal. S tem bomo opravili v prvih poglavjih. Da bi bil ta, dolgočasnejši del za nami čim hitreje, bomo mestoma malo preskakovali in v kakem primeru uporabili zanko `for` ali seznam, čeprav ju bomo zares predstavili šele nekoliko kasneje, ali `import`, ne da bi prej povedali, da moduli v Pythonu sploh obstajajo. Od bralca pričakujemo, da bo v osnovi razumel, za kaj gre, na detajle pa potrpežljivo počakal.

Kako dolga je pot od tam, ko jezik *navidez* znamo, do razumevanja v najglobjo globino, je odvisno od globine jezika. Ob "tehničnih" jezikih (zbirnik, C...) o kaki globini ne moremo govoriti. Tudi moderni jeziki povezani s spletom (naj mi ljubitelji PHPja odpustijo) se izogibajo razumevanja potrebni filozofiji – morda namerno, da bi bili tako dostopnejši "širokim (pol)programerskim masam".

Python je drugačen. Čeprav se ga začetnik hitro nauči in programer hitro priuči, se jima za vsako lastnostjo, ki jo približno poznata, postopno razkriva več in več ozadja. Za zanko `for` se skrivajo iteratorji; moduli, razredi ter celo globalne in lokalne spremenljivke temeljijo na slovarjih; tip je shranjen kot objekt tipa tip (ki je tudi sam objekt tipa tip (ki je tudi sam objekt tipa tip (...))) Če bi to zamolčali, bralec ne bi bil le slabši programer, temveč bi ga tako prikrajšali za najprivlačnejši del zgodbe. Obenem bomo skrbeli, da bo vse povedano služilo praksi in, predvsem, da ne bo postalo prezapleteno, temveč bo zabavno. Vsi dobri programski jeziki so zabavni, torej morajo biti knjige tudi.

V knjigi zato tudi ne bomo preveč formalni. Ob marsikateri nerodni besedi ali poenostavljeni razlagi se utegnejo teoretiki križati (če so pobožni in preklinjati, če niso). Pa naj.

Ob izboru tem bomo imeli v mislih bralca, ki mu je domače programiranje v C++ ali podobnem jeziku. Tako se ne bomo pretirano dolgo zadrževali ob rečeh, ki so mu znane od ondod. Ob razredih, denimo, bomo govorili o Pythonovih posebnostih in ne o tem, kaj je metoda in čemu služi. O funkcijskem programiranju bomo napisali veliko več, kot bi si to morda zaslužilo, vendar predvsem zato, ker je takšen način razmišljanja bralcu verjetno tuj, pri programiranju v Pythonu, pa mu utegne priti zelo prav.

Največja ovira pri uporabi novega jezika je (s)poznavanje njegovih knjižnic. Kdor obvlada C++, se resda lahko v pol ure nauči še Javo, vendar bo v njenem svetu gol in bos, dasiravno se mu obutev in odelo brezplačno ponujata na vsakem vogalu svetovnega spleta. Tudi Python je v tem pogledu med najbolj založenimi jeziki, a že garderobe, ki jo dobimo v osnovni namestitvi, je toliko, da jo le malokdo v celoti pozna. Tu si bomo površno ogledali nekaj najpomembnejših, za ostalimi bo bralec brskal po svojih tekočih potrebah.

Knjiga temelji na Pythonu 2.5. V dvajset letih, kar jezik obstaja, so bile nove različice Pythona kvečjemu minimalno nezdružljive s predhodnimi, zato bo praktično vse, o čemer boste brali, veljalo tudi za nekoliko starejše različice (vsaj od 2.2 ali 2.3) pa tudi za novejše iz serije 2.X, ki bo živa še dolgo. Že več let pa je vzporedno v pripravi Python 3.0, ki bo pometel vse, kar se je v jeziku izkazalo za zgrešeno ali pa je bilo nadomeščeno z boljšim, vendar nikoli odpravljeno zaradi združljivosti. Veliko sprememb bo začetniku neopaznih, resnejši programer pa jih bo opazil, a tudi razumel in pozdravil. V knjigi jih bomo omenili, kadar se nam bo zdelo potrebno, vedno pa ne.

Posebno pomemben del knjige so naloge. Bralec naj jih ne le rešuje, temveč predvsem bere objavljene rešitve, tudi kadar je nalogo sam znal pravilno rešiti. Pogosta težava začetnikov v Pythonu, ki obvladajo druge jezike, je, da so njihovi programi videti kot dobeseden prevod iz Cja. Namen komentiranih rešitev je poudariti razlike v slogu programiranja. Če v Pythonu programiramo v Cju, so programi počasni, programiranje pa (vsaj) tako naporno kot v Cju.

In, še bolj pomembno: knjige o programiranju je potrebno brati z računalnikom pred sabo in vse, kar preberemo, sproti preskušati. Komur se ne da, pa naj namesto Demšarja raje vzame Cervantesa ali Vonneguta. Koristi prav tako ne bo, bo pa vsaj zabavno.

Hvala

recenzentoma prof. Blažu Zupanu in doc. Tomažu Dobravcu za osupljivo natančno branje knjige.

In Ireni, ki potrpežljivo prenaša, da je njen mož stalno na službeni poti v delovni sobi, za računalnikom. In Matevžu, Martinu in Nuši, ki so me (premalokrat) zvlekli izpred njega z velikimi kamioni bagri traktorji, s kockami iz darila in, no ja, neusmiljenim predirljivim vreščanjem.

Vsebina

Prvi koraki v Python

Bralec knjige si bo očitno moral priskrbeti Python, če ne želi brati v prazno. V Linuxu se ta navadno namesti kar samodejno, saj večje distribucije brez njega ne delujejo. Ostali si bodo našli primerno okolje na spletu: večina je brezplačna.

Ko bo to opravljeno, bomo skupaj napisali preprosto funkcijo in se poigrali z njo, da ob njej spoznamo osnovne poteze jezika.

Tolmač in razvojna okolja

Python deluje na veliko različnih sistemih. Skoraj povsod ga dobimo v obliki ukazne lupine, python (ali python.exe). Ta za praktično rabo ni posebej prikladna, pač pa jo uporabljamo za izvajanje že napisanih programov. Skripto shranjeno v datoteki *mojprogram.py* najlažje poženemo z `python mojprogram.py`.

Nekoliko udobnejša alternativa je ipython. Ta ima lupino, podobno tisti v programu Mathematica, ki oštevilčuje vhodne ukaze in rezultate, tako da jih lahko uporabljamo tudi kasneje. Poleg tega ponuja pomoč pri vpisovanju, kot je dopolnjevanje s pritiskom na tabulator in podobno. Za začetnika pa je tudi to še vedno prešpartansko.

Kdor je vajen okolja Eclipse, bo kasneje lahko ostal kar na njem, le pydev mu bo dodal, pa bo dobil imenitno razvojno okolje za večje projekte. Za sledenje tej knjigi pa je Eclipse prevelik kanon.

Za uvajanje v Python potrebujemo nekaj vmes. V MS Windows je to PythonWin ali PyScripter. Osebno mi je bližji prvi, študenti pa so odkrili in mnogi pograbili drugega. Je sodobnejši, vizualno privlačnejši in ima bistveno več nastavitev kot PythonWin, a nekomu vajenemu PythonWina, manjka ravno tista, ključna (katerakoli že to je). Nekatera pakiranja Pythona vsebujejo okolje Idle, ki je napisano v Tcl/Tkju in zato deluje na več operacijskih sistemih, vendar je v primerjavi s PythonWinom in PyScripterjem videti dokaj okorno. Za Linux in Mac Os X bo prikladnejše okolje Eric, pa tudi za MS Windows obstaja in mu pravzaprav

ne manjka nič. Spet tretji prisegajo na Komodo, ki prav tako teče na vseh treh operacijskih sistemih.

Razvojnih okolij za Python ne manjka. Bralec si bo za začetek najbrž izbral nekaj od gornjega, kasneje pa naj se le še razgleda za čim po lastnem okusu.

Prvi program

Avtor knjige stoji trdno na Nasredinovskem stališču, da programski jezik, v katerem izpis "Pozdravljen svet!" zahteva deset vrstic kode, med katerimi nekatere že zahtevajo krajši komentar, druge pa klepljejo skupaj razrede, ni primeren za pisanje programa "Pozdravljen svet!". Pravzaprav to velja že za jezike, ki zahtevajo več kot eno vrstico kode. V jezikih, v katerih je izpis takšnega sporočila dolg le eno vrstico, pa tega programa očitno nima smisla kazati. S knjigami, ki na začetku kot kak norček na avtobusu pozdravljajo ves svet okrog sebe, je torej v vsakem primeru nekaj narobe.

Našo bomo zato začeli s približno najkrajšim smiselnim programom v Pythonu: s funkcijo, ki vrne *n*-ti člen Fibonaccijevega zaporedja.

```
# Izracun n-tega Fibonaccijega stevila
def fibonacci(n):
    a = b = 1
    for i in range(n):
        a, b = b, a+b
    return a
```

Zdaj bo bralec odprl izbrano razvojno okolje – na MS Windows verjetno PythonWin, v Linuxu morda Erica. Med različnimi podokni, na katere je razdeljeno okno, ki se odpre, je navadno urejevalnik, v katerega lahko pišemo program, ga shranimo in poženemo, ter ukazna vrstica, v kateri sicer lahko prav tako programiramo, vendar jo navadno uporabljamo bolj ali manj le za poskusne klice funkcij. Poleg tega se vanjo v večini okolij izpisujejo rezultati pognanih programov. Okolja se v podrobnostih razlikujejo, vendar odkrivanje tega prepuščamo bralcu (že ob tem odstavku je najbrž kdo zazehal). Le onim s PythonWinom še namignimo, da bo ta postal prijaznejši, če v *View* / *Options* / *General Options* obkljukajo *Dockable windows* in ga ponovno zaženejo.

Gornjo funkcijo odtipkamo v urejevalnik in datoteko shranimo na disk. V večini okolij zadošča, da jo shranimo le enkrat, toliko, da ji damo ime. Kasneje se bo pred vsakim zaganjanjem shranila samodejno. Skripto nato zaženemo; v PythonWinu se lahko navadite bližnjice: *Ctrl-R*. V PythonWinu bodite posebej pozorni še na sintaktične napake, saj jih javlja zelo diskretno: ob poskusu zagona programa se tiho pritoži v statusni vrstici in postavi kurzor na mesto napake.

Če je med nami kdo iz Šparte, bo delal kar s programom python ali, če ni čisto čisto pravi Špartanec, ipython. Tadva nima vdelanega urejevalnika, temveč le ukazno vrstico. Program bo zato odtipkal v ločenem urejevalniku (vi, emacs, Notepad...), nato pognal python in odtipkal `execfile("fibonacci.py")` (ali kakorkoli je datoteki že ime).[1]

Kakorkoli že izvedemo gornji program: program vsebuje definicijo funkcije, zato je rezultat njegovega izvajanja definirana funkcija. Zdaj pa jo pokličimo. To bomo storili iz ukazne vrstice okolja (Shell v Ericu, Interactive Windows oz. spodnji del okna v PythonWinu, sicer pa jo prepoznamo tudi po >>>).

```
>>> print fibonacci(6)
13
>>> print fibonacci(50)
20365011074
```

Ukazne vrstice so navadno narejene tako, da smemo `print` izpustiti in izpis bo v večini primerov enak.

```
>>> fibonacci(6)
13
```

Seveda bi lahko gornje klice dodali v sam program, za definicijo funkcije `fibonacci` (v tem primeru ukaza `print` ne smemo izpustiti, sicer se bo funkcija poklicala, izpisalo pa se ne bo nič). Če storimo tako, program, ko ga izvedemo, ne bo le definiral funkcije, temveč tudi izpisal njen rezultat.

Mimogrede, s klici iz ukazne vrstice, preskušamo tudi tuje funkcije. Če se ne moremo spomniti vrstnega reda argumentov funkcije, ki smo jo nazadnje

1 Program sam bi lahko pognal tudi tako, da bi v sistemski ukazni lupini odtipkal `python fibonacci.py`, vendar bi tako program le definiral funkcijo in se končal, poklical pa je ne bi.

uporabili predlansko pomlad, nam ni potrebno brskati po dokumentaciji, temveč jih z ugibanjem dobimo v ukazni vrstici. Če ne vemo, ali funkcija `math.log` računa naravne ali desetiške (ali binarne?) logaritme, tudi to izvemo kar v ukazni vrstici.

```
>>> import math
>>> math.log(2.7)
0.99325177301028345
```

Zdaj pa funkcijo `fibonacci` nekoliko spremenimo. Dodajmo možnost nastavljanja prvih dveh členov, privzeti vrednosti pa naj bosta `1, 1`.

```
def fibonacci(n, a=1, b=1):
    for i in range(n):
        a, b = b, a+b
    return a
```

Ko funkcijo spremenimo, moramo program izvesti še enkrat, da nova definicija funkcije "povozi" staro. In že lahko preverimo, kako deluje.

```
>>> fibonacci(6)
13
>>> fibonacci(6, 4, 10)
100
```

Zdaj pa je napočil čas, da se zmotimo.

```
>>> fibonacci(6, 4, "a")
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "c:\d\fibo.py", line 3, in fibonacci
    a, b = b, a+b
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Kako razhroščevati, je odvisno od okolja, ki ga uporabljamo. Špartanci razhroščujejo špartansko: po programu nasejejo goro diagnostičnih izpisov. Večina grafičnih okolij za Python omogoča nastavljanje prekinitvenih točk (*breakpoints*) in opazovanje spremenljivk na mestu, kjer se je program ustavil. Bralcu nam tega gotovo ni potrebno opisovati. Poleg tega lahko okolja, podobno kot okolja za druge jezike, aktivirajo razhroščevalnik v trenutku, ko pride do napake in pomagajo programerju raziskati, kaj se dogaja. V Pythonu nekatera okolja (PythonWin) temu pravijo *Post-mortem debugging*. Le-ta deluje le, če funkcijo pokličemo iz same skripte, ne iz ukazne vrstice.

Napačni klic torej dodajmo na konec skripte, za definicijo funkcije.

```
def fibonacci(n, a=1, b=1):
    for i in range(n):
        a, b = b, a+b
    return a

print fibonacci(6, 4, "a")
```

Zdaj spet poženimo program, vendar v PythonWinu v dialogu, ki se pojavi pred zagonom, pod *Debugging* izberimo *Post Mortem of Unhandled Exceptions*. Ko program poženemo, dobimo enako sporočilo o napaki kot prej, vendar se program ustavi na mestu, kjer se je zgodila napaka. Kot v drugih jezikih nam je tudi zdaj navadno na voljo izpis sklada z lokalnimi in globalnimi spremenljivkami, opazujemo lahko njihove vrednosti, jih spreminjamo... Ker je Python tolmačen jezik, pa nam je na voljo še nekaj več: ukazna vrstica. V tej lahko izvemo ne le vrednosti spremenljivk, temveč celo odtipkamo del kode, pokličemo kako funkcijo in podobno. Vse, kar v ukazni vrstici počnemo v tem trenutku, se nanaša na trenutno stanje programa: koda, ki jo odtipkamo, se vede, kot da bi bila napisana na mestu, kjer se je program ustavil.

Kaj je šlo narobe, tako lahko hitro odkrijemo.

```
[Dbg]>>> a
5
[Dbg]>>> b
'a'
[Dbg]>>> a+b
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Okolja za Python sicer ne dosegajo obsega in udobja velikih komercialnih razvojnih okolij, kakršno je Visual Studio, zaradi same narave jezika pa sta razvoj in vzdrževanje programov v Pythonu kljub temu bistveno hitrejša in preprostejša, kot smo jih vajeni v klasičnih prevajanih jezikih.

Osnovne poteze jezika

Knjiga je namenjena bralcu, ki že zna programirati, kar navadno pomeni, da pozna C, C++, C#, Javo ali Pascal. Takšen bralec bo moral za začetek več pozabljati kot pomniti. Da ugotovimo kaj, postavimo eno ob drugo funkciji, ki Fibonaccijeva števila računata v C++ (in njegovih modnih izpeljankah) in v Pythonu.

```
int fibonacci(int n)
{
    int a = 1, b = 1;
    for(int i = n; i--; ) {
        int t = b;
        b = a+b;
        a = t;
    }
    return a;
}
```

```
def fibonacci(n):

    a = b = 1
    for i in range(n):
        a, b = b, a+b



    return a
```

Kaj naj bralec torej pozabi? Za začetek, očitno, zavite oklepaje oziroma `begin-end`, če prihaja iz Pascala. Če je bil vajen v svojem materinem jeziku kodo lepo zamikati, naj to počne še naprej in Python bo iz zamikov prepoznal, kaj je gnezdeno znotraj česa. Če imamo zamike, čemu pisati še oklepaje?! Če tega doslej ni počel, ga bo Python (edino pravilno!) prisilil, da s tem začne.

Glavi definicije funkcije sledi dvopičje in v naslednji vrstici smo povečali zamik. Podobno je z zanko `for`: glava, dvopičje, zamik. In vedno je tako: dvopičje, zamik. Zamik je lahko poljuben. Tule smo kodo funkcije zamaknili za štiri presledke in pomembno je le, da je vse, kar je na tem nivoju, zamaknjeno za štiri presledke (oziroma več, znotraj novih gnezdenih stavkov). Uporaba tabulatorjev je dovoljena, a zelo odsvetovana. Če bralčev najljubši urejevalnik samodejno zamika s tabulatorji, si bo bralec naredil uslugo, če ga tega odvadi. Sicer pa bo bralca bolj kot zamikanje, ki se ga tako ali tako drži tudi v jeziku, v katerem je programiral doslej, motilo pozabljanje dvopičja. En teden, ne več. Obljubim. Potem se človek navadi.

Komentarje začenjamo z znakom #: vse od njega do konca vrstice tolmač prezre. Zamik komentarjev je lahko poljuben. Python ne pozna bločnih komentarjev (komentarjev prek več vrstic). Navadno jih pišemo tako, da vsako vrstico začnemo

s #. Drug, a ne pogost način, je, da del kode zapremo med trojne narekovaje. Tako jo spremenimo v niz (s trojnimi narekovaji pišemo nize prek več vrstic), s katerim ne počnemo ničesar, torej ne naredi ničesar.

Podpičja med stavki smemo pisati, treba pa jih ni (zato jih tudi ne pišemo), razen, če v eno vrstico napišemo več ukazov.

```
a = 1; b = a +2
```

Takšno programiranje se ne šteje za lepo in se ga izogibamo. Včasih, a ne pogosto, se primeri obratno, namreč, da želimo kak daljši stavek, navadno klic kake funkcije z veliko argumenti, razbiti na več vrstic. V tem primeru na konec vrstice damo levo poševnico, tako kot pri definiciji makrov v Cju.

```
a = \
1
```

A še to se zgodi redko, saj poševnica ni potrebna, če je zaradi odprtih oklepajev jasno, da se vrstica nadaljuje.

```
a = max(1, 5, 23, 12, 5, 3, 1, 2
   1, 4, 1, 5, 6)
```

Zamik v drugi in morebitnih naslednjih vrsticah je lahko poljuben, nič pa ni narobe, če jih poravnamo manj šlampasto kot v gornjem primeru.

O nenavadnem prirejanju v naši funkciji se bomo pomenili kasneje, za zdaj naj zadošča, da vemo, da

```
a, b = b, a+b
```

`a`-ju priredi `b` in `b`-ju `a+b`. Pri tem desno stran izračuna, preden začne prirejati, tako da se vrednosti `a` in `b` ne pokvarita, zato ne potrebujemo tretje spremenljivke, kot smo jo v C++. Ob tem takoj povejmo, da tu ne gre za kako rokohitrsko bližnjico, ki so jo v jezik uvedli v podporo lenobi. Trik je le stranska posledica nečesa večjega in splošnejšega.

Spremenljivk ne deklariramo, niti zanje ne zahtevamo ali sproščamo prostora. Podobno ne moremo navesti tipov argumentov funkcij. Funkcija dobi argument, s katerim jo pokličemo. Če je kakega neprimernega tipa, se bo zataknilo, ko se bo zataknilo. Kako je to videti, smo že spoznali: ko smo funkcijo poklicali s številom

in nizom, je prišlo do napake na mestu, kjer smo ju poskušali sešteti, ne prej.

Kaj pa, če bi jo poklicali z realnimi števili? Ali dvema nizoma? Še vedno deluje! Nejeverniku izpišimo prvih osem členov Fibonaccijevega zaporedja, ki se začne z "a", "t".

```
>>> for i in range(10):
...     print fibonacci(i, "a", "t")
...
a
t
at
tat
attat
tatattat
attattatattat
tatattatattattatattat
```

Seveda, čemu le ne bi delovala, saj z našima argumentoma `a` in `b` ne počne drugega, kot da ju malo sešteva. In nize pač lahko seštevamo, mar ne?

Python je *dinamično tipiziran* jezik: vsa koda v Pythonu je nekaj podobnega kot *template* v C++. Tipi spremenljivk se nikoli ne preverjajo (razen, če programer to stori eksplicitno, z ustrezno funkcijo), pač pa se lahko zgodi, da določena operacija, recimo seštevanje, ni mogoča na določenih tipih.

Po drugi strani je Python *močno tipiziran*, kar pomeni, da ne bo izzival nesreče s tem, da bo poskušal, ko kaj ne gre skupaj, primerno pretvoriti objekte tako, da bo šlo.[2] Primer za to je "napaka", ki smo jo storili pri klicu funkcije. Kot *dinamično tipiziran* jezik, je Python dopuščal, da sta a in b število in niz vse do tretje vrstice funkcije, kjer naj bi k številu 4 dodal niz "a". Kot *močno tipiziran* pa se ni poskušal izmazati s "4a" (kot bi storil JavaScript, kjer je glede na namen jezika to sicer tudi smiselno), temveč je povedal, da takšno seštevanje ni mogoče.

2 S par izjemami, brez katerih bi bilo težko živeti. Dovoljeno je, denimo, sešteti celo in realno število.

Naloge

1. Ugotovi, s kako velikimi števili še zna delati gornja funkcija. Zmore izračunati stoto Fibonaccijevo število? Pa tisočo? Sto tisočo? Koliko mest ima?

2. Napiši funkcijo za izračun fakultete.

Rešitve

1. Do sto tisoč gre brez težav, število pa ima 20899 mest, kar lahko doženemo takole

```
>>> s = fibonacci(100000)
>>> from math import log
>>> log(s, 10)
20898.623527635951
```

   Lahko pa izpišemo kar dolžino števila `s`, spremenjenega v niz.

```
>>> len(str(s))
20899
```

   Seveda pa ga lahko tudi izpišemo in z veliko potrpežljivosti ročno preštejemo števke.

2. Naloga od bralca pričakuje nekaj raziskovalnega duha in iznajdljivosti. Znajti se je moral namreč s funkcijo `range`, ki smo jo bežno že srečali. Rešitev je lahko takšna:

```
def fakulteta(n):
    f = 1
    for i in range(n):
        f *= i+1
    return f
```

   ali, lepše, takšna:

```
def fakulteta(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f
```

   Ljubitelji rekurzije pa so morda napisali takole:

```
def fakulteta(n):
    if n == 0:
```

```
        return 1
    else:
        return n * fakulteta(n-1)
```

Tak, ki ve, kar bodo ostali bralci izvedeli šele malo kasneje, pa je napisal preprosto

```
def fakulteta(n):
    return n*fakulteta(n-1) if n>0 else 1
```

Kasneje bomo spoznali še veliko preprostejšo rešitev v slogu funkcijskega programiranja.

Besednjak

Osnovni podatkovni tipi

Python pozna cela (`int`, `long`) in realna števila (`float`). Cela števila so dveh vrst. "Običajna", `int`, ki so lahko velika do $\pm 2^{31}$ ali $\pm 2^{63}$ ali kaj podobnega, odvisno od procesorja. Dolga cela števila (`long`) so omejena le z dolžino pomnilnika – kdor je reševal naloge na koncu prejšnjega poglavja, je že srečal število z več kot dvajset tisoč mesti. Dolgo število eksplicitno zapišemo tako, da na konec dodamo črko L; "dolgo" deset zapišemo kot `10L`. Vendar v praksi tega nikoli ne počnemo, saj Python sam spremeni število v dolgo, kadar je to potrebno.[3]

Python pozna tudi kompleksna števila (razred `complex`), s katerimi zna nekaj malega računati.

```
>>> (1+2j)*(4+3j)
(-2+11j)
```

Za kaj bolj zapletenega (e^{1+2i}...) je potrebno uporabiti primerno knjižnico; Pythonu je že priložen modul `cmath` z osnovnimi funkcijami (`sin`, `cos`, `exp`...) na kompleksnih številih.

Podatkovni tip `bool` je izpeljan iz `int`. Tako je iz zgodovinskih razlogov – tipa `bool` Python včasih ni poznal, zato so logične operacije vračale `0` in `1`. V Pythonu imena konstant navadno začenjamo z velikimi začetnicami, torej se konstanti tipa `bool` imenujeta `True` in `False`. Ker je `bool` izpeljan iz `int`, ga lahko uporabimo tudi na vseh mestih, kjer bi sicer pričakovali `int`, na primer pri indeksiranju.

Konstanta `None` pomeni nič. Vračajo jo, denimo, funkcije, ki ne vračajo ničesar.

---

3 Naj vas ne zmede, da imena tipov v Pythonu ne ustrezajo tem v Cju. Pythonov `float` je v resnici Cjevski `double` in `int` je v resnici `long`. Tako imajo tipi v Pythonu lepa imena (nekomu, ki ne bi poznal Cja, ime "`double`" pač ne bi pomenilo ničesar in bi si ob njem verjetno predstavljal par števil). Obenem pa ni razloga, da števil ne bi shranjevali z dvojno natančnostjo. Podobno velja za cela števila.

(Matematično razvneti bralec naj na tem mestu ne sitnari z Russlom. Pač, nekatere funkcije vrnejo rezultat, druge nič, torej `None`. Prevedite v angleščino, pa bo zvenelo prav.) Mnogokrat jo uporabljamo tudi, kjer bi v Cju uporabili `NULL`, se pravi, kjer bi lahko kaj bilo, a ni ničesar. `None` je tipa `NoneType`.

O nizih za zdaj povejmo le, da obstajajo in da jih lahko pišemo med enojne ali dvojne narekovaje. Za razliko od PHP imata obe vrsti narekovajev isti pomen. Tipa `char` Python nima.

Vdelanih tipov je še precej – seznam, terka, slovar, množica, funkcija, razred/tip, modul... – a z njimi se bomo srečali malo kasneje.

Operatorji

Za prirejanje uporabljamo enojni enačaj, =. Isto vrednost lahko prirejamo več spremenljivkam, vendar prirejanje ne vrača rezultata tako kot v Cju.

```
>>> a = b = 1
>>> a = (b = 2) + 1
Traceback (  File "<interactive input>", line 1
    a = (b = 2) + 1
          ^
SyntaxError: invalid syntax
```

Pač pa zna prirejanje nekaj drugega, kar bo prišlo prav: razpakirati zaporedje v spremenljivke. To lahko počne z marsičim (konkretno, z vsem, prek česar moremo z zanko `for`), recimo nizom,

```
>>> a, b = "xy"
>>> a
'x'
>>> b
'y'
```

Tole je sicer zabavno, a ne prav pogosto. Pač pa navadno razpakiramo terko. Terka je, če spet poškilimo za kako poglavje naprej, nekakšen seznam, ki ga navadno zapiramo v okrogle oklepaje. Naredimo lahko torej tole:

```
t = (7, 12, "tine")
a, b, c = t
```

Po tem bodo `a`, `b` in `c` imeli vrednosti `7`, `12` in `"tine"`. Spremenljivke `t` seveda ne potrebujemo, pišemo lahko kar

```
a, b, c = (7, 12, "tine")
```

Tudi oklepaji pravzaprav niso potrebni – pisati jih moramo le, kadar je to potrebno iz sintaktičnih razlogov, navadno zaradi kake prioritete. Torej,

```
a, b, c = 7, 12, "tine"
```

Tako kot v prvem primeru, kjer smo imeli ločeno spremenljivko t, se desna stran tudi tu še vedno izračuna pred prirejanjem. Kako v Pythonu zamenjamo vrednosti dveh spremenljivk?

```
a, b = b, a
```

Zdaj polistajte malo nazaj, do programa za izračun Fibonaccijevih števil, kjer boste našli

```
a, b = b, a+b
```

Zdaj vemo, kaj je ta stavek v resnici pomenil, ne?

Razpakirati je mogoče tudi celotne strukture, pri čemer pa si moramo pomagati z oklepaji.

```
t = 1, (2, 3)
a, (b, c) = t
```

Matematične operacije so takšne kot drugod: +, -, *, /. Deljenje celih števil je celoštevilsko.

```
>>> 1/2
0
```

Ker se bo to v Pythonu 3.0 spremenilo in bo `1/2` tam ena polovica, imajo že sedanje različice Pythona tudi operator //, ki vedno deli (in bo tudi v prihodnosti vedno delil) celoštevilsko. Kadar torej v resnici hočemo celoštevilsko deljenje, ne bo nič narobe, če že zdaj uporabimo //. Navsezadnje bo koda tako le jasnejša.

Operator ** pomeni potenciranje. Tako osnova kot potenca sta lahko neceli števili, le da mora biti pri neceli potenci osnova pozitivna (računanje `(-1)**0.5` nas bi pripeljalo v kompleksna števila, kadar želimo delati z njimi, pa moramo to

zahtevati bolj eksplicitno). % je ostanek po deljenju. Izračunamo lahko tudi ostanek po deljenju necelega števila z necelim.

Python želi biti čimbolj "zračen" in berljiv, zato logične operatorje pišemo z besedami, torej `or`, `and` in `not`. Kadar nastopajo v logičnih izrazih, so poleg `False` neresnični tudi `0`, `None`, prazen niz, prazen seznam, prazen slovar in vse ostalo, kar je lahko še prazno. Vsa ostala števila, nizi, seznami in drugi objekti so resnični (če jih ne definiramo drugače).

Logični izraz se, tako kot skoraj v vseh drugih jezikih, računa od leve proti desni (pri čemer ima `and` seveda višjo prioriteto od `or`) in to le, dokler rezultat ni znan. Pythonova posebnost je, da rezultat logičnega izraza ni nujno `False` ali `True`, temveč kar vrednost, pri kateri se je računanje končalo.

```
>>> True or False
True
>>> True and False
False
>>> 5 and False
False
>>> 5 and 12
12
>>> 5 or 12
5
>>> "Miran" or "Andrej"
'Miran'
>>> 1 and 2 and 0 and 3 or 4 and 5 or 6
5
>>> None or 0 or ""
''
```

Je videti neuporabno? Potem pač ne uporabljajte. Čeprav...

```
>>> ime = ""
>>> ime or "<brez imena>"
'<brez imena>'
>>>
>>> ime = "Miran"
>>> ime or "<brez imena>"
'Miran'
```

To bi pa utegnilo priti kje prav, ne? Pa še kaka podobna raba bi se našla, vendar tule ne bo odveč nekaj previdnosti. Python do različice 2.5 ni imel Cjevskega operatorja `?:`. Pa smo se znašli in `a ? b : c` zapisovali kot `a and b or c`. Če je `a` neresničen, `a and b` ne more uspeti in rezultat izraza bo vrednost `c`. Če pa je `a` resničen, dobimo `b`. Tako torej simuliramo neobstoječi `?:`, razen... No, razen če je `b` neresničen in v tem primeru bomo dobili `c`, tudi kadar je `a` resničen. Če se vam zdi tole nekoliko zmedeno, bo to zato, ker je tole nekoliko zmedeno. Ob dobronamernem pisanju takšnih zmedenih and-or izrazov se je zmedlo veliko ljudi in pisalo kodo, ki je včasih, a ne vedno, delovala, odkrivanje hroščev zaradi napak v takšnih izrazih pa je lahko kar naporno. Debate so se vlekle leta; če bi operator `?:` pisali tako kot v Cju, to ne bi bilo videti pythonovsko, boljšega predloga pa ni bilo in v pomanjkanju soglasja je benevolentni dosmrtni diktator Pythona presodil, da je boljše ne imeti tega operatorja kot imeti skrpucalo. Ker pa s(m)o zato še naprej pridno and-orali (včasih pravilno, včasih ne), je omenjeni diktator naredil izjemo in omenjeni operator vendarle dodal, čeprav zanj ni bilo dobre sintakse. Zdaj je nekoliko japonska:[4]

```
b if a else c
```

Nihče ne pravi, da je lepo, vendar nihče tudi ni našel nič boljšega.

Ob tem se `b` seveda izračuna le, če je `a` resničen. Če bi torej napisali

```
fibonacci(10) if 42 == 6*8 else fibonacci(11)
```

bi se poklical le `fibonacci(11)`.

Primerjanje pišemo po Cjevsko, ne pascalsko, se pravi == za enakost in `!=` za neenakost. Primerjanja je mogoče nizati.

```
>>> 1 < 2 < 3 < 4 != 5
True
>>> 1 < 2 < 3 < 4 != 5 < 4
False
```

Rezultat izraza je resničen, če so resnične vse relacije v njem. V praksi seveda ne srečamo takšnih klobas, zelo pa nam tak izraz pride prav pri ugotavljanju, ali je vrednost določene spremenljivke znotraj meja, npr. `if mn <= a <= mx` ali `if`

4 Zakaj naj bi bilo to japonsko, ne vem. Tako so ocenili moji študenti.

`"a" <= c <= "z"`. Primerjamo lahko marsikaj, ne le števil, temveč tudi nize, zaporedja, tipe...

Redko uporabljeni operator `is` preverja "istost" in vrne `True`, kadar sta oba operanda en in isti objekt, kot npr. v (nesmiselnem) izrazu `a is a`. Veliko programerjev v Pythonu zanj niti ne ve in primeri, ko bi ga dejansko potrebovali, so prej zgledi slabo zasnovanega programa kot kaj drugega. Redno ga uporabljamo le za primerjanje z `None`, v stavkih kot `if a is None` ali, v nikalni obliki, `if a is not None`, ki je ekvivalentna `if not a is None`. A tudi tu bi operatorja == in != naredila natanko isto. No, enako.

Operator `in` preverja, ali se določen objekt nahaja v seznamu, nizu, slovarju ali čem podobnem. Prav tako kot `is` ima tudi `in` nikalno obliko, `not in`.

Operatorji `~`, `|`, `&` in `^` predstavljajo *negacijo*, *ali*, *in* in *ekskluzivni ali* na bitih. Operandi morajo biti cela števila. Operatorja << in >> pomenita pomik v levo in v desno za določeno število bitov.

Vsi aritmetični operatorji imajo tudi različice, v katerih priredijo rezultat operandu. Tako `a += b` k `a` prišteje `b` in `a **= 2` kvadrira `a`. Pač pa Python nima operatorjev ++ in `--`. Iz določenih razlogov, ki jih bo bralec kasneje umel (ali pa tudi ne), bi povzročila več navlake kot koristi.

Kot v drugih jezikih tudi v Pythonu z oglatimi oklepaji indeksiramo (`l[12]`), z zavitimi pokličemo funkcijo (`fibonacci(12)`), s piko pa dostopamo do polj oz. metod objekta (`s.split()`) in do objektov znotraj modulov (`random.randint()`).

Štetje

V Pythonu seveda štejemo od `0`. Prvi element tabele, na primer, ima indeks `0`. Začetniku se to seveda zdi čudno (na nekem forumu se je nekdo usajal nad tem, kako zanikrno načrtovan jezik je Python, saj ima potem peti element indeks štiri – kdo se more navaditi česa tako bedastega!?), vendar resni programerji vemo, da je tako edino prav.

Ob različnih priložnostih moramo podajati intervale. Ti vključujejo spodnjo mejo, ne pa tudi zgornje. Števila od 5 do 12 so torej 5, 6, 7, 8, 9, 10, 11. Tisti, ki se jim zdi to čudno, naj bodo potolaženi: tudi to ni bedasto in nekonsistentno, temveč, enako kot prej, edino prav. Vam bo že prišlo pod kožo.

Le dva argumenta si poglejmo – tretji, grafični nas čaka, ko se bomo pogovarjali o indeksiranju. Koliko je števil od 5 do 12? 12-5=7, torej 7, bi rekli. In imeli bi prav, vendar le zato, ker interval od 5 do 12 vključuje eno mejo, ne pa obeh. Medtem ko se v kakem drugem jeziku mučimo z vprašanjem, koliko zvezdic bo izpisal program

```
for j = 5 to 12
    print "*"
next j
```

(navadno jih izpiše 12-5, torej 8?!) v Pythonu te dileme ni: ekvivalentni program v Pythonu izpiše sedem zvezdic, vse od "pete" do "enajste".

Drugi: če vzamemo vsa števila od 5 do 12 in od 12 do 15, dobimo vsa števila od 5 do 15. Če bi interval vseboval tudi zgornjo mejo, bi 12 dobili dvakrat. Hm, no, ja, kolikokrat pa takole združujemo intervale?! Večkrat, kot si mislite, večkrat.

Včasih ob intervalu podamo tudi korak. Če želimo vsa števila od 5 do 12 s korakom 4, bomo dobili 5 in 9. Korak je lahko tudi negativen. Tedaj mora biti začetni element seveda večji od prvega, ali pa bomo pač dobili prazno zaporedje. Tako kot prej velja tudi v tem primeru pravilo, da interval vsebuje prvi element, ne pa zadnjega. Vsa števila od 12 do 5 s korakom -1 so torej 12, 11, 10, 9, 8, 7, 6.

Kot preprost primer za pravkar povedana pravila si poglejmo funkcijo `range([zacetek, ]konec[, korak])`, ki sestavi seznam celih števil iz podanega intervala. Privzeta vrednost začetka je `0` in privzeti korak je `1`. Oglejte si naslednji pogovor in si ga vzemite k srcu.

```
>>> range(5, 12)
[5, 6, 7, 8, 9, 10, 11]
>>> range(5, 12) + range(12, 15)
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> len(range(5, 12))
7
>>> range(5, 12, 4)
```

```
[5, 9]
>>> range(12)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> len(range(12))
12
>>> range(12, 5, -1)
[12, 11, 10, 9, 8, 7, 6]
>>> range(12, 5, -4)
[12, 8]
```

Ukazi

Začnimo z ničemer: ukaz `pass` ne naredi ničesar. Uporabimo ga, kjer bi morali kaj reči, pa nimamo kaj – v funkciji, ki jo bomo še nekoč definirali, za zdaj pa bi jo radi pustili prazno, v zanki, ki je še ni, ob lovljenju izjem (`except`), za katere nam pravzaprav ni mar...

Ukaz `print` izpiše, kar mu damo izpisati. Če želimo izpisati več stvari, jih ločimo z vejico in `print` bo med njimi izpisal presledke. Po izpisu gre `print` v novo vrstico. Temu se lahko izognemo, če `print` končamo z vejico: v tem primeru bo dodal presledek.

```
>>> for i in range(10):
...     print "*",
...
* * * * * * * * * *
```

Vedênje ukaza `print` je prikladno za začetnika, večini pa se zdi grdo in nepraktično. Zaradi ohranjevanja združljivosti ostaja, kakršen je, v Pythonu 3.0 ga bodo degradirali v običajno funkcijo, ki se bo poleg tega vedla zgledneje. Kako si lahko pomagamo zdaj, s tem, kar imamo, bomo videli v poglavjih o datotekah in, predvsem, v onem o nizih, kjer se bomo naučili oblikovanja v slogu Cja (`%5.3f` in take reči).

Kadar se s Pythonom pogovarjamo v ukazni vrstici, `print` običajno izpustimo.[5]

5 V resnici učinek ni popolnoma enak. Ukaz `print` izpiše objekt, če ga izpustimo, pa dobimo objekt v podobni obliki, kot bi ga opisali v programu. Razlika je najbolj očitna

```
>>> 2+2
4
```

Ukaz `if` je videti takole:

```
if a > 12 or b == 42:
    a = 12
elif a < 0 or b == 2:
    a = 0
elif a > 10:
    b = 10
else:
    a = b = 0
```

Dela `elif` in `else` seveda nista obvezna. V gornjem primeru je `elif` koristen, ker bi morali brez njega pisati

```
if a > 12 or b == 42:
    a = 12
else:
    if a < 0 or b == 2:
        a = 0
    else:
        if a > 10:
(...)
```

za kar pa seveda ne bi dobili veliko točk za eleganco.

Oklepaji okrog pogojev niso potrebni, razen, seveda, kadar jih potrebujemo zaradi prioritete operatorjev (`or` znotraj `and`...). Če jih kdo želi pisati, to sme početi, vendar, se bo s tem izdal za prišleka.

Tudi zanka `while` je takšna, kot bi jo pričakovali: besedi `while` sledi pogoj, ki ga ni treba zapirati v oklepaje, nato dvopičje in zamik. Če ne bi vedeli za zanko `for`, bi prvih deset števil izpisali tako:

```
i = 1
while i <= 10:
    print i
    i += 1
```

ob nizih in jo boste opazili, če v konzoli enkrat poskusite, kaj se izpiše, če vpišete `'miran'` in kaj, če ukažete `print 'miran'`.

V zanki `for` se skriptni in prevajani jeziki pogosto razlikujejo in Python se tu zgleduje po drugih skriptnih jezikih. Zanka `for` pomeni iteracijo prek zaporedja, tako kot `foreach` v Visual Basicu, Perlu, PHPju[6], Javascriptu...

```
>>> l = ["Miran", "Miha", "Mitja"]
>>> for e in l:
...     print e
...
Miran
Miha
Mitja
>>> for e in "Miran":
...     print e, ord(e)
...
M 77
i 105
r 114
a 97
n 110
```

Za običajno zanko, takšno, v kateri gre nek števec od začetne do končne meje, lahko uporabimo funkcijo `range`, ki smo jo že videli. Če želimo šteti od `0` ali, še pogosteje, če nam je za števec vseeno in bi radi le nekajkrat ponovili del kode, izpustimo spodnjo mejo. Funkcija `range(n)` vrne prvih n števil (od `0` do `n-1`), torej zanka

```
for i in range(10):
    pass
```

desetkrat ne naredi ničesar. Če želimo (spet) izpisati števila od 1 do 10, pa rečemo

```
for i in range(1, 11):
    print i
```

Saj ni pretirano lepo, a tudi pretirano pogosto pravzaprav ne. Kot smo videli, ko smo se menili o funkciji `range`, ji lahko določimo tudi korak. Ta je lahko tudi negativen, pri čemer moramo seveda obrniti tudi meji. Števila od 10 do 1 torej izpišemo z

```
for i in range(10, 0, -1):
    print i
```

6 PHPja vešči bralec naj bo pozoren na drugačen besedni red v Pythonovi zanki.

Sveži priseljenec v Python pogosto pozabi, da moremo z zanko `for` prek poljubnega seznama, ne le seznama števil (kasneje bomo spoznali, da je `for` kos tudi še marsičemu drugemu). Če je torej `l` seznam nečesa, kar je vredno izpisati, bomo med začetnikom in nekom, ki je v Pythonu doma, ločili po tem, da prvi piše

```
for i in range(len(l)):
    print l[i]
```

drugi pa, seveda,

```
for e in l:
    print e
```

Druga koda je preglednejša, malenkost hitrejša, predvsem pa splošnejša, saj deluje tudi v primerih, ko dolžine `l`-ja ni mogoče določiti. (Je tudi to možno? Koliko zanimivih reči nas še čaka!)

V zankah `for` in `while` lahko uporabimo `break` in `continue`, katerih pomen je enak kot v drugih jezikih. Pythonova posebnost je `else`, ki lahko sledi zanki in se izvede, če se zanka ni prekinila z `break`. Navadno ga uporabljamo po zankah, v katerih kaj iščemo ali kaj poskušamo narediti; ko najdemo ali naredimo, zanko prekinemo z `break`, za ustrezne ukrepe po morebitnem neuspehu, se pravi takrat, ko se `break` ni izvedel, pa poskrbi `else`.

Tako si lahko radovedneži, ki jih že od mladih nog muči vprašanje, ali je katero od prvih stotih Fibonaccijevih števil deljivo s 1131, potešijo radovednost z naslednjim programom.

```
a = b = 1
for i in range(100):
    if a % 1131 != 0:
        print a, "je deljiv s 1131"
        break
    a, b = b, a+b
else:
    print "Nobeno od prvih 100 Fibonaccijevih stevil ni
deljivo s 1131"
```

Manj razburljiv primer je spodnja skripta, ki poskuša prebrati spletno stran in obupa po petih neuspešnih poskusih.

```
import urllib
for i in range(5):
    try:
        r = urllib.urlopen(url).read()
        break
    except:
        pass
else:
    print "Ne bo slo..."
```

Za `try` in `except` še nismo slišali, vendar si predstavljamo, kaj počneta. Več o njima kasneje. Ukaz `pass` smo uporabili, ker je znotraj `except` zavoljo sintakse potrebno nekaj reči in če ne želimo, da se ob napaki (običajno prekoračitvi časa čakanja znotraj `urlopen`) kaj posebnega zgodi, pač rečemo, naj gre napaka mimo.

Ukaza `goto` Python nima. Če ga kdo pogreša, naj ga bo sram.

Kako definiramo funkcijo, smo že videli: z `def`, ki mu sledi ime funkcije, nato pa imena argumentov v oklepaju. Videli smo tudi že, kako argumentom določamo privzete vrednosti (tako pač kot v drugih jezikih). Funkcije, ki sprejmejo poljubno število argumentov in podobna čudesa, prihranimo za kasneje. Tudi za to, da bi razčiščevali, ali se argumenti prenašajo po referenci ali vrednosti, je še prezgodaj.

Procedur v pascalskem pomenu besede ali Cjevskih funkcij tipa `void` Python nima: vse funkcije vračajo rezultat. Pač pa funkcija, ki ne vrne ničesar, vrne `None`. Rezultat vrnemo z ukazom `return <vrednost>`. Če vrednost izpustimo in napišemo samo `return`, se izvajanje funkcije konča, rezultat, ki ga vrne, pa je, kot rečeno, `None`. Ena in ista funkcija lahko vrača rezultate različnih tipov. Primer smo videli v funkciji za izračun Fibonaccijevih števil, ki zna računati tudi Fibonaccijeve nize. Prav tako lahko funkcija včasih vrne "zaresen" rezultat in včasih le `None`, kot bi lahko počela nekoliko prej napisana koda za branje spletne strani, če bi jo preoblikovali v funkcijo.

Lahko funkcija vrača tudi več kot eno vrednost? Deseto Fibonaccijevo število in še njegov kvadrat? Da in ne. Cja vajeni bralec refleksno pomisli na kazalec in oni, ki programira v C++, na argument v obliki reference, v katerega naj funkcija zapiše rezultat. Ne, ne, takšno razmišljanje o spremenljivkah in argumentih je Pythonu tuje. V resnici je veliko preprosteje in vse imamo že pripravljeno. Se spomnite

nenavadne lastnosti operatorja za prirejanje, ki omogoča, da napišemo `a, b = 3, 7`? Funkcijo napišemo tako:

```
def fibonacci(n):
    a = b = 1
    for i in range(n):
        a, b = b, a+b
    return a, a**2
```

Zdaj vrača terko, seznam dveh števil. Pokličemo jo lahko tako, da rezultat mimogrede razpakiramo.

```
a, a2 = fibonacci(10)
```

Če o terkah ne razmišljamo, je funkcija videti, kot da bi vračala dve vrednosti in tudi klic je takšen, da tidve vrednosti pač priredimo dvema spremenljivkama. V tem smislu funkcija lahko vrne tudi več vrednosti.

V resnici pa jih pravzaprav ne. Če bi poklicali `t = fibonacci(10)`, bi `t` ostal terka, ki bi jo razpakirali kasneje ali pa do njenih elementov dostopali z indeksi. Lahko torej funkcija vrne več kot eno vrednost? Kakor se vzame.

Naloge

1. Sestavi funkcijo za izračun največjega skupnega delitelja z Evklidovim algoritmom.

2. Napiši funkcijo `trojka(minc, maxc)`, ki izpiše vse Pitagorejske trojke (a, b, c), pri čemer je $a^2+b^2=c^2$ in `minc` $\leq$ c $\leq$ `maxc`.

3. Napiši rekurzivno funkcijo za izračun binomskih koeficientov!

4. Napiši funkcijo, ki ugotovi, ali je dano število *n* praštevilo tako, da ga poskuša deliti z vsemi števili manjšimi od korena iz *n*.

5. Popolna števila so števila, ki so enaka vsoti svojih deliteljev (brez sebe samega), tako kot je, npr. 28 = 1+2+4+7+14. Napiši funkcijo, ki za dano število ugotovi, ali je popolno.

Rešitve

1. Rešitev je praktično psevdo koda.

```
def evklid(a, b):
    while b > 0:
        a, b = b, a % b
    return a
```

2. Nič posebnega. Možne vrednosti c so med minc in maxc; k slednjemu moramo v zanki prišteti 1. Da bi vsako trojko izpisali le enkrat, spustimo a le do c deljenem s korenom 2 (razmislite, zakaj). Edina nenavadnost programa je ostanek po deljenju z 1. Tako preverimo ali je število celo. Trik je zaradi zaokrožitvenih napak nekoliko nevaren, vendar si to odpustimo, saj se učimo novega jezika, ne programiranja.

```
import math

def trojka(minc, maxc):
    for c in range(minc, maxc+1):
        for a in range(1, 1+c/math.sqrt(2)):
            b = math.sqrt(c**2 - a**2)
            if not b % 1:
                print a, int(b), c
```

3. Kot pri Evklidu se tudi tu pohvalimo s kratkostjo. Sicer pa: dolgčas.

```
def binomski(r, c):
    if not c or r == c:
        return 1
    return binomski(r-1, c-1) + binomski(r-1, c)
```

4. Edino, na kar moramo paziti pri reševanju, so meje v zanki `for`.

```
import math

def prastevilo(n):
    for d in range(2, 1+math.sqrt(n)):
        if not n % d:
            return False
    return True
```

5. Naloga je precej podobna prejšnji.

```
def popolno(n):
    s = 0
    for d in range(1, n):
        if not n % d:
            s += d
    return s == n
```

Osnovne podatkovne strukture

Zdaj pa je že čas, da se lotimo reči, specifičnih za Python. Najprej se bomo posvetili seznamom, terkam, slovarjem, nizom ... Vse te podatkovne strukture so del jezika: vanj niso le vdelane in na voljo programerju, temveč jih – predvsem terke in slovarje – Python sam interno uporablja, kjer le more.

Podatkovne strukture so med seboj usklajene in konsistentne. Vse indeksiramo na enak način, se prek njih na enak način sprehodimo z zanko `for`, tudi imena metod, ki delajo isto, so enaka... Ko se naučimo uporabljati eno, nam to pomaga pri naslednji. In še več: funkcija, ki zna delati z eno strukturo, bo kaj verjetno znala delati tudi s kako drugo.

Datoteka

Datoteka ne sodi ravno v poglavje o podatkovnih strukturah, sploh pa ne na prvo mesto, vendar nam bo znanje branja datotek prišlo prav v primerih in pri nalogah, zato se jih – čeprav na škodo reda v knjigi – vendarle lotimo kar takoj.

Datoteko lahko odpremo na več načinov. Najpreprosteje je poklicati konstruktor `file` ali funkcijo `open`, ki jima kot argument podamo ime datoteke in način, na katerega naj jo odpreta. Ta je lahko "r" za branje, "w" za pisanje in "a" za dodajanje, k čemur lahko dodamo še "b", s čimer povemo, da je datoteka binarna, zato naj sistem pusti znake za konec vrstice pri miru. Skratka, vse tako kot v Cju. Za nekaj dodatnih možnosti bo bralec izvedel v dokumentaciji.

```
>>> f = file("slika.png", "rb")
```

Zapremo jo na dva načina. Če jo želimo zapreti ročno, pokličemo metodo `close`. Če tega ne storimo, bo za zapiranje poskrbel destruktor: ko gre objekt, ki predstavlja datoteko, v smeti (če smo, na primer, datoteko odprli v funkciji, se to zgodi ob izhodu iz funkcije), se bo datoteka sama od sebe zaprla takrat. Pogosto pa datoteke niti ne priredimo spremenljivki: tedaj se pač zapre tedaj, ko izgine iz "konteksta". Gornjo datoteko torej zapremo z `f.close()`, prav tako pa bi se

zaprla, če bi napisali

```
>>> f = 12
```

Za objekt-datoteko po tem namreč ne ve nihče več, zato gre v smeti, destruktor pa mimogrede poskrbi še za zapiranje datoteke. V resnici metodo `close` v Pythonu pokličemo zelo redko: na datoteko navadno preprosto pozabimo, Python pa že ve, kaj storiti s pozabljenimi rečmi. Drug tipičen primerček bomo videli vsak čas.

Iz datoteke lahko seveda beremo. Metoda `read` prebere podano število znakov ali, če jo pokličemo brez argumentov, celotno datoteko. Rezultat vrne kot niz. Če gre za besedilno datoteko, bo vseboval besedilo, če za kaj drugega, kot recimo sliko, bo niz vseboval vsebino datoteke, kakršna pač je (vključno z znaki s kodo `0`, če je to morda zaskrbelo koga, ki je iz Cja vajen drugače).

Takole odpremo datoteko, preberemo njeno vsebino v niz, pogledamo, kako dolg je (toliko kot datoteka) in izpišemo prvih 30 znakov. (Z `[:30]` si za zdaj ne belite glave, o rezanju nizov se bomo še pogovorili.)

```
>>> f = file("slika.png", "rb")
>>> s = f.read()
>>> len(s)
61989
>>> s[:30]
'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x04\x00\x00\x032\x08
\x02\x00\x00\x00\x90'
```

Besedilne datoteke pogosto beremo po vrsticah. Metoda `readline` prebere eno samo vrstico, `readlines` pa vse vrstice, ki jih vrne kot seznam nizov. Še bolj praktično pa jih je brati z zanko `for`. Če iteriramo prek odprte datoteke (s sintakso, ki je enaka oni za iteriranje prek česarkoli drugega), po vrsti dobivamo njene vrstice. Prebrani nizi vsebujejo tudi znak za novo vrstico.[7]

```
>>> for l in file("stevilke.txt", "r"):
...     l
...
'6.5 0 47 55 127 280 \n'
'3.6 1 59 121 246 256 272 \n'
```

7 Tule se bralcu splača preveriti razliko med izpisovanjem s `print l` in izpisovanjem tako, kot smo to počeli zgoraj, torej samo `l`. Odkod izvira, pa bo zares razumel, ko bo prebral poglavje o razredih in posebnih metodah `__str__` in `__repr__`.

```
'11.2 2 8 70 99 167 187 201 227 284 \n'
'12.1 3 17 82 85 115 154 221 240 \n'
'16.2 4 52 122 128 138 179 220 \n'
'6.2 5 7 30 40 60 106 111 168 259 270 \n'
```

Datoteko smo odprli kar v glavi zanke, ne da bi jo prirejali kaki spremenljivki. Kdaj se datoteka zapre? Takoj, ko zanjo ne ve nihče več, torej, ko je konec zanke.

Če je datoteka odprta za pisanje, lahko vanjo pišemo. Najpogosteje uporabimo `write`, ki kot argument prejme niz, katerega vsebino zapiše v datoteko. Podobno kot pri metodi `read` lahko niz vsebuje besedilo ali pa karkoli drugega. Takole lahko prepišemo datoteko:

```
>>> s = file("slika.gif", "rb").read()
>>> file("slika-kopija.gif", "wb").write(s)
```

Takšno prepisovanje, pri katerem datoteko v celoti preberemo v pomnilnik, si seveda lahko privoščimo le, če datoteke niso prevelike.

Spremenljivka `s` niti ni potrebna, vse bi lahko opravili kar z

```
>>> file("slika-kopija.gif", "wb").write(file("slika.gif",
"rb").read())
```

Obe datoteki se zapreta, takoj, ko je vrstica izvedena.

Metode datotek so dokaj standardne – `seek`, `tell`, `flush`... Datoteka nudi tudi nekaj introspekcije – izvemo lahko, na kakšen način je odprta, kaj se uporablja kot znak za novo vrstico, s kakšno kodno stranjo je zapisana in podobno.

Na standardni izhod pišemo z ukazom `print`. Ker je ta v Pythonu pred verzijo 3.0 takšen, kakršen je, nam je včasih prikladneje gledati na standardni izhod kot na datoteko. Dobimo jo v modulu `sys`, pod imenom `sys.stdout`. Spodnja skripta dvakrat izpiše 42.

```
>>> print 42
>>> import sys
>>> sys.stdout.write("%i\n" % 42)
```

Razliki sta dve, ena pretežno v korist, druga v škodo `stdout`. Stavek `print` zna izpisati karkoli, tudi število 42. Metoda `write` zahteva niz. Slabost stavka `print`

je, da na konec izpisa doda prehod v prazno vrsto, `\n` (ali presledek, če na konec vrstice dodamo vejico), v `write` pa o tem odločamo sami. V praksi večinoma uporabljamo `print`.

Modul `sys` ima, mimogrede, še datoteki `stdin`, standardni vhod, in `stderr`, standardni izhod za napake. Vse tri, vhod in oba izhoda, je mogoče preusmeriti preprosto tako, da spremenljivkam `stdin`, `stdout` in `stderr` priredimo druge vrednosti.

```
sys.stdout = file("c:/log.txt", "wt")
```

Seznam

Sezname si bomo ogledali nekoliko temeljiteje kot ostale razrede v tem poglavju (in knjigi sicer) saj se bodo metode, načini indeksiranja in podobno kasneje ponavljali tudi pri drugih strukturah.

Seznam je objekt tipa `list`. Njegovi elementi so *objekti* poljubnega tipa; ker je vse objekt, je v seznam očitno mogoče stlačiti karkoli.

```
l = [1, 2, "miran", [], True, fibonacci, fibonacci(12)]
```

V tale seznam smo torej spravili dve števili, niz, seznam (prazen), spremenljivko tipa `bool`, funkcijo `fibonacci` in še dvanajsto Fibonaccijevo število. Dodali bi lahko še kak modul in razred, ko bi ju le znali uvoziti in definirati.

V praksi so elementi seznama tipično istega tipa (le kak `None` se prikrade vmes), zato za nadaljnje primere sestavimo nekaj običajnejšega.

```
l = ["Ana", "Berta", "Cilka", "Dani", "Ema", "Fani", "Helga"]
```

Kot smo omenjali v čisto prvem poglavju, smemo vrstico, ki ne zaključi vseh odprtih oklepajev, nadaljevati v naslednji, ne da bi morali to posebej označiti s poševnico. Pri seznamu nam to pogosto pride prav.

```
l = ["Ana", "Berta", "Cilka",
           "Dani", "Ema", "Fani",
       "Helga", ]
```

Zamik v naslednjih vrsticah je lahko poljuben, priporočen pa je, kot vedno, red. Na

konec seznama smemo, kot smo to storili zgoraj, dodati tudi vejico. Ta pride posebej prav, kadar med programiranjem dodajamo nove elemente ali spreminjamo njihov vrstni red, tako kot v takšnemle seznamu.

```
letalisca = [
    ”Ljubljana - Joze Pucnik”,
    ”London – Gatwick”,
    ”Pariz – Charles de Gaulle”,
]
```

Ker imajo vse tri vrstice na koncu vejico, jih smemo zamenjevati med seboj, ne da bi se morali pri tem mučiti z urejanjem vejic.

Indeksiranje. Do poljubnega elementa seznama pridemo z indeksiranjem. Indeksiranje je hitra operacija.[8]

```
>>> l[0]
’Ana’
>>> l[2]
’Cilka’
```

Indeksirati je mogoče tudi od zadaj. Zadnji element ima indeks -1, drugi od zadaj je -2...

```
>>> l[-1]
’Helga’
>>> l[-2]
’Fani’
```

Z rezanjem (*slicing*) dobimo podseznam. Pravila so takšna kot pri funkciji `range`: interval vključuje začetni element, ne pa tudi končnega.

```
>>> l[0:2]
[’Ana’, ’Berta’]
>>> l[2:5]
[’Cilka’, ’Dani’, ’Ema’]
```

Pravilo o zgornji in spodnji meji spet odlično deluje, rezina 2:5 vsebuje 5-2=3

8 Seznam je shranjen kot tabela kazalcev na objekte. Tega nam sicer dejansko ni potrebno vedeti niti, če razvijamo dodatke v Cju, saj do elementov seznama dostopamo prek ustreznih funkcij, ne neposredno.

elemente. Tule je priložnost še za dodatno motivacijo za način, na katerega deluje rezanje (in funkcija `range`).

Če si meja rezin ne predstavljamo kot indekse elementov, temveč kot mesta možnih rezov med njimi, so elementi od drugega do petega Cilka, Dani in Ema, torej drugi, tretji in četrti element. Slika tudi lepo kaže, da seznam, ki ga sestavimo iz elementov od ničtega do drugega in od drugega do petega, sestavljajo vsi elementi od ničtega do četrtega, pri čemer se drugi ne ponovi.

```
>>> l[0:2]+l[2:5]
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
```

V rezinah smemo uporabljati tudi negativne indekse ali kombinacijo pozitivnih in negativnih. Tako je `l[2:-2]` seznam brez prvih in brez zadnjih dveh elementov (kar premislite pravila, pa boste videli, da bo res tako).

```
>>> l[2:-2]
['Cilka', 'Dani', 'Ema']
```

Gornji primer je nekoliko umeten, v praksi pa nas pogosto zanima nekaj prvih ali zadnjih elementov ali pa vsi elementi razen prvih oziroma zadnjih toliko in toliko. Tudi to se zelo elegantno opiše in ujame z različno predznačenimi indeksi in pravili o gornji in spodnji meji. Eno ali drugo (ali obe meji) smemo izpustiti in v tem primeru se razume, da želimo dobiti vse od začetka oziroma vse do konca. Tako sta `l[:2]` prva dva elementa seznama in `l[2:]` vse od drugega elementa naprej (torej vse razen prvih dveh). `l[-2:]` sta zadnja dva elementa (vse od `-2`-gega naprej) in `l[:-2]` vsi razen zadnjih dveh – vse do `-2`-gega, a brez njega, torej manjkata minus drugi in minus prvi.

```
>>> l[:2]
['Ana', 'Berta']
>>> l[2:]
['Cilka', 'Dani', 'Ema', 'Fani', 'Helga']
>>> l[-2:]
['Fani', 'Helga']
>>> l[:-2]
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
```

```
>>> l[:]
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fani', 'Helga']
```

V zadnjem primeru smo odrezali ves seznam od začetka do konca. Neuporabno? Ne, ni, tako naredimo kopijo seznama. Mimogrede, kopija je plitva, saj oba seznama vsebujeta *iste*, ne le *enake* elemente.

K rezanju, končno, lahko dodamo še korak.

```
>>> l[0:5:2]
['Ana', 'Cilka', 'Ema']
```

Kot poprej smemo izpustiti začetek in napisati tudi `l[:5:2]`. Korak je kajpa lahko tudi negativen.

```
>>> l[5::-1]
['Fani', 'Ema', 'Dani', 'Cilka', 'Berta', 'Ana']
```

Tule (in le tule) se eleganca nekoli sfiži, saj ima seznam šest elementov, ne le pet. Razlog je pač v tem, da to ni seznam od petega do ničtega *brez ničtega*, temveč je tokrat vključen *tudi* ničti element. Kaj pa naredi tole?

```
>>> l[::-1]
```

Ob indeksiranju smo se zadržali precej časa, a ta ni bil vržen stran. Enaka pravila veljajo pri indeksiranju vsega, kar je mogoče indeksirati. Vse, kar smo zgoraj počeli s seznamom, bi lahko počeli tudi z nizom. Namesto prvih dveh elementov seznama bi pač dobili prvi črki niza in tako naprej.

Operacije. Elemente seznama je mogoče spreminjati, brisati, jih dodajati... O prirejanju ni vredno izgubljati besed.

```
>>> l
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fani', 'Helga']
>>> l[4] = "Erika"
>>> l
['Ana', 'Berta', 'Cilka', 'Dani', 'Erika', 'Fani', 'Helga']
```

Element seznama pobrišemo z `del`. Beseda `del` je rezervirana, videli jo bomo še v neki drugi, bolj zanimivi vlogi, v seznamu pa jo uporabimo tako:

```
>>> del l[4]
>>> l
['Ana', 'Berta', 'Cilka', 'Dani', 'Fani', 'Helga']
```

Podoben učinek ima metoda `pop`. Tudi `l.pop(4)` bi pobrisal četrti element, a poleg tega še vrnil njegovo vrednost. Metodo `pop` lahko kličemo (in navadno tudi v resnici kličemo) brez argumentov; v tem primeru stori isto, kot če bi poklicali `pop(-1)`: vrne zadnji element in ga odstrani iz seznama.

Če bi radi pobrisali element, katerega indeksa ne poznamo, poznamo pa njegovo vrednost, uporabimo metodo `remove`.

```
>>> l.remove("Dani")
>>> l
['Ana', 'Berta', 'Cilka', 'Fani', 'Helga']
```

Elemente moremo seveda tudi dodajati, na vsaj štiri načine. Najpogosteje uporabimo metodo `append`, ki doda element na konec seznama.

```
>>> l.append("Ingrid")
>>> l
['Ana', 'Berta', 'Cilka', 'Fani', 'Helga', 'Ingrid']
```

Če ne želimo dodajati na konec seznama, temveč kam vmes, uporabimo `insert`.

```
>>> l.insert(3, "Ester")
>>> l
['Ana', 'Berta', 'Cilka', 'Ester', 'Fani', 'Helga', 'Ingrid']
```

Ko bi ne bilo metode `insert`, bi lahko dosegli enak rezultat z

```
>>> l = l[:3] + ["Ester"] + l[3:]
```

Prednost metode `insert` ni le (očitno) večja preglednost, temveč tudi hitrost, saj `insert` dejansko vrine element,[9] v spodnjem primeru pa sestavimo nov seznam iz

9 Formalno je časovna zahtevnost takšnega `insert(i, o)` sicer O(n), saj mora premakniti vse elemente od i-tega za en element desno v tabeli. Ker pa je `list` predstavljen s tabelo kazalcev, je ta operacija, razen če je seznam res zelo dolg, zelo hitra – napisana je v Cju in jo lahko procesor izvede hitro, saj mora le premakniti podatke v kosu pomnilnika – torej se v primerjavi s kodo, napisano v Pythonu, zgodi praktično "v trenutku", se pravi v času O(1).

treh kosov, namreč prvih treh elementov `l`-ja, seznama, ki vsebuje element `"Ester"` in vseh elementov `l`-ja od tretjega naprej.

K seznamu lahko dodamo tudi seznam.

```
>>> l.extend(["Jasna", "Karla"])
>>> l
['Ana', 'Berta', 'Cilka', 'Ester', 'Fani', 'Helga', 'Ingrid',
'Jasna', 'Karla']
```

V Pythonu naj bi bilo vsako stvar mogoče narediti le na en, očiten način. Dodajanje seznamov je ena od grobih izjem. Ker sezname lahko seštevamo, kot smo že parkrat mimogrede storili, rezultat seštevanja pa je nov, združeni seznam, bi bilo očitno konsistentno, če bi tudi operator += nad seznami storil isto kot `extend`. Namesto da smo uporabili `extend`, bi lahko zgoraj napisali tudi

```
>>> l += ["Jasna", "Karla"]
```

Zadnji, manj očitni, a tudi ne prav pogosto uporabljeni način je prirejanje rezinam.

```
>>> l[3:3] = ["Cireja", "Creda"]
>>> l
['Ana', 'Berta', 'Cilka', 'Cireja', 'Creda', 'Ester', 'Fani',
'Helga', 'Ingrid', 'Jasna', 'Karla']
```

Tole je nekoliko izprijeno, priznam. Rezina 3:3 je namreč prazna, saj vsebuje vse elemente od tretjega do tretjega (a brez tretjega). Rezinam običajno prirejamo vrednosti tako, da jih zamenjamo z drugo rezino, tako kot v naslednjem primeru, ko bomo zamenjali vse elemente od tretjega do šestega s seznamom dveh elementov, namreč Dragico in Evgenijo.

```
>>> l[3:6] = ["Dragica", "Evgenija"]
>>> l
['Ana', 'Berta', 'Cilka', 'Dragica', 'Evgenija', 'Fani',
'Helga', 'Ingrid', 'Jasna', 'Karla']
```

Mimogrede, rezino lahko zamenjamo tudi s praznim seznamom,

```
>>> l[2:4] = []
>>> l
['Ana', 'Berta', 'Evgenija', 'Fani', 'Helga', 'Ingrid',
'Jasna', 'Karla']
```

To je ekvivalentno brisanju,

```
>>> del l[2:4]
```

Z operatorjem `in` izvemo, ali se določen element nahaja v seznamu ali ne.

```
>>> 'Berta' in l
True
>>> 'Dani' in l
False
```

Metoda `index` pove, kje v seznamu se (prvič) nahaja podani element, `count` pa, kolikokrat ga najdemo v njem. Če elementa v seznamu ni, `count` vrne `0`, `index` pa javi napako. Če je bralec že poučen o algoritmih in podatkovnih strukturah, ve, kdaj te funkcije uporabiti in kdaj ne: ker lahko element v seznamu najdemo le tako, da po vrsti pregledujemo elemente seznama, bomo takrat, ko nas čaka veliko iskanja po večjem številu elementov, namesto seznama raje uporabili množico ali slovar, ki sta prav tako vdelana v Python.

Metoda `reverse` obrne seznam. Obrniti seznam smo se sicer mimogrede naučili ob rezinah, vendar je razlika v tem, da `l[::-1]` sestavi nov seznam, `l.reverse()` pa ga obrne *na mestu* (torej `l.reverse()` obrne, spremeni sam seznam `l`). Metoda `sort` seznam uredi, prav tako na mestu. Kot argument ji lahko podamo funkcijo, ki jo bo uporabil za primerjanje; sprejeti mora dva elementa takšnega tipa, kakršni so elementi v seznamu, in vrniti `-1`, `0` ali `1`, glede na to, ali je prvi element manjši, enak drugemu ali večji od njega. Če funkcijo izpustimo, bo `sort` elemente primerjal, kot bi jih primerjali operatorji <, == in >. Med argumente lahko dodamo še `reverse=True`. V tem primeru bo seznam urejen od večjega proti manjšemu elementu.

Dva seznama je mogoče primerjati. Primerjanje deluje leksikografsko, podobno kot primerjanje nizov po abecedi: paroma jemlje elemente seznamov, dokler ne naleti na par, ki se razlikuje. Če se to ne zgodi, je "manjši" tisti seznam, ki je krajši. Če sta tudi enako dolga, sta, očitno, enaka.

```
>>> [1, 2, 3, 5] < [1, 2, 4, 5]
True
>>> [1, 2, 3, 5] < [1, 2, 4]
True
>>> [1, 2, 3] < [1, 2, 3, 5]
True
```

```
>>> [1, 2, 3] < [1, 2, 3]
False
```

Dolžine seznama ne dobimo s kako metodo (`len`, `length`, `size` ali kaj podobnega), temveč s funkcijo `len`. Razlogi za to so tehnični.[10]

```
>>> len(l)
8
```

Za konec še nekoliko nevarna reč: seznam lahko pomnožimo s celim številom.

```
>>> ["Ana", "Berta"] * 3
['Ana', 'Berta', 'Ana', 'Berta', 'Ana', 'Berta']
```

Kje se skriva nevarnost? Tule.

```
>>> l = [[]]*5
>>> l
[[], [], [], [], []]
>>> l[0].append(1)
>>> l
[[1], [1], [1], [1], [1]]
```

Na prvi pogled smo sestavili seznam, ki vsebuje pet praznih seznamov. V resnici pa imamo seznam s petimi kopijami (istega!) praznega seznama. Ko v prvo kopijo dodamo element `1`, se ta seveda pojavi tudi v drugih kopijah.

Kako, če smo že ravno pri tem, sestavimo seznam petih praznih seznamov? Preprostejši, a počasnejši način je

```
l = []
for i in range(5):
    l.append([])
```

Zaresen programer v Pythonu pa naredi nekaj, kar se bomo naučili šele kasneje, namreč

```
l = [[] for i in range(5)]
```

---

10 Dolžina je univerzalnejša reč kot kak `append` ali `remove`, ki ju poznajo le seznami, zato je implementirana na nekoliko drugačen način. Drži, na prvi pogled to ni videti prav lepo in da dejanske razloge razume šele, kdor dokaj dobro pozna interno implementacijo tipov v Pythonu, tudi ni preveč dobro opravičilo.

Naloge

1. Če Python ne bi znal primerjati seznamov, bi si morali ustrezno funkcijo napisati sami. Kakšna bi bila?

2. Poišči najmanjši element v seznamu.
   a. Vrni najmanjši element.
   b. Vrni najmanjši element in njegov indeks – mesto v seznamu, kjer se pojavi.
   c. Vrni najmanjši element in seznam vseh mest v seznamu, kjer se pojavi.

   Deluje funkcija tudi na seznamu nizov? Pa seznamu seznamov?

3. Sprogramiraj Eratostenovo sito v obliki funkcije, ki vrne seznam praštevil.

4. Napiši funkcijo, ki prejme seznam objektov in vrne seznam nepadajočih podseznamov. Tako naj za seznam `[1, 2, 3, 3, 4, 2, 4, 8, 7, 8]` vrne `[[1, 2, 3, 3, 4], [2, 4, 8], [7, 8]]`.

5. Nek stroj upravljamo s kontrolnimi nizi, sestavljenimi iz malih in velikih črk angleške abecede, pri čemer velika črka začne, mala pa konča operacijo. Operacije se morajo končevati v obratnem vrstnem redu, kot so se začenjale. Tako je niz ABCcbCDEedca oblikovan pravilno, niz ABCcbCDEeac pa ne, ker se operacija A konča pred operacijo C, čeprav se je tudi začela pred njo. Prav tako niz ABCbc ni oblikovan pravilno, ker ne zaključi operacije A, ABcba pa zato, ker zaključi operacijo c, ki je ni nikoli začel. Napiši funkcijo, ki pove, ali je kontrolni niz pravilno oblikovan.

6. Kakšen je rezultat izraza `["Ne", "Da"][x==y]`?

7. Napiši funkcijo, ki preveri, ali je dani niz palindrom (beseda, ki se nazaj bere enako kot naprej). Predpostavi, da niz že vsebuje zgolj male črke in nobenih drugih znakov.

Rešitve

1. Za začetnika je povsem primerna takšna rešitev.

```
def primerjaj(l, k):
    for i in range(min(len(l), len(k))):
        c = cmp(l[i], k[i])
        if c:
            return c
    return cmp(len(l), len(k))
```

Zanka teče do dolžine krajšega od seznamov. Za primerjanje elementov seznama uporabimo funkcijo `cmp`, ki vrača natanko to, kar potrebujemo, -1, 0 ali +1. Kadar je njen rezultat različen od `0`, sta elementa različna in primerjanje končamo. Če se zanka izteče, sta seznama do dolžine krajšega od njiju enaka in tedaj je manjši tisti, ki je krajši.

Izkušenejši bi naredil malenkost drugače: funkcija `zip` nam omogoča z zanko `for` prečkati dva seznama hkrati. O tem bomo pisali (veliko) kasneje, za zdaj pa le pokažimo elegantnejšo rešitev.

```
def primerjaj(l, k):
    for e1, e2 in zip(l, k):
        c = cmp(e1, e2)
        if c:
            return c
    return cmp(len(l), len(k))
```

2. Rešitev prve različice naloge je čista klasika.

```
def najmanjsi_a(l):
    najm = l[0]
    for e in l:
        if e < najm:
            najm = e
    return najm
```

Druga je nerodnejša. Lahko se je lotimo tako, da element poiščemo naknadno.

```
def najmanjsi_b(l):
    najm = l[0]
    for e in l:
        if e < najm:
            najm = e
    return najm, l.index(najm)
```

To je seveda vnebovpijoče grdo, saj klic `index` zahteva ponoven prehod čez seznam in primerjanje elementov z `najm`. Nalogo lahko rešimo podobno, kot bi jo v Cju.

```
def najmanjsi_b(l):
    najm, naji = l[0], 0
    for i in range(len(l)):
        if l[i] < najm:
            najm, naji = l[i], i
    return najm, naji
```

Podobno kot v prejšnji nalogi bi stari maček uporabil nekaj, česar še ne poznamo in rešitev zapisal malo drugače.

```
def najmanjsi_b(l):
    najm, naji = l[0], 0
    for i, e in enumerate(l):
        if e < najm:
            najm, naji = e, i
    return najm, naji
```

Za rešitev naloge pod točko c se naslonimo na drugo različico rešitve naloge b (izkušenejši pa bi se na zadnjo, tretjo).

```
def najmanjsi_c(l):
    najm, naji = l[0], [0]
    for i in range(len(l)):
        if l[i] < najm:
            najm, naji = l[i], [i]
        elif l[i] == najm:
            naji.append(i)
    return najm, naji
```

Razlika med nalogama c in b pravzaprav ni velika. V b smo si zapomnili indeks najmanjšega elementa, tu sestavimo seznam z indeksom najmanjšega elementa, torej `[i]`. Ko naletimo na manjši element, sestavimo nov seznam, ko na enak element, pa v seznam le dodamo nov indeks.

3. Za rešitev bi težko rekli, da je posebej elegantna, čeprav je spet kar blizu psevdokodi algoritma.

```
def eratosten(n):
    n += 1
    p = [True] * n
    prastevila = []
    for i in range(2, n):
        if not p[i]:
            continue
        prastevila.append(i)
        for j in range(i, n, i):
            p[j] = False
    return prastevila
```

K `n` smo kar takoj prišteli 1, sicer bi morali to početi v vsaki zanki posebej. Nato sestavimo seznam z `n+1` `True`ji. Prvih dveh ne potrebujemo, vendar ju je preprosteje spregledati kot sestavljati krajšo tabelo in potem stalno preračunavati indekse. V seznam `prastevila` bomo nato sproti zapisovali praštevila. V nadaljevanju se ne dogaja nič pretresljivega.

4. Nalogo je najpreprosteje reševati z rezinami.

```
def nepadajoci(l):
    zacetek = 0
    podseznami = []
    for i in range(1, len(l)):
        if l[i] < l[i-1]:
            podseznami.append(l[zacetek:i])
            zacetek = i
    podseznami.append(l[zacetek:])
    return podseznami
```

5. Ta naloga pa zahteva, da seznam uporabimo kot sklad. Metodo `pop` imamo, vlogo `push` pa prevzame `append`.

```
def kontrolni(s):
    sklad = []
    for c in s:
        if "A" <= c <= "Z":
            sklad.append(c.lower())
        elif not sklad or sklad.pop() != c:
            return False
    return not sklad
```

Ko naletimo na veliko črko, potisnemo na sklad enako malo črko. Če naletimo na karkoli drugega, mora biti sklad neprazen in vrhnji element sklada mora biti enak trenutni črki. Funkcija vrne `True`, če med branjem niza ne pride do napake in če je

sklad na koncu prazen.

Program lahko obrnemo še nekoliko drugače. Na sklad na začetku damo stražarja, namreč `None`, ki ne more biti enak nobenemu znaku v nizu. Tako nam ni potrebno preverjati njegove praznosti. Stavek `return` na koncu funkcije se s tem sicer zaplete, funkcija pa je hitrejša, saj se pogoj v zanki poenostavi. Mimogrede lahko tudi zamenjamo pogoj v `if` z metodo `isupper`, ki vrne `True`, če je niz sestavljen iz samih velikih črk.

```
def kontrolni(s):
    sklad = [None]
    for c in s:
        if c.isupper():
            sklad.append(c.lower())
        elif sklad.pop() != c:
                return False
    return sklad.pop() is None
```

6. `"Ne"`, če sta `x` in `y` različna, `"Da"`, če sta enaka.

7. Funkcijo? Je temu vredno reči funkcija?

```
def palindrom(s):
    return s == s[::-1]
```

Pohvalimo se lahko tudi, da funkcija ne deluje zgolj na nizih, temveč na vseh strukturah, ki jih je mogoče takole indeksirati. Naše navdušenje kali le, da pri tem naredimo kopijo niza. Če bi to že znali, bi težavo rešili s ščepcem funkcijskega programiranja.

```
def palindrom(s):
    return all(e==f for e, f in zip(s, reversed(s)))
```

V Pythonu 3.0 ta koda ne bi naredila kopije seznama, v 2.5 pa bi bilo potrebno `zip` zamenjati z `izip` iz modula `itertools`.

Terka

Terka (*tuple*[11]) je zelo podobna seznamu, le precej bolj omejena je. Ne moremo je spreminjati (za takšne razrede rečemo, da so *immutable*). Z njo ne moremo početi skoraj ničesar, le sestavimo jo lahko. Navadno jo zapišemo z okroglimi oklepaji.

```
t = (”Ana”, ”Berta”, ”Cilka”, ”Dani”, ”Ema”, ”Fani”, ”Helga”)
```

Kadar je očitno, da gre za terko, smemo oklepaje izpustiti. Namesto gornjega bi lahko pisali

```
t = ”Ana”, ”Berta”, ”Cilka”, ”Dani”, ”Ema”, ”Fani”, ”Helga”
```

Tole smo že obdelali, ko smo se pogovarjali o prirejanju, zdaj pa je čas za primere, ko oklepajev ne moremo izpustiti. Prvi je očitno prazna terka.

```
t = ()
```

Terka z enim elementom že ni več težavna, saj moremo napisati kar

```
t = ”Ana”,
```

Terke z enim elementom so sicer zabavna reč: tole spodaj ni terka

```
t = (”Ana”)
```

temveč zgolj niz "Ana". Če koga bega, ali oklepaji ne povedo svojega, naj se vpraša, kaj bi v Cju (ali kjerkoli drugje, tudi v Pythonu) pomenilo

```
a = (1)
```

Pač pa lahko seveda pišemo

```
t = (”Ana”, )
```

Pisanje oklepajev v terkah je priporočeno zaradi preglednosti, edini primer, ko jih praviloma izpuščamo, je že omenjeno prirejanje z razpakiranjem, tako kot tule

```
a, b = b, a
```

---

11 Terminološka opomba: angleški *tuple* je okrajšava za *n-tuple*, ta pa je posplošitev *triple*, *quadruple*, *quintuple*... V slovenščini govorimo o trojkah, četverkah, peterkah, v splošnem pa *n-terkah*, kar lahko po zgledu iz angleščine krajšamo v "terka". Podobna je hrvaška *torka* in poljska *krotka*, ki prihaja iz k-tke.

in `return`, ki vrača terko. To je namreč običajno, da je iz kode na prvi pogled očitno, kaj počnemo. Ob takšnih prirejanjih in vračanjih v resnici niti ne razmišljamo več o terkah.

Še en pogost primer, ko so oklepaji obvezni, so klici funkcij, ki jim kot argument dajemo terko. Če bi v seznam iz prejšnjega razdelka želeli dodati par `"Lenka", "Matjaz"` in to poskušali storiti z `l.append("Lenka", "Matjaz")`, bi se Python pritožil, da metoda `append` sprejme le en argument, ne dveh. Pravilno je `l.append(("Lenka", "Matjaz"))`.

Terko lahko sestavimo tudi s klicem konstruktorja `tuple`, ki mu moramo podati nekaj takšnega, kar je mogoče prehoditi z zanko `for`. Tipično je to druga terka ali seznam, lahko pa tudi, recimo, niz.

```
>>> tuple("Miran")
('M', 'i', 'r', 'a', 'n')
```

S terko je mogoče početi vse, kar smo počeli s seznami, le nobenih metod nima in spreminjati je ne moremo. Pozna indeksiranje v vsem razkošju – od spredaj in zadaj, s takšnimi in drugačnimi rezinami – a vrednosti lahko le beremo in ne prirejamo. Mogoče je izvedeti njeno dolžino, z operatorjem `in` poizvedeti, ali vsebuje določen element, ne moremo pa, recimo, prešteti pojavitev določenega elementa, saj nima metode `count`.

Čemu bi kdo uporabljal to pokveko, ko nam je vendar na voljo seznam, ki je toliko boljši? Ker je hitrejša? Niti ne, seznam in terka sta enako hitra. Terko uporabljamo v mnogih primerih, ki zahtevajo seznam, ki ga ni mogoče spreminjati. Prvi primer bomo srečali ob slovarjih in naslednjega, ko bomo izvedeli, kako delujejo funkcije.

Niz

Nize smo že večkrat oplazili, zdaj bomo o njih povedali nekaj več.

Zapis. Nize smemo pisati med enojne ali dvojne narekovaje. To je praktično predvsem, ker lahko znotraj niza zapisanega z dvojnimi narekovaji brez težav uporabljamo enojne narekovaje in obratno. Če pa bi želeli znotraj niza, zapisanega z dvojnimi narekovaji, uporabljati dvojne narekovaje, moramo prednje postaviti

levo poševnico. Slednja se tudi v Pythonu obnaša tako kot v mnogih drugih jezikih: `\t` je tabulator, `\n` nova vrstica in `\\` (ena) leva poševnica. Tiste, ki znajo PHP, bo morda motilo: leva poševnica ima enak pomen ne glede na to, ali je niz zaprt v enojne ali dvojne narekovaje. Med narekovaji ni razlike.

Niz se mora končati v vrstici, v kateri se je začel. Če želimo napisati niz, ki se razteza prek več vrstic, ga začnemo in končamo s trojnimi narekovaji.

```
s = """Tule imam povedati,
nekaj prav posebno dolgega,
dasiravno ne prav pomembnega."""
```

Včasih želimo, naj poševnica nima posebnega pomena. Najpogosteje nam to pride prav pri pisanju regularnih izrazov, kjer bi sicer hitro prišli do četvernih poševnic, česar nihče več ne bi mogel brati, v MS Windows pa tudi pri pisanju imen direktorijev (priporočena praksa je sicer uporaba običajnih poševnic). V tem primeru pred narekovaje damo črko `r` (kot raw).

```
>>> s = r"Tule je \ samo \ in tudi tole ni tabulator: \t"
```

Operacije. Nizi se vedejo podobno kot seznami in terke. Lahko jih indeksiramo in režemo.

```
>>> s = r"c:\python25\python.exe"
>>> s[3]
'p'
>>> s[:5]
'c:\\py'
>>> s[-4:]
'.exe'
```

Po njih lahko iteriramo z zanko `for`.

```
>>> for c in "miran":
...     print c, ord(c)
...
m 109
i 105
r 114
a 97
n 110
```

Z operatorjem `in` preverjamo, ali niz vsebuje določen podniz (in z `not in`, ali ga

ne), z `len` ugotovimo njegovo dolžino, nize lahko primerjamo, lahko jih seštevamo, niz lahko pomnožimo s celim številom (`"*" * 10` naredi niz desetih zvezdic), mu z operatorjem += dodamo drug niz...

Tole bo nekoliko presenetljivo (za Javance ne, za Cjevce pač): tako kot terka je tudi niz nespremenljiv (*immutable*).

```
>>> s[2]="x"
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Takšno prirejanje zahteva nekaj telovadbe:

```
s = s[:2]+"x"+s[3:]
```

Na srečo se izkaže, da v Pythonu posamezne znake že sestavljenega niza spreminjamo zelo redko.

Pardon, se bo pritožil bralec. Če so nizi nespremenljivi, kaj pa potem počne ravnokar omenjeni operator +=? Kadar nespremenljiv razred podpira operator +=, to stori tako, da vrne nov objekt. Razmislite tole

```
>>> l = k = [1, 2]
>>> k += [3]
>>> l
[1, 2, 3]
>>> k
[1, 2, 3]
>>>
>>> a = b = "Ana"
>>> b += "Marija"
>>> a
'Ana'
>>> b
'AnaMarija'
```

Do razlike, kot rečeno, pride, ker je seznam spremenljiv, zato += pripne nov seznam k obstoječemu seznamu. Čeprav smo `[3]` pripeli h `k`, se je spremenil tudi `l`, ker gre še vedno za en in isti objekt. Z nizi pa je drugače. Niza `b` ni mogoče spreminjati, zato prištevanje naredi nov niz, `a` pa še vedno obdrži staro vrednost.

Pa metode kot `insert` in podobne? Metod, ki bi spreminjale niz, ni. Če želimo v niz kaj vriniti, moramo sestaviti nov niz. Na srečo so nam v pomoč rezine.

```
>>> b = b[:3] + " - " + b[3:]
>>> b
'Ana - Marija'
```

Sicer nizu ne manjka metod. Znotraj niza lahko iščemo podniz ali ga zamenjujemo z drugim podnizom. Velike črke v nizu lahko pretvorimo v male in obratno. Nizu lahko odbijemo beli prostor (*whitespace*) na začetku in koncu, ga poravnamo z dodatnimi presledki... Vseh metod očitno nima smisla naštevati, bralec si bo že ogledal dokumentacijo. Le nekaj posebej uporabljanih pokažimo.

Metoda `split` razdeli niz na podnize, kakor jih ločuje podano ločilo. Privzeto ločilo je beli prostor.

```
>>> "Ana Berta Cilka Dani".split()
['Ana', 'Berta', 'Cilka', 'Dani']
>>> "Ana - Berta - Cilka – Dani".split()
['Ana', '-', 'Berta', '-', 'Cilka', '-', 'Dani']
>>> "Ana - Berta - Cilka - Dani".split(" - ")
['Ana', 'Berta', 'Cilka', 'Dani']
```

Nič tako strašno posebnega, a zelo uporabno, posebej skupaj z izpeljevanjem seznamov (ki pride še na vrsto).

Metoda `join` naredi ravno nasprotno: seznam nizov združi v en sam niz. Zasukana je nekoliko nenavadno: ločilu povemo, naj združi nize iz podanega seznama. Metoda nam prihrani veliko duhamorne kode.

```
>>> l = ["Ana", "Berta", "Cilka", "Dani"]
>>> ", ".join(l)
'Ana, Berta, Cilka, Dani'
```

"Ločilo" je lahko seveda poljuben, tudi prazen niz.

```
>>> "".join(l)
'AnaBertaCilkaDani'
```

Z metodama `startswith` in `endswith` se pozanimamo, ali se niz začne oz. konča z določenim podnizom. Pogosta raba je takšna:

```
if s.endswith(".gif"):
   (...)
```

Oblikovanje. Nad nizom je definiran operator %, ki z njim počne nekaj drugega kot s števili.

```
>>> s = "Koliko je %i/%i? Nekako %.3f." % (11, 5, 2.2)
>>> s
'Koliko je 11/5? Nekako 2.200.'
```

Torej: če nad nizom uporabimo operator %, na desni strani pričakuje terko. Niz mora vsebovati oznake, ki jih bo % zamenjal s števili, nizi ali čemerkoli že, terka na desni pa mora vsebovati ustrezno število objektov ustreznega tipa. Oznake so takšne kot v Cju, torej jih je bralec verjetno že vajen, razlika pa je v tem, da je za takšnole reč v Cju potrebne nekaj več telovadbe (klic funkcije `sprintf` z ustrezno pripravljenimi kazalci `char *`), v Pythonu pa je to vdelano v sam jezik.

Oznaka `%s` ne zahteva nujno niza: pripadajoči element terke je lahko karkoli. `%s` je torej univerzalen, vendar ne omogoča nastavljanja števila decimalnih mest in podobnega, kar omogočajo oznake, specializirane za posamezen tip.

```
>>> "%s %s %s" % ("a", 12, 3.14159265)
'a 12 3.14159265'
```

V primeru, da terka vsebuje en sam element, smemo namesto nje navesti kar sam element.

```
>>> print "x=%02i" % (1, )
x=01
>>> print "x=%02i" % 1
x=01
```

Od PHPja razvajeni bralec si najbrž želi še možnosti, da bi v niz vključil kar imena spremenljivk. Python nudi nekaj podobnega: v niz lahko vključimo imena spremenljivk, na desno stran operatorja % pa namesto terke postavimo slovar, ki vsebuje ustrezne vrednosti (kaka povrh pa nikogar ne zmoti). Ker o slovarjih še ne vemo ničesar, le pokažimo, kako doseči enak učinek kot v PHP.

```
>>> a = 11
>>> b = 5
>>> kol = 2.2
```

```
>>> print "Koliko je %(a)i/%(b)i? Nekako %(kol).3f" % vars()
Koliko je 11/5? Nekako 2.200
```

Imena spremenljivk torej v oklepajih vrinemo takoj za %, ustrezni slovar pa priskrbi vdelana funkcija `vars()`.

Unicode. Delo z nizi v Pythonu je preprosto, posebej za nekoga, ki se je doslej boril z njimi v Cju, ne pa v drugih skriptnih jezikih ... a le dokler ne pridemo do šumnikov. Pozorni bralec je opazil, da smo se jim v primerih izmikali. S problemom naborov znakov se mora spopadati vsak jezik, ki ni ravno od včeraj. Python se je s tem spopadel tako, da je starim, osembitnim nizom, `str`, pridružil nove, `unicode`.

Niz vrste unicode zapišemo z enakimi narekovaji kot običajne nize, le črko `u` (kot unicode) moramo dati prednje.

```
>>> s = "Navaden niz"
>>> t = u"Unicode niz"
```

Kakšna je razlika med njima, najboljše vidimo, če pogledamo, kaj bo z njima storil Python 3.0. Preimenoval ju bo v to, v kar je zdivergiral njun pomen v trenutnih verzijah. Sedanji `str` se bo po novem imenoval `bytes`, `unicode` pa bo postal `str`.

V "običajnem" nizu, trenutnem `str`, po novem ne smemo videti nujno besedila, temveč poljubno zaporedje bajtov. Da je tako, smo se prepričali že ob branju datotek. Metode za branje datotek, na primer `read`, vsebino datoteke vrnejo kot zaporedje bajtov v spremenljivki tipa `str`. Podobno je z drugimi viri podatkov, na primer s stranmi, ki jih zajamemo s spleta: rezultat je zaporedje bajtov.

Kaj moremo storiti s takšnim zaporedjem, je odvisno od tega, kaj vsebuje. Če smo prebrali datoteko s sliko, nas morda zanima njena širina. V datotekah .gif je zapisana v sedmem in osmem bajtu in to s tanjšim koncem (*little endian*). Nerodno je le, da `str`, takšen, kot je, vsebuje črke, ne bajtov (v smislu številk med 0 in 255), zato je potrebno njegove elemente pretvarjati z `ord`.

```
>>> s = file("/d/usesa.gif", "rb").read()
>>> ord(s[6]) + ord(s[7])*256
808
```

Velikokrat pa `str` (iz datoteke, spleta ali koderkoli že) vsebuje besedilo. V tem primeru ga lahko tudi izpišemo, vendar bo izpis zagotovo pravilen le, če gre za čisti ASCII. Že kaj se bo zgodilo z nizi, ki vsebujejo še šumnike, je odvisno od (medsebojno navadno skreganih) nastavitev operacijskega sistema, konzole in od vrste datoteke.

In sedaj pridejo na vrsto nizi vrste `unicode`. Recimo, da smo z neke spletne strani prebrali zaporedje bajtov 89, 101, 109, 194, 154, 97, 114. Shranjeni so kot niz vrste `str`. Že, da so nekatere številke večje od 127, pove, da ne gre za niz ASCII. Lahko ga poskusimo izpisati.

```
>>> print s
DemÂšar
```

Da bi ugotovili, kaj v nizu v resnici piše, moramo vedeti, s kakšnim kodnim naborom je zapisana stran. Pogledamo v izvorno kodo strani in izvemo, denimo, da v UTF-8. Zdaj je potrebno zaporedje 89, 101, 109, 194, 154, 97, 114 dekodirati iz UTF-8 v Unicode. Za to niz vrste `str` nudi metodo `decode`, ki dekodira zaporedje bajtov, ki ga vsebuje niz.[12]

Ko niz dekodiramo, dobimo objekt tipa `unicode`, ki ga Python praviloma zna tudi izpisati, če to dopuščata operacijski sistem in implementacija interpreterja in če so nameščeni ustrezni nabori znakov.

```
>>> t = s.decode("utf-8")
>>> print t
Demšar
>>> type(s), type(t)
(<type 'str'>, <type 'unicode'>)
```

Podobno bi ravnali tudi z nizi v drugih kodnih naborih, vključno s pred-

12 O tem, kaj je UTF-8 in kaj Unicode, se bo bralec poučil na ustreznih spletnih straneh, tule povejmo le, da je Unicode standard, ki vsakemu znaku, od angleških do kitajskih, priredi mesto v 32-bitni tabeli. UTF-8 je eden od možnih (in tudi najpopularnejši) načinov zapisa, v katerem je večina znakov zahodnih abeced shranjena v enem samem bajtu, nekateri, denimo naši šumniki, pa potrebujejo dva. Naloga `decode` je spremeniti znake, kodirane v UTF-8, v Pythonov interni zapis (32 bitov). Podobno nalogo `decode` opravlja tudi v osembitnih kodnih straneh, denimo CP-1250, kjer mora kode znakov zamenjati s pripadajočimi kodami v Unicode.

Unicodeovskimi, kot sta CP-1250 ali ISO 8859, le argumente v klicu `decode` bi ustrezno spremenili.

Niz vrste `unicode` ima obratno metodo `encode`, ki vrne niz vrste `str`, zapisan na ustrezen način, na primer z UTF-8. Z dodatnim argumentom povemo še, kaj naj se zgodi, kadar kakega znaka s podano kodno stranjo ni mogoče zapisati, tako kot, recimo, v CP-1250 ne moremo pisati kitajskih pismenk.

Ko torej delamo z nizi, ki prihajajo iz različnih virov, zapisanih z različnimi kodnimi stranmi, je najboljše, da jih takoj po branju dekodiramo v `unicode`, pred pisanjem pa kodiramo, v kar je pač treba.

Naloge

1. Napiši funkcijo, ki kot argument prejme EMŠO (kot niz) in vrne datum, mesec in leto rojstva osebe.

2. Napiši program, ki prebere vse datoteke v podanem direktoriju, ki imajo končnico .mp3 in izpiše podatke o njej (izvajalec, naslov...), ki jih prebere iz datoteke. (Namigi: funkcija `listdir` v modulu `os` vrne seznam datotek v podanem direktoriju; ta je lahko tudi `"."`. Alternativa je `glob.glob`. Kako je sestavljen mp3, poiščite na spletu.)

3. Napiši funkciji za približno primerjanje nizov:
   a. Dva niza sta enaka, če se ujemata v vseh znakih, razen na mestih, kjer je v enem od nizov "?". Tako sta niza "ASTRONO?IJA" in "ASTRO?OGIJA" enaka.
   b. Niza sta enaka, če sta enako dolga in se ujemata vsaj v 90 % znakov.

4. V seznamu nizov poišči tistega z največ znaki "a".

Rešitve

1. Prvi približek rešitve je trivialen. Prvi dve črki predstavljata dan, drugi dve mesec, tretji dve leto.

```
def datumIzEmso(emso):
    return emso[:2], emso[2:4], emso[4:7]
```

Manjša težavica je le, da letu manjka prva števka, `1` ali `2`. Ob predpostavki, da je verjetnost, da naletimo na Slovenca, ki je bil rojen v času Brižinskih spomenikov, zanemarljivo majhna, bomo k letnici prilepili `2`, če je prva števka letnice enaka `0`, sicer pa `1`. Prevod tega v Python je dobeseden. (Kakšna japonščina neki!)

```
def datumIzEmso(emso):
    return emso[:2], emso[2:4], ("2" if emso[4]=="0" else "1") +
emso[4:7]
```

Rokohitrska rešitev pa je takšna.

```
def datumIzEmso(emso):
    return emso[:2], emso[2:4], "12"[emso[4]=="0"] + emso[4:7]
```

Izraz `emso[4]=="0"` je sicer resničen ali neresničen (`True` ali `False`), vendar je tip `bool` izpeljan iz `int`. Indeksiranje zahteva `int`, torej ima, kar se tiče indeksiranja, `emso[4]=="0"` vrednost `0` ali `1`. Tako je rezultat `"12"[emso[4]=="0"]` enica ali dvojka.

Če je treba, lahko podatke pretvorimo v števila tako, da pokličemo funkcijo/tip `int`.

```
def datumIzEmso4(emso):
    return int(emso[:2]), int(emso[2:4]), \
           int("12"[emso[4]=="0"] + emso[4:7])
```

2. Namen naloge je bralca spomniti, da smemo vsebino datoteke prebrati v niz. Obenem pa zahteva tudi nekaj malega telovadbe z datotekami – branje zadnjih 128 bajtov, kjer se skriva opis, ki se začne s črkami "TAG", ki jim sledijo zahtevani podatki. Nizi s podatki so fiksne dolžine, neporabljeni del pa zapolnjen z znaki `\x00`, ki jih zamenjamo s presledki.

```
def mp3tagi(dir):
    import os
    for fname in os.listdir(dir):
        if fname.endswith(".mp3"):
            f = file(fname, "rb")
            f.seek(-128, 2)
            s = f.read().replace("\x00", " ")
            if s[:3] == "TAG":
                print
                print "Naslov: ", s[5:33]
                print "Izvajalec:", s[33:63]
                print
```

3. Tole ni posebna umetnost.

```
def enaka(s, t):
    if len(s) != len(t):
        return False
    for i in range(len(s)):
        if s[i] != t[i] and s[i] != "?" and t[i] != "?":
            return False
    return True
```

Pri reševanju ene od nalog smo že omenili funkcijo `zip`, posebej pa je nismo opisovali. Če je bralec tedaj uganil, kaj počne in si jo zapomnil, je tudi to nalogo morda rešil elegantneje.

```
def enaka(s, t):
    if len(s) != len(t):
        return False
    for si, ti in zip(s, t):
        if si != ti and si != "?" and ti != "?":
            return False
    return True
```

Pri reševanju naloge pod točko b uporabimo dva drobna trika.

```
def enaka(s, t):
    if len(s) != len(t):
        return False
    ujemajocih = 0
    for i in range(len(s)):
        ujemajocih += s[i] == t[i]
    return ujemajocih >= 0.9 * len(s)
```

Rezultat izraza `s[i] == t[i]` obravnavamo, kot da bi bil število (`0` ali `1`). Drugi trik ni Pythonovski, temveč splošen. Izraz v stavku `return` bi lahko zapisali tudi kot `ujemajocih/len(s) >= 0.9`, vendar bi v takšni obliki pri praznem seznamu dobili deljenje z `0`. Tako, kot smo ga preobrnili v funkciji, pa deluje tudi s praznimi seznami.

Ko se bomo pogovarjali o funkcijskem programiranju, pa se spomnite vrniti k tej nalogi in razvozlati tole rešitev:

```
def enaka(s, t):
    from operator import eq
    return len(s)==len(t) and sum(map(eq, s, t)) >= 0.9 * len(s)
```

4. Tole je le variacija naloge, ki smo jo že reševali, namreč iskanje najmanjšega elementa v seznamu.

```
def najvec_ajev(l):
    naj_ajasti, naj_ajev = None, -1
    for s in l:
        ajev = s.count("a")
        if ajev > naj_ajev:
            naj_ajasti, naj_ajev = s, ajev
    return naj_ajasti
```

   Za prazne sezname bi bilo lepše sprožiti izjemo. Ker tega še ne znamo, smo funkcijo napisali tako, da v takem primeru vrne `None`. Če seznam ni prazen, pa ima že prvi niz več kot -1 znakov "a" (čeprav morda `0`).

Slovar

Slovar (*dictionary*, asociativno polje, *associative array*), ki ga predstavlja razred `dict`, je podatkovna struktura, v kateri do elementov dostopamo prek unikatnih ključev. Slovar ni urejen: elementi niso zloženi ne po vrstnem redu dodajanja, ne po kaki drugi za programerja uporabni relaciji urejenosti. Zagotavlja nam hiter dostop do posameznih elementov, hitro dodajanje in brisanje, cena, ki jo moramo plačati, pa je nekoliko večja poraba pomnilnika.[13]

Kadar slovar eksplicitno definiramo, za to uporabimo zavite oklepaje, znotraj katerih naštejemo pare *<ključ>: <vrednost>*.

```
d = {"Miha": 12, "Miran": 29, "Matevz": None}
```

Vrednosti so lahko različnih tipov, vendar so običajno, tako kot v seznamu, vse istega tipa. Glede ključa smo omejeni: biti mora nespremenljiv. Bralec, ki ve kaj o slovarjih, razume, zakaj je tako: če bi lahko ključ (potihoma, brez slovarjeve vednosti) spremenili, bi to zmedlo strukturo, s katero je slovar implementiran. Sicer pa zdaj vemo, zakaj je priročno, da terk in nizov ne moremo spreminjati: ko bi ne bilo tako, bi jih ne mogli uporabiti kot ključe v slovarju – kar bi bila škoda.[14]

---

13 Za firbce: slovar je implementiran kot razpršena tabela (*hash table*).

14 Že slišim ugovore: kaj pa števila? Kako da števila lahko nastopajo kot ključi, čeprav jih lahko spremenimo, a? Ne bo držalo. Tudi števila so nespremenljiva (*immutable*). 2 je 2

Slovar se obnaša kot seznam toliko, kolikor je to smiselno. Slovar je mogoče indeksirati, vendar kot indekse uporabljamo ključe, ne števil. Rezine nimajo smisla, saj vrstni red elementov v slovarju ni določen. Elemente je mogoče spreminjati, dodajati in brisati.

```
>>> del d["Miran"]
>>> d
{'Miha': 12, 'Matevz': None}
>>> d[42] = "7*6"
>>> d
{42: '7*6', 'Miha': 12, 'Matevz': None}
```

Prirejanje elementa s ključem, ki še ne obstaja, pomeni dodajanje, prirejanje z obstoječim ključem pa zamenjavo.

Z operatorjem `in` se vprašamo, ali slovar vsebuje določen ključ. Medtem ko pri seznamih in nizih to zahteva linearno preiskovanje, je operacija `in` v slovarjih hitra.

Metode `keys`, `values` in `items` vrnejo sezname vseh ključev, vseh vrednosti in vseh parov (terk) s ključem in vrednostjo. Pogosto jih uporabimo v zanki `for`.

```
>>> for k, v in d.items():
...     print "%s\t%s" % (k, v)
...
42      7*6
Miha    12
Matevz  None
```

Primer je poučen zaradi načina razpakiranja terke: podobno kot operator = zna terko razpakirati tudi `for`. Gornja koda je sicer brez potrebe zapletena, terka bi lahko ostala terka, saj operator % tako ali tako zahteva na desni strani terko.

```
>>> for t in d.items():
...     print "%s\t%s" % t
...
42      7*6
Miha    12
Matevz  None
```

---

in ne more nikoli postati 3. Če številu nekaj prištejemo, dobimo drugo število.

Če zanko for spustimo prek slovarja kar tako, vrača ključe – tako, kot da bi jo nagnali prek seznama, ki ga vrne metoda `keys`.

```
>>> for t in d:
...     print t
...
42
Miha
Matevz
```

Slovarjev ne moremo kar tako seštevati, pač pa metoda `update` k slovarju doda vsebino drugega slovarja (in, če tako nanese, povozi katero od obstoječih vrednosti). Metoda `pop` je podobna istoimenski metodi seznama, le da vedno zahteva argument, namreč ključ. Tudi tu vrne vrednost elementa s tem ključem in ga odstrani iz slovarja. Sem ter tja pride prav metoda `setdefault`, ki ji podamo ključ in vrednost. Če element s tem ključem že obstaja, ga pusti pri miru in vrne njegovo vrednost. Če ga še ni, ga naredi, nastavi na dano vrednost in jo tudi vrne.

```
>>> d = {"a": 42}
>>> d.setdefault("a", 48)
42
>>> d
{'a': 42}
>>> d.setdefault("b", 33)
33
>>> d
{'a': 42, 'b': 33}
```

Med ostalimi metodami omenimo le še `get`. Ta opravlja podobno vlogo kot indeksiranje, vendar v primeru, da želenega ključa ne najde, vrne podano privzeto vrednost.

```
>>> d.get(42, "ga ni")
'6*7'
>>> d.get(43, "ga ni")
'ga ni'
```

Naloge

1. V datoteki so našteti vsi zaposlen(c)i v nekem podjetju, vsak v eni vrstici. Imenu

delavca sledi tabulator (`\t`) in ime šefa (ta je lahko zaposlen v tem podjetu ali pa tudi ne). Vsak delavec ima le enega šefa. Napiši funkcijo, ki prebere datoteko v primerno podatkovno strukturo (se pravi slovar ;) in funkciji, ki za posameznika vrneta vse njegove nadrejene in vse njegove podrejene. Da bosta hitrejši, lahko namesto v enem podatke shranjujete tudi v dveh slovarjih.

2. Za tekstovno datoteko, ki ne vsebuje drugega kot s presledki in prehodi v novo vrstico ločene besede, izračunaj frekvence vseh besed.

Rešitve

1. Podatke bomo shranili v dveh slovarjih. Prvi, `nadrejeni`, bo za vsako osebo (ključ) shranil njegovega šefa (vrednost). Drugi, `podrejeni`, bo za vsakega šefa (ključ) shranil seznam vseh neposredno podrejenih (vrednost). Pri branju iz datoteke s `strip` odluščimo znak za prehod v novo vrsto, nato pa vrstico s `split("\t")` razdelimo v del pred in za tabulatorjem. Pisanje v slovar nadrejenih je preprosto. Pri pisanju v slovar podrejenih pa najprej s `setdefault` poskrbimo, da je oseba že v slovarju (če je, `setdefault` le vrne obstoječi seznam podrejenih, če ga ni, naredi novega). Nato vanj dodamo novega podrejenega.

```
def preberi(fname):
    podrejeni = {}
    nadrejeni = {}
    for line in file(fname):
        pod, nad = line.strip().split("\t")
        nadrejeni[pod] = nad
        podrejeni.setdefault(nad, []).append(pod)
    return podrejeni, nadrejeni
```

Funkcija, ki vrne nadrejene, kot argument dobi osebo in slovar nadrejenih. (Če bi šlo zares, bi uporabili razred; funkcija za iskanje nadrejenih bi bila metoda in slovar nadrejenih bi bil shranjen v objektu.) Nato se sprehodi po slovarju: izraz `oseba in nadrejen` je resničen, če ima oseba kakega nadrejenega. V tem primeru dodamo nadrejenega v seznam in iščemo njegovega nadrejenega.

```
def vrniNadrejene(oseba, nadrejeni):
    vsiNadrejeni = []
    while oseba in nadrejeni:
        oseba = nadrejeni[oseba]
        vsiNadrejeni.append(oseba)
    return vsiNadrejeni
```

Iskanje podrejenih bomo za vajo rešili na dva načina, rekurzivno in iterativno. Za

rekurzivno rešitev preverimo ali ima oseba kakega podrejenega in če ga ima, sestavimo seznam posredno podrejenih iz vseh seznamov, ki nam jih vrnejo rekurzivni klici funkcije za vse neposredno podrejene. K seznamu posredno podrejenih dodamo še neposredno podrejene in to vrnemo kot rezultat. Če oseba nima podrejenih, vrnemo prazen seznam.

```
def vrniPodrejene(oseba, podrejeni):
    if oseba in podrejeni:
        neposrednoPodrejeni = podrejeni[oseba]
        posrednoPodrejeni = []
        for podoseba in neposrednoPodrejeni:
            posrednoPodrejeni += vrniPodrejene(podoseba,
podrejeni)
        return neposrednoPodrejeni + posrednoPodrejeni
    return []
```

Iterativna rešitev je, izjemoma, privlačnejša od rekurzivne, to pa zaradi lepe lastnosti zanke `for`: če seznam, prek katerega iterira, znotraj zanke daljšamo, bo zanka iterirala tudi prek novih elementov. Tako nam ni treba drugega kot dopolnjevati seznam vseh podrejenih z novimi in novimi podrejenimi, ki jih dobimo kot osebe, ki so podrejene tem, ki so že v seznamu. No, tale opis je nekoliko zapleten; program je preprostejši.

```
def vrniPodrejene(oseba, podrejeni):
    vsiPodrejeni = [oseba]
    for oseba in vsiPodrejeni:
        if oseba in podrejeni:
            vsiPodrejeni += podrejeni[oseba]
    del vsiPodrejeni[0]
    return vsiPodrejeni
```

Na koncu programa ne smemo pozabiti odstraniti prve osebe s seznama, saj oseba, po katere podrejenih sprašujemo, ni podrejena sama sebi. Ker je kratko lepo, pa program še nekoliko skrajšamo na račun malenkost počasnejšega izvajanja.

```
def vrniPodrejene(oseba, podrejeni):
    vsiPodrejeni = [oseba]
    for oseba in vsiPodrejeni:
        vsiPodrejeni += podrejeni.get(oseba, [])
    return vsiPodrejeni[1:]
```

2. Naloga zahteva, da malo potelovadimo z nizi, datotekami in slovarji.

```
def frekvence(ime):
    f = {}
    for beseda in file(ime).read().split():
        f[beseda] = f.get(beseda, 0) + 1
    return f
```

Celotno datoteko preberemo v pomnilnik (`read`) in jo takoj razdelimo na besede (`split`). Če bi bila prevelika, bi jo brali vrstico za vrstico in vsako posebej razdeljevali na besede. Zanka `for` teče prek besed. Za vsako besedo iz slovarja preberemo število dosedanjih pojavitev, če se beseda v slovarju še ne pojavi, pa bo metoda `get` vrnila podano privzeto vrednost `0`. K temu prištejemo `1` in zapišemo nazaj v slovar.

Kdo, ki mu je bližji Perl, bi morda pomislil, da bi bilo v zanki preprosteje napisati `f[beseda] += 1`. *Preprosteje* prav zagotovo, žal pa ne bi delovalo, saj je potrebno besedo najprej dodati v slovar, preden lahko povečamo pripadajočo vrednost, se pravi števec. Privzetih vrednosti Python ne pozna.

Množica

Zadnja vdelana podatkovna struktura, ki si jo bomo ogledali, je množica. Ker smo vse oklepaje porabili za seznam, terko in slovar, množice ne moremo sestaviti s primernimi oklepaji, temveč le tako, da pokličemo ustrezni konstruktor, ki mu kot argument podamo kako zaporedje, na primer seznam.

```
>>> s = set([1, 2, 3])
```

V večini primerov sestavljamo prazno množico, ki jo bomo šele kasneje napolnili z elementi. Prazno množico dobimo, če pokličemo `set` brez argumentov.

Množica je v sorodu s slovarjem, zato sme vsebovati le elemente, ki se ne morejo spremeniti.[15]

Množici lahko z `add` in `remove` dodajamo in odvzemamo elemente. Variacija na `remove` je `discard`, ki element, če obstaja, pobriše, sicer pa se ne pritožuje (za razliko od `remove`, ki javi napako). Celotno množico lahko spraznimo (`clear`) ali ji dodamo vse elemente druge množice (`update`). Ali množica vsebuje določen element, preverimo z operatorjem `in`. Seveda pa razred `set` pozna tudi običajne matematične operacije nad množicami: unijo (`union`), presek (`intersection`), razliko (`difference`), simetrično razliko (`symmetric_difference`) in preverjanje ali je določena množica podmnožica (`issubset`) ali "nadmnožica" (`issuperset`) neke druge.

15 Prav tako kot slovar je predstavljena z razpršeno tabelo (*hash table*).

Ker je množica (`set`) spremenljiva, je ne moremo uporabiti kot ključ v slovarju, niti ne more biti vsebovana v drugi množici. Če bi to potrebovali, uporabimo zamrznjeno množico (`frozenset`), ki je nespremenljiva. Sestavimo jo tako kot običajno množico, podpira pa le tiste metode, ki množice ne spremenijo. Če že imamo množico, pa bi jo radi zamrznili, pokličemo `frozenset` in ji kot argument namesto seznama damo množico.

Naloge

1. Python včasih ni imel posebne podatkovne strukture za množice. S katero strukturo smo si pomagali tedaj? Napiši funkcije `add(m, e)`, `remove(m, e)` in `contains(m, e)`, ki v množico m (implementirano z ono, drugo strukturo) dodajo element `e`, ga iz nje pobrišejo in preverijo, ali ga vsebuje.

Rešitve

1. Pomagamo si, seveda, s slovarjem. Elementi množice so ključi, vrednosti pa karkoli, najbolj naravno je, če se odločimo za `None`.

```
def add(m, e):
    m[e] = None

def remove(m, e):
    del m[e]

def contains(m, e):
    return e in m
```

Tule pa je primer uporabe.

```
>>> s = {}
>>> add(s, "Miran")
>>> add(s, "Tone")
>>> contains(s, "Tone")
True
>>> remove(s, "Tone")
>>> contains(s, "Tone")
False
```

Medsebojna zamenljivost podatkovnih struktur

Konsistentnost podatkovnih struktur v kombinaciji z nepreverjanjem tipov je imenitna reč, saj omogoča pisanje funkcij, ki morejo delati z različnimi podatkovnimi strukturami, ne da bi morali to posebej načrtovati.

Za primer si poglejmo le metodo `join`.

```
>>> l = ["Ana", "Berta", "Cilka"]
>>> t = ("Ana", "Berta", "Cilka")
>>> d = {"Ana": 1, "Berta": 2, "Cilka": 3}
>>> s = set(l)
>>> f = file("samSebe.py")
>>>
>>> ", ".join(l)
'Ana, Berta, Cilka'
>>> ", ".join(t)
'Ana, Berta, Cilka'
>>> ", ".join(d)
'Cilka, Berta, Ana'
>>> ", ".join(s)
'Cilka, Berta, Ana'
>>> ", ".join(f)
'# Program, ki izpise samega sebe\n, for line in
file("samsebe.py"):\n,     print line,\n, print'
```

Vrstni red izpisanega ni vedno enak, ker so nekatere podatkovne strukture urejene, druge ne, zato je vrstni red pri slednjih na videz naključen. Pomembno pa je, da zna metoda `join` delati z vsemi, ne da bi morala biti pripravljena na različne možne tipe argumentov. Brez težav lahko napišemo svojo različico `join`, `zdruzi`, le da jo bomo napisali kot funkcijo, ki kot argument dobi ločilo in zaporedje, ne kot metodo.

```
def zdruzi(locilo, zaporedje):
    prvi = True
    s = ""
    for element in zaporedje:
        if not prvi:
            s += locilo
        else:
            prvi = False
```

```
        s += element
    return s
```

Tole deluje nad vsemi zaporedji, prek katerih nas lahko popelje zanka `for`.

Spodnja funkcija preveri, ali je prvi element zaporedja manjši od drugega.

```
def manjsi(zaporedje):
    return zaporedje[0] < zaporedje[1]
```

Funkcija sprejme (vsaj) sezname, terke in nize. Nad slovarji in množicami nima smisla, saj tam ne moremo govoriti o "prvem" in "drugem" elementu. Da bi delovala tudi nad datotekami, pa bi se morali še malo bolj potruditi.

Nad povedanim v tem razdelku se večina ne bi smela čuditi. Podobne lepe lastnosti ima celo STL v C++, kjer ista funkcija deluje na vseh podatkovnih strukturah, ki nudijo dovolj močne iteratorje. Razlika je le v tem, da se vzorcem (*template*) v C++ začetniki izogibajo, ker jih ne razumejo, profesionalci pa, ker se bojijo, kako se bodo nanje odzvali različni prevajalniki. Večina zato uporablja bolj ali manj le vzorce iz STL in morda sem ter tja še kak droben vzorček, ki ga napiše sama. V Pythonu pa na ta način programiramo, ne da bi se tega zavedali.

Spremenljivke, definicije, imenski prostori

Med spoznavanjem Pythona smo naleteli na več stvari, ki so morda pustile neprijeten vtis. Videli smo, da se nespremenljivi objekti včasih vedejo nerodno drugače kot spremenljivi - pri prvih operator += sestavi nov objekt, pri drugih spreminja obstoječega. Naleteli smo na težave, ko smo sestavljali seznam petih praznih seznamov. Najbolj očitno pa je izogibanje vprašanju o tem, ali se argumenti v funkcijah prenašajo po vrednosti ali po referenci.

Bralca smo ves čas tešili z obljubo, da bo vse to nekoč dobilo smisel. Večkrat smo že namignili, da Python svoje podatkovne strukture uporablja tudi interno, v implementaciji samega jezika. Zdaj, ko jih poznamo, je napočil čas za pojasnjevanje.

Poglavje bo od bralca, vajenega nižjenivojskih jezikov, zahtevalo nekaj miselnih preskokov in truda. Vendar jih je vredno.

Spremenljivka

Python nima spremenljivk. Ima le objekte. Vse je objekt: ne le seznami in slovarji, datoteke in nizi, ne le instance razredov, ki jih definiramo sami, tudi modul je objekt, tudi funkcija je objekt...

Ko napišemo `"Miran"` ali `1` ali `file("slika.gif")` ali `def f()` (in tako naprej) s tem ustvarimo objekt tipa `str` ali `int` ali `file` ali `function`. Kot je bilo posebej aktualno (in opazno, zaradi zapiranja) pri datotekah, takšen objekt, če ga ne priredimo nobeni "spremenljivki", pride in gre. Kaj se zgodi tule?

```
>>> [1, 2] + [3]
[1, 2, 3]
```

Python sestavi dva objekta tipa `list`, namreč `[1, 2]` in `[3]`. Nato preveri ali prvi podpira operator za seštevanje. Ker ga in ker sta oba operanda istega tipa,

pokliče operator + prvega objekta in mu kot argument poda drugi objekt. Operator sestavi nov objekt, ki predstavlja vsoto (`[1, 2, 3]`). Tolmač nato uniči objekta `[1, 2]` in `[3]`, izpiše `[1, 2, 3]` in nato uniči še tega. To lahko mirno stori, saj objekt ni več dostopen.[16]

Včasih objektu damo *ime*, da bi ga lahko še kdaj uporabili. Če bi gornji primer spremenili v

```
>>> a = [1, 2] + [3]
```

bi bil epilog drugačen: novi objekt je dobil *ime* "a", zato ga tolmač pusti pri življenju. Še enkrat, ker je pomembno: "a" je samo ime objekta. Če bomo o "a" razmišljali kot o spremenljivki v Cjevskem (in, pravzaprav, zbirniškem) pomenu besede, torej kot o programerju prijaznem simboličnem imenu za nek naslov v pomnilniku, bo vse videti protislovno, kot se nam je dogajalo doslej. Ne, "a" je ime objekta.

Če zdaj nadaljujemo z

```
>>> b = a
```

ima naš seznam kar dve imeni, `a` in `b`. Pri tem ne gre za reference v slogu C++, kjer bi bil po

```
char c;
char &d = c;
```

`d` nekakšen sinonim za `c`. Ne, tu sta `a` in `b` dve imeni objekta-seznama `[1, 2, 3]`.

```
>>> b.append(4)
>>> a
[1, 2, 3, 4]
```

Ker gre za en in isti objekt z dvema imenoma, je vseeno, katero ime uporabljamo. Podobno bi se obnašal C++. Tule pa se podobnost konča:

```
>>> a = "Miran"
```

16 Večina lupin ga ohrani še nekaj časa pri življenju, saj imajo spremenljivko _, s katero dostopamo do zadnjega izračunanega izraza. ipython na takšen način ohranja kar celo zgodovino izračunov.

Kakšna je zdaj vrednost `b`? Kdor odgovori prav in ve, zakaj je tako, je že doživel razsvetljenje in temu bo jasno še marsikaj drugega. Ime `b` se še vedno nanaša na objekt `[1, 2, 3]`. Stavek `a = "Miran"` je namreč ustvaril nov objekt `"Miran"` in mu dal ime `a`. Ker se isto ime ne more nanašati na dve različni stvari, `a` zdaj ni več ime objekta `[1, 2, 3]`. Tule je očitno pomembno, da `b` ni "sinonim za `a`", saj bi sicer tudi `b` spremenil vrednost in postal `"Miran"`, kot bi se to zgodilo v Cju (če bi ta dovoljeval, da je ista spremenljivka enkrat seznam, drugič niz).

Objekt `[1, 2, 3]` je torej preživel, ker je znan še pod drugim imenom, `b`. Smrtni udarec mu lahko zadamo tako, da rečemo

```
>>> b = file("slika.gif")
```

s čimer tudi `b` ne imenuje več seznama `[1, 2, 3]` in ta odide v smetnjak. Drug način, da pokončamo `[1, 2, 3]` je ta, da pobrišemo ime `b`.

```
>>> del b
```

Operator `del` pobriše ime in ker je bilo to edino ime, pod katerim je seznam znan, seznam odide v pozabo.

In tule je še eno vprašanje, da preverimo, ali so razsvetljeni res razsvetljeni.

```
>>> a = [1, 2, 3]
>>> b = a
>>> del a
>>> b
[1, 2, 3]
```

Da bo `b` še vedno `[1, 2, 3]`, je nekako jasno. Vendar: razumemo, kaj naredi `del a`? Kako, da ne uniči objekta-seznama, kot bi se to zgodilo v C++ (po čemer bi `b` obvisel v zraku in povzročil sesutje)? Odgovor je zapisan par vrstic višje: `del a` pobriše le ime `a`, ne pa objekta, na katerega se ime nanaša. Objekt je nemogoče pobrisati.[17] Objekt izgine sam, ko nihče več ne ve zanj. Pobrisati ga prej, bi bilo narobe, saj bi se nekatera imena potem nanašala na stvari, ki jih ni več. Pobrisati ga kasneje, je nesmiselno, saj objekti, do katerih ne moremo, zgolj tratijo pomnilnik.

---

17 Mimogrede, tudi ukaz `del` uporabljamo resnično redko. Sam ne pomnim, da bi ga kdaj uporabil v kak drug namen kot za kako razhroščevanje destruktorjev.

Ob slednjem moramo opozoriti, da so vredni življenja tudi objekti, ki sicer nimajo svojega imena, vendar moremo priti do njih prek drugih objektov.

```
>>> b = [1, 2, 3]
>>> l = ["A", b]
>>> del b
```

Po `del b` je seznam `[1, 2, 3]` še vedno živ, saj ga najdemo na drugem mestu v seznamu `l`. Njegovo "ime", če hočete, je `l[2]`. Če pa nadaljujemo z `l = 42`, z `del l[2]` ali z `l[2]=15`, pa seznam `[1, 2, 3]` končno škripne.

Ko bomo poslej govorili o spremenljivkah, se zavedajmo, da ne gre za spremenljivko kot ime pomnilniškega naslova, kot v Cju, temveč vedno in zgolj za ime nekega objekta.

Definicije funkcij

Naslednje vprašanje, ki se bo zdelo programerjem v prevajanih jezikih nenavadno, je: "kaj je definicija funkcije". Večkrat smo že ponovili: funkcija je objekt. Takšen objekt pač, ki ga lahko pokličemo, nima pa kakega posebno bogatega izbora drugih metod. Za zgled uporabimo kar našega ljubega Fibonaccija z začetka knjige.

```
def fibonacci(n):
    a = b = 1
    for i in range(n):
        a, b = b, a+b
    return a
```

Vse razen prve vrstice je koda funkcije – koda, ki se bo izvedla ob klicu funkcije. Torej: ta koda je v resnici funkcija. Vloga prve vrstice je, da temu objektu-funkciji priredi ime (`fibonacci`) in navede argumente, da bo tolmač vedel, kako jo klicati in kako ne.

Definicija funkcije se mora *izvesti*. Izvajanje definicije pomeni prirejanje imena kodi. Da, čisto tako kot `a = 12` priredi objektu 12 ime `a`, gornja definicija kodi funkcije priredi ime `fibonacci`.

Ker je definicija funkcije le koda, ki priredi ime kodi, lahko definicijo postavimo kamorkoli v kodo.

```
>>> if 12 < 5:
...     def fun(x):
...             print x
...
>>> fun(33)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
NameError: name 'fun' is not defined
```

V času pisanja knjige 12 ni bilo večje od 5, zato se definicija funkcije ni izvedla in funkcija zategadelj ne obstaja. Bolj realističen primer je takšen (čeprav to ni najelegantnejša rešitev, niti ni potrebna, saj takšna funkcija že obstaja):

```
if sys.platform == "win32":
    def zdruziPot(l):
        return "\\".join(l)
else:
    def zdruziPot(l):
        return "/".join(l)
```

V C in C++ bi morali za kaj podobnega predprocesorju del kode zapreti v `#if`, `#else`, `#endif`, tu s tem opravi običajen `if`, saj se definicija funkcije ne *prevede*, temveč *izvede* kot katerakoli druga koda.

Ker definicija funkcije ni nič drugega kot prirejanje imena kodi, tudi ime funkcije ni nič posebnega – ime pač, kot ime kateregakoli drugega objekta. Funkciji lahko dodelimo tudi dodatno ime, pobrišemo starega...

```
>>> fib2 = fibonacci
>>> del fibonacci
```

Funkciji smo priredili ime `fib2` in pobrisali staro ime. Mar to pomeni, da tudi funkcije lahko "umrejo"? Seveda. Če dodamo še `del fib2` ali `fib2 = "Tine"`, in če se funkcija ne valja še v kakem seznamu ali slovarju ali kje drugje, bo šla v smeti. Funkcija je objekt kot vsi drugi in tudi zanjo veljajo ista pravila kot za vse druge objekte.

Podobno je, kot bomo videli, z moduli in razredi. Uvoz modula sestavi objekt tipa

`modul` in mu priredi ime – pač ime modula. Definiranje razreda sestavi določen objekt (ta je tipa `classtype`) in mu priredi ime, tokrat seveda ime razreda. Tako kot smo zgoraj ravnali s funkcijo, lahko preimenujemo tudi razred ali modul. In prav tako lahko gre tudi razred v smeti, ko za njegovo ime ne ve nihče več.[18]

Imenski prostori

V Cju spremenljivke živijo v pomnilniku. Lokalne spremenljivke so na skladu, dolgotrajnejše na kopici ali v prostoru, določenem za statične spremenljivke... Za vsako spremenljivko se ve (in, po potrebi, izve), kje v (navideznem) pomnilniku se nahaja. Kako je s tem v Pythonu? Kje so njegovi objekti?

Kot se nemalokrat zgodi, do odgovora na vprašanje pridemo z razmišljanjem. Ko bi sami pisali jezik, kot je Python, kam bi dali spremenljivke? Povedali smo, in to vsaj trikrat, da so Pythonove spremenljivke zgolj imena za objekte. Programu so torej na voljo objekti (števila, nizi, funkcije, moduli, razredi...), ki imajo unikatna imena. Ali, obrnjeno drugače, vsakemu (obstoječemu) imenu pripada nek objekt.

Jasno? Seveda, slovar. Saj tako je, pravzaprav, tudi v prevajanih jezikih, le da tam slovar vzdržuje prevajalnik (*compiler*), tu pa ga pač tolmač (*interpreter*). Spremenljivke se torej nahajajo v slovarjih, da, običajnih, Pythonovih slovarjih, kjer so ključi nizi (`str`), vrednosti pa karsižebodi. Razlika med Pythonom in prevajanimi jeziki je, da je v Pythonu slovar spremenljivk dostopen tudi programu samemu.

Slovarjev spremenljivk je več. Eden vsebuje lokalne spremenljivke, takšne, ki so vidne znotraj funkcije in bodo izginile, ko bo funkcija opravila svoje. Druge so globalne – bodisi čisto zares globalne, bodisi globalne s perspektive funkcij v nekem modulu. Videli bomo, da so s pomočjo slovarjev definirani celo razredi, polja objektov... Bolj ali manj vse v Pythonu je slovar.

Na kakšen način pridemo do teh slavnih slovarjev? Enega smo pravzaprav že

18 Moduli so izjema: ko je modul uvožen, se shrani v slovar modulov, ki ga ohranja pri življenju zato, da se vsak modul uvozi samo enkrat. Zakaj je to dobro, bo postalo očitno ob razlagi modulov.

uporabili, ko smo spoznavali oblikovanje nizov z operatorjem %. Se spomnite funkcije `vars()`, za katero smo rekli, da bo priskrbela spremenljivke, ki jih omenjamo v nizu? Vdelana funkcija `vars()` vrne slovar z vsemi spremenljivkami, ki so vidne v danem koščku kode. Vendar je ta slovar "nagoljufan", sestavljen iz dveh osnovnih.

Slovar vseh lokalnih spremenljivk vrne funkcija `locals`. Če ste pridno preskušali primere iz knjige, vam bo `locals` v tem trenutku vrnil nekaj takšnega:

```
>>> locals()
{'pywin': <module 'pywin' from 'C:\Python25\Lib\site-
packages\pythonwin\pywin\__init__.pyc'>, 'fibonacci': <function
fibonacci at 0x013AF3F0>, 'ime': 'Miran', 'kol':
2.1000000000000001, 'urllib': <module 'urllib' from 'C:\Python25\
lib\urllib.pyc'>, '__doc__': None, 'math': <module 'math' (built-
in)>, '__builtins__': <module '__builtin__' (built-in)>,
'zdruzi': <function zdruzi at 0x013887F0>, '__name__':
'__main__', 'line': 'print', 'c': 'n', 'd': {'Cilka': 3, 'Berta':
2, 'Ana': 1}, 'f': <open file 'samSebe.py', mode 'r' at
0x013CE1D0>, 'i': 9, 'k': 'Miha', 'l': ['A', [1, 2, 3]], 's':
set(['Cilka', 'Berta', 'Ana']), 't': ('Ana', 'Berta', 'Cilka'),
'v': 7}
```

Tale slovar za razliko od onega, ki ga vrne `vars`, ni bil sestavljen šele ob klicu funkcije `locals`. Ne, to je pravi slovar, oni, v katerem so v resnici shranjene lokalne spremenljivke. Ko rečemo

```
>>> d
{'Cilka': 3, 'Berta': 2, 'Ana': 1}
```

je to le prikladna okrajšava, sintaktični sladkorček, če hočete. V resnici pa se zgodi tole:

```
>>> locals()["d"]
{'Cilka': 3, 'Berta': 2, 'Ana': 1}
```

Če ne verjamete, pa po daljši poti zapišimo `x = v + i` (kot je videti iz gornjega slovarja, spremenljivki `v` in `i` obstajata, njuni vrednosti sta 7 in 9, `x` pa še ne obstaja).

```
>>> locals()["x"] = locals()["v"] + locals()["i"]
```

Iz slovarja imen in pripadajočih objektov smo torej pobrali objekta, katerima sta prirejeni imeni `v` in `i`, ju sešteli ter vsoti priredili ime `x`. S tem smo naredili novo spremenljivko, `x`. Nejevernik bo brž preveril, če je res.

```
>>> x
16
```

Poleg `locals` obstaja, kot bi lahko uganili, še funkcija `globals`, ki vrne slovar globalnih spremenljivk. Včasih gre za "resnično globalne" spremenljivke, drugič to pomeni zgolj spremenljivke modula.

Slovarje posamičnih objektov dobimo v njihovem polju `__dict__`. Tako nam `math.__dict__` izda vso vsebino modula `math` (ki ga moramo predtem uvoziti), `obj.__dict__` pa metode ter polja in njihove vrednosti objekta `obj`.

Funkcije

Definicije funkcij smo že srečali. Zdaj je čas, da si podrobneje ogledamo, kako vse je mogoče obrniti definicijo in kako vse je mogoče funkcijo poklicati.

Osnovo poznamo: v oklepaju, ki sledi imenu funkcije, preprosto naštejemo argumente, brez kakršnih koli tipov. Tudi, kaj bo funkcija vračala, nam ni treba povedati. Vračala bo, kar bo hotela, kadar in če bo hotela.

Komentarji. Začnimo s tem, kar je avtorju bralčeve najljubše knjige o Pythonu najbolj tuje: dokumentiranje. V prvi vrstici funkcije lahko napišemo niz, morda tudi takšnega čez več vrst, ki ga v tem primeru zapremo v trojne narekovaje. Tak niz na izvajanje funkcije ne vpliva, saj niza nikomur ne priredimo, ga nikamor ne izpišemo... Pač pa tolmač opazi tak niz, ko izvaja definicijo funkcije in ga uporabi kot dokumentacijo funkcije (točneje, ker je funkcija objekt, ima lahko polja; dokumentacija se shrani v polje `func_doc`). Tam ga najdejo in uporabijo razvojna okolja in ga prikažejo kot pomoč pri pisanju (*tooltip*).[19]

```
def fibonacci(n):
    """Vrni n-to Fibonaccijevo stevilo"""
    a = b = 1
    for i in range(n):
        a, b = b, a+b
        return a
```

Privzete vrednosti. Videli smo tudi že, kako določimo privzete vrednosti: tako kot v mnogih drugih jezikih za imenom argumenta dodamo enačaj in vrednost. Seveda argumentom, ki imajo privzete vrednosti, ne smejo slediti takšni brez njih, kot v tejle napačno definirani funkciji.

```
def f(a=1, b): # sintakticna napaka!
    pass
```

19 Za pisanje takšnih komentarjev obstajajo vodila, ki si jih za red in disciplino dovzetni bralec lahko prebere na strani `http://www.python.org/dev/peps/pep-0257/`.

Privzeta vrednost ni nujno konstanta: uporabimo lahko poljuben izraz, ki ga je mogoče izračunati v trenutku, ko se *izvaja definicija* funkcije. Tole ni dobro:

```
def f(x=a):
    pass

a = 12
```

Takole pa gre

```
a = 12

def f(x=a):
    pass
```

Poljuben izraz je lahko tudi klic funkcije.

```
def f(x=fibonacci(10)):
    pass
```

Privzeti argument bo deseto Fibonaccijevo število. Na mestu, kjer definiramo funkcijo, mora biti funkcija `fibonacci` seveda že definirana. Funkcija `fibonacci` bo poklicana samo enkrat, namreč takrat, ko bo tolmač definiral funkcijo `f` in bo moral izračunati privzeto vrednost argumenta. (Tole si zapomnite za takrat, ko vas bo mikalo funkcijo napisati tako, da se bo privzeta vrednost izračunala ob *vsakem* klicu funkcije: to ne gre.)

Po referenci ali po vrednosti? Vprašanje za Python ni smiselno. Točneje, nobeden od ponujenih odgovorov ni pravilen. Besedila o Pythonu se pogosto zatečejo v razlago, po kateri naj bi se spremenljivi objekti prenašali po referenci, nespremenljivi pa po vrednosti. Izvirnega avtorja te nesmiselne neumnosti bi bilo potrebno zgrabiti za ušesa.

Bralec te knjige pa že ve dovolj, da bo razumel. Tule je poučen primer.

```
>>> def f(a, b):
...     a = 13
...     b.append(2)
...
>>> x = [7]
>>> y = [3]
>>> f(x, y)
>>> x
```

```
[7]
>>> y
[3, 2]
```

Če bi razmišljali v terminologiji referenc in vrednosti, se `a` tu obnaša, kot da bi bil klican po vrednosti, saj spreminjanje `a`-ja ni vplivalo na `x`, `b` pa po vrednosti, saj je spreminjanje `b`-ja spremenilo tudi `y`. Razlaga?

Preprosta je, le mantre se moramo spomniti. Spremenljivka v Pythonu je zgolj ime za objekt. In tudi argument se obnaša tako kot vsaka druga spremenljivka. Argument je le lokalno ime (lokalno, znotraj funkcije) za podani objekt.

Pa sledimo poteku gornje kode. Najprej imamo objekta `[7]` in `[3]`. Imamo imeni `x` in `y`, prvo se nanaša na prvi objekt, drugo na drugega. Ko pokličemo funkcijo, ta poleg "globalnih" `x` in `y` vidi še dve novi, lokalni imeni, `a` in `b`. Prvo se nanaša na isti objekt kot `x`, drugo na isti objekt kot `y`, saj je bila funkcija poklicana z argumentoma `x` in `y`.

Takšno je stanje pred `a = 13`. Ta stavek pa sestavi nov objekt, `13`, in poskrbi, da se (lokalno) ime `a` po novem nanaša nanj. Objekta-seznama `[7]` to prav nič ne zadeva, zato `x` še vedno kaže na isti in enak objekt kot prej.

Z `b` in `y` je drugače. Ker v funkciji spreminjamo seznam `b`, to vpliva tudi na seznam `y`, saj gre za dve imeni za en in isti seznam.

Argumenti se torej ne prenašajo ne po vrednosti ne po referenci. Argumenti se v funkcijah vedejo kot nova imena za podane objekte; če njihovo vrednost *spreminjamo* (kar je možno le pri spremenljivih objektih), se spreminjajo tudi objekti, ki smo jih podali kot argument, če imenu *prirejamo* drug objekt, pa to na objekt, ki je bil podan kot argument, ne vpliva.

Tule moramo omeniti nesrečne operatorje za spreminjanje objekta na mestu, kot je +=. Kot vemo, ta spremenljive objekte spreminja, ko ga pokličemo za nespremenljiv objekt, pa sestavi nov objekt in ga priredi istemu imenu. V tem in le tem pogledu je sicer napačna razlaga klicev po referenci in po vrednosti pravilna: če v funkciji nad argumentom uporabimo operator +=, bo to spremenilo argument spremenljivega tipa, argumenta nespremenljivega tipa pa ne. Čeprav se ob to včasih spotaknemo, nekonsistentnost vendarle nima hujših posledic, saj ob

pisanju funkcije navadno predvidevamo, kakšnega tipa bodo argumenti ter kaj in kako moremo in smemo početi z njimi.

Lokalne in globalne spremenljivke. Tole je pa enostavneje. Če spremenljivki, ki se pojavlja v funkciji, ničesar ne prirejamo, tolmač predpostavi, da gre za globalno spremenljivko. Točneje, gre za spremenljivko, ki je definirana "nekje izven funkcije". Zanimive so spremenljivke, ki jim kaj prirejamo. Glede na to, da deklaracij ni: kako določimo, ali naj bodo lokalne ali globalne? In če ne rečemo ničesar, kakšne so?

Ker smo že nekoliko starejši mački v Pythonu, znamo odgovor na zadnje vprašanje hitro poiskati sami.

```
>>> def f():
...     aaa = 12
...
>>> f()
>>> aaa
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
NameError: name 'aaa' is not defined
```

Dobro, to je razjasnjeno: če spremenljivka ni eksplicitno razglašena za globalno, je lokalna.[20] Python se je odločil za varnejšo različico od obratne. Ko ena funkcija kliče drugo, ji ta ne more zapackati spremenljivke, ki ima slučajno enako ime. Tako bi se namreč dogajalo, če bi bile spremenljivke globalne.

Spremenljivko razglasimo za globalno, tako da jo ... no ja, *deklariramo* kot takšno.

```
>>> def f():
...     global aaa
...     aaa = 12
...
>>> f()
>>> aaa
12
```

20 Na tem mestu izražam bralcem, ki so veščejši JavaScripta enako sočutje, kot ga pričakujem zase, kadar se, veščejši Pythona, podam v JavaScript. Tam je namreč ravno obratno: spremenljivka je globalna, če je z `var` ne deklariramo kot lokalno.

Zdaj pa se, glede na to, da bralstvo te knjige ni brez programerskih izkušenj, dogovorimo še nekaj: globalne spremenljivke so grdobija. Pravkar sem preveril 70.000 vrstic dolg projekt v Pythonu, pri katerem sodelujem, in nameril, da smo potrebo po globalni spremenljivki začutili (in se ji odzvali) natanko štirikrat.[21]

Mestni in poimenski argumenti. Ob klicu funkcije lahko podajamo argumente na dva načina. Najprej naštejemo argumente, katerih pomen je določen z njihovim mestom (da bo preprosteje, jih imenujmo mestni argumenti, *positional arguments*). Sledijo poimenski argumenti. To je uporabno predvsem pri funkcijah z velikim številom argumentov, ki imajo večinoma privzete vrednosti, ki jih le redko spreminjamo.

```
def button(widget, master, label, callback = None, disabled=0,
     tooltip=None, debuggingEnabled = 1, width = None, height
= None,
     toggleButton = False, value = "", addToLayout = 1):
```

V večini klicev te funkcije dejansko podamo le prve tri argumente, ostale vrednosti pa so že nastavljene tako, kot nam je običajno prav. Klicali jo bomo lahko takole

```
button(box, dlg, "Cancel")
```

Kaj, če bi želeli poleg tega nastaviti še `toggleButton` na `True`? Namesto duhamornega sloga C++,

```
button(box, dlg, "Cancel", None, 0, None, 1, None, None, True)
```

smemo pisati kar[22]:

```
button(box, dlg, "Cancel", toggleButton=True)
```

Pomen prvih treh argumentov je, kot rečeno, določen s položajem – to so pač prvi trije argumenti, `widget`, `master` in `label`. Na kaj se nanaša zadnji, pove

21 Že to je dober argument za to, da moramo posebej povedati, kdaj želimo globalno spremenljivko in ne, kdaj lokalno: manjkrat bo potrebno pisati deklaracije in tudi manjkrat se bomo zmotili, ker jih bomo pozabili, poleg tega pa začetniku tako namignemo, kakšne naj bodo praviloma spremenljivke: lokalne, ne globalne.

22 To seveda ni Pythonov izum. Če ne drugega, je bralec gotovo že videl samodejno generirano kodo uporabniških vmesnikov v Visual Basicu ali posnete makre v MS Officeu.

njegovo ime. Čeprav so poimenski argumenti najuporabnejši v navezi s privzetimi vrednostmi, smemo poimensko navajati tudi takšne, ki sicer nimajo privzetih vrednosti.

```
button(box, label="Cancel", master=dlg, toggleButton=True)
```

Naslednje pa ne bo delovalo:

```
button(box, label="Cancel", toggleButton=True)
```

Težava je v tem, da klic ne poda vrednosti argumenta `master`, ki nima privzete vrednosti.

Mestni argumenti morajo biti podani pred poimenskimi. Tole je prepovedano.

```
button(label="Cancel", box, master=dlg, toggleButton=True)
```

Poljubno število argumentov. Tega ne potrebujemo ravno vsak dan, vendar: je mogoče napisati funkcijo, ki prejme poljubno število argumentov? Nekaj takega kot `...` v C++? Da, pa še veliko elegantneje deluje.

```
def f(a, b, c=3, *arg):
    print "a=%s, b=%s, c=%s, arg=%s" % (a, b, c, arg)
```

Kot argument, pred katerega ime postavimo `*` (to seveda nima zveze s kazalci!), bo funkcija prejela terko, ki bo vsebovala vse neporabljene mestne argumente. Gornjo funkcijo lahko pokličemo z dvema argumentoma (predstavljala bosta `a` in `b`), tremi (v tem primeru je določen še `c`) ali več. Če jo pokličemo s sedmimi argumenti, bodo zadnji štirje končali v terki `arg`.

```
>>> f(1, 2)
a=1, b=2, c=3, arg=()
>>> f(1, 2, 5)
a=1, b=2, c=5, arg=()
>>> f(1, 2, 5, 15, 32, "X", f)
a=1, b=2, c=5, arg=(15, 32, 'X', <function f at 0x015479F0>)
```

Podobno lahko pograbimo tudi poimenske argumente, le da ti ne gredo v terko temveč, kot bi bralec utegnil uganiti, v slovar. Slovar bo, tako kot prej terka, vseboval le neporabljene poimenske argumente.

```
>>> def g(a, b, c=3, **kwargs):
...     print "a=%s, b=%s, c=%s, kwargs=%s" % (a, b, c, kwargs)
...
```

```
>>> g(1, 2)
a=1, b=2, c=3, kwargs={}
>>> g(b=1, a=3, c=4)
a=3, b=1, c=4, kwargs={}
>>> g(d=15, b=1, a=3, c=4, e=6)
a=3, b=1, c=4, kwargs={'e': 6, 'd': 15}
```

Seveda imamo lahko tudi oboje, terko in slovar. In seveda imamo lahko tudi samo terko in slovar ter nič drugega. Spodnjo funkcijo lahko kličemo kakor hočemo in v nobenem primeru ne bo naredila nič.

```
>>> def poziralkaArgumentov(*args, **kwargs):
...     pass
...
>>> poziralkaArgumentov(1, 2, 4, 56, 1, a=13, tralala=4)
```

(Da ne bi kdo mislil, da to ni uporabno! Razne knjižnice pogosto zahtevajo funkcijo za povratni klic, *callback*, nam pa zanj ni nič mar. Tedaj jim podtaknemo takšnega omnivora, pa naj ga kličejo, kakor želijo.)

Klic s seznamom argumentov. Kako pokličemo funkcijo z vnaprej pripravljenim seznamom argumentov? Se razumemo? Ne? Na začetku knjige smo si pripravili (tudi) Fibonaccija s takšno glavo:

```
def fibonacci(n, a=1, b=1):
```

Zdaj pa bi radi deseti člen Fibonaccijevega zaporedja, ki se začne s številoma `zac=(5, 6)`: `zac` je torej terka, ki vsebuje argumente. Seveda lahko kličemo

```
fibonacci(10,  zac[0], zac[1])
```

kar je povsem sprejemljivo, dokler je število argumentov dovolj majhno in, predvsem, dokler vemo, koliko jih je. Vsekakor pa je elegantneje

```
fibonacci(10, *zac)
```

Nekaj mestnih argumentov smo torej podali kot običajno (točneje: enega, `10`), ostale pa kar v terki. Število mestnih argumentov, ki jih dobi funkcija, je torej `1+len(zac)`. Ali jih bo marala toliko ali ne, je odvisno od funkcije. Bo že povedala, če ji kaj ne bo prav.

Radovedni bralec se je nemara vprašal, ali je "normalni" klic `fibonacci(10, 1,`

`1)` mogoče zamenjati z "nenormalnim" `fibonacci(*(10, 1, 1))`? Zlobni se radovednemu smeji, češ, to bi bilo res "nenormalno". Vendar se moti. Naj vam zaupam drobno skrivnost: interno se prvi klic prevede v drugega. In tudi klic `fibonacci(10, *zac)` deluje tako, da funkcija dobi terko `(10,)+zac`. Pythonove funkcije si v resnici podajajo terke argumentov.

Pa poimenski argumenti? Enako, seveda! Če imamo slovar `zac={a: 5, b: 6}`, moremo Fibonaccija poklicati s `fibonacci(10, **zac)`. Seveda smemo pisati tudi `fibonacci(10, **{a: 5, b: 6})` ali celo `fibonacci(10, **prvaClena())`, kjer je `prvaClena` neka funkcija, ki vrača slovar, ki vsebuje elementa s ključema `a` in `b` (ali samo enim ali celo nobenim od njiju, vsekakor pa ne sme vsebovati argumentov z drugačnimi imeni, saj Fibonacci tega ne mara).

Pravilo za zadnja razdelka je torej: v definiciji funkcije argumenta, ki sta označena z * in ** (vsake vrste je lahko le po eden), pobereta dodatne mestne in poimenske argumente. Ko kličemo funkcijo, pa z * in ** označimo terko in slovar z dodatnimi mestnimi in poimenskimi argumenti.

Anonimne funkcije (lambda). Včasih moramo na hitro, mimogrede, definirati kratko funkcijo, ki bo le nekaj malega izračunala in si ne zasluži niti imena. Če si zaželimo urediti seznam nizov `l`, vendar ne po abecedi temveč po dolžini, seznamovi metodi `sort` kot argument podamo funkcijo, ki naj jo uporabi za primerjavo namesto privzete `cmp`.

```
def primerjajPoDolzini(x, y):
    return cmp(len(x), len(y))

l.sort(primerjajPoDolzini)
```

Ker so takšni primeri pogosti – drug tipičen primer so odzivi na dogodke pri programiranju uporabniških vmesnikov –, je mogoče funkcije, kot je `primerjajPoDolzini` napisati kot anonimno lambda-funkcijo.

```
l.sort(lambda x,y: cmp(len(x), len(y)))
```

Lambda-funkcijo definiramo s ključno besedo `lambda`, ki ji sledijo argumenti (pri tem lahko uporabimo vse trike iz arzenala – privzete vrednosti, terke ...), sledi dvopičje in nato *izraz*, ki ga je treba izračunati iz podanih argumentov. To je vsa umetnost.

Takšna funkcija nima niti imena in če je nihče nikamor ne shrani, neopazno zapusti svet, ko je ne potrebujemo več. Tako gornja izgine takoj, ko je urejanje končano. V ostalih pogledih pa je čisto običajna funkcija. No, skoraj. Lambda-funkcija sme vsebovati le izraz. Vanjo ne moremo stlačiti ne (običajne) zanke `for` (dovoljeni pa so generatorski *izrazi*, ki jih bomo še spoznali), običajnega `if` (dovoljen pa je operator `if-else` po zgledu Cjevskega `?:`) ali česa podobnega. Niti `print`a ne. Dopušča pa klicanje drugih funkcij, vendar le tako, da jih nekako vključimo v izraz.

Zanimivo pri lambda-funkcijah je, da jim lahko celo priredimo ime. Funkcijo `primerjajPoDolzini` bi lahko definirali z

```
primerjajPoDolzini = lambda x,y: cmp(len(x), len(y))
```

Razen par značilnosti, ki bi jih izbrskali le forenziki, se tako definirana funkcija ne bi razlikovala od one "prave", kakršno smo napisali zgoraj.

Naloge

1. Napiši funkcijo, ki računa Fibonaccijeva števila poljubnega reda – torej takšno, pri katerih naslednji člen ni nujno vsota zadnjih dveh temveč, denimo, zadnjih štirih členov. Tako bi bila Fibonaccijeva števila tretjega reda, ki se začnejo z 1, 3, 6 videti takole: 1, 3, 6, 10, 19, 35, 64... Prvi argument funkcije naj pove, kateri člen nas zanima, ostali pa naj podajajo začetne člene, iz števila katerih funkcija tudi razbere red.

2. Napiši ovojnico za razhroščevanje funkcij. Funkcija kot argument dobi funkcijo, ki jo je potrebno poklicati in vse argumente. Ob klicu izpiše podane argumente, pokliče ovito funkcijo ter izpiše in nato vrne njen rezultat. Omogočati mora, da jo uporabljamo takole:

```
>>> a = ovojnica(fibonacci, 10, 1, 2, 3)
Klic:  <function fibonacci at 0x014794B0>
Argumenti:  (10, 1, 2, 3)
Poimenski argumenti:  {}
Rezultat:  423
>>> a
423
```

3. Je v Pythonu mogoče napisati funkcijo, ki zamenja vrednosti dveh spremenljivk? Je spodnji primer resničen ali ponarejen?

```
>>> def swap(a, b):
...       a, b = b, a
...
>>> a, b = 3, 4
>>> swap(a, b)
>>> print a, b
4 3
```

4. Uredi seznam oseb, ki so predstavljene z imenom in priimkom (npr. `["Humphrey Bogart", "Ingrid Bergman", "Paul Henreid", "Claude Rains", "Conrad Veidt", "Sidney Greenstreet", "Peter Lorre"]`), po priimkih. Predpostavimo, da je priimek vedno zadnja beseda (pri osebah z več imeni, a enim priimkom, to deluje pravilno, pri osebah z več priimki pa jih uvrsti glede na zadnji priimek).

5. Sestavi funkcijo, ki sestavi funcijo za množenje s konstanto. Uporabljali bi jo takole.

```
>>> f8 = mnozenje_s_k(8)
>>> f8(4)
32
>>> f8(3)
24
```

6. Uredi seznam EMŠO, na primer `emso = ['1103966500016', '0511998500017', '1807931000014', '2207968500015', '2004974000012', '1703944500012', '1509994000015', '1207963000021', '0912983500019', '1609987500015']`, po datumih rojstva.

7. Sprogramiraj funkcijo, ki vrne kompozitum dveh funkcij.

```
>>> from math import *
>>> cossqrt = kompozitum(cos, sqrt)
```

`cossqrt` je po tem funkcija, ki za dano število x izračuna $\cos(\sqrt{x})$.

```
>>> cossqrt(pi**2)
-1.0
```

Rešitve

1. Rešitev je šaljivo preprosta.

```
def fibonacci(n, *s):
    for i in range(n):
        s = s[1:] + (sum(s), )
    return s[0]
```

2. Rešitev ne zahteva posebnih trikov, vedeti moramo le, kako vse lahko pišemo glavo funkcije.

```
def ovojnica(f, *args, **kwargs):
    print "Klic: ", f
    print "Argumenti: ", args
    print "Poimenski argumenti: ", kwargs
    res = f(*args, **kwargs)
    print "Rezultat: ", res
    return res
```

3. Ponarejen. Empiriki to hitro doženejo tako, da ga poskusijo ponoviti. Teoretiki pa razmislijo o tem, kako se prenašajo argumenti in kaj je spremenljivka ter pridejo do enakega sklepa.

   Takšno funkcijo bi bilo mogoče napisati za specifične primere. Tako lahko "zamenjamo" dva seznama tako, da v resnici prepišemo vsebino iz prvega v drugega in obratno. V splošnem pa to ni mogoče: v Pythonu vrednosti dveh spremenljivk ne moremo zamenjati, ker spremenljivk v Cjevskem pomenu besede sploh nima. Načelno bi lahko zamenjali dve imeni v imenskem prostoru klicatelja, vendar bi to zahtevali kar nekaj grobosti do sklada.

4. Naj bo seznam shranjen v `l`. Uredimo ga z

```
l.sort(lambda x,y: cmp(x.split()[-1], y.split()[-1]))
```

   Rešitev je sicer kratka, a nekoliko neučinkovita, saj funkcijo `split` za vsak niz pokliče večkrat. Veliko boljšo rešitev si bomo lahko privoščili, ko bomo čez par poglavij spoznali izpeljane sezname in bomo lahko napisali

```
ll = [(x.split()[-1], x) for x in l]
ll.sort()
l = [x[1] for x in ll]
```

   Ali pa tedaj, ko bomo odkrili generatorje in napisali (ne prav pregledno) enovrstično

```
l = [x[1] for x in sorted((x.split()[-1], x) for x in l)]
```

5. Rešitev je na prvi pogled preprosta, v sebi pa skriva bistvo definicije funkcije v Pythonu.

```
def mnozenje_s_k(k):
    def f(x):
        return x*k
    return f
```

Če bi bil Python prevajan jezik, bi se funkcija `f` prevedla že pred izvajanjem programa. V Pythonu pa se definicija funkcije `f` *izvede* ob vsakem klicu funkcije `mnozenje_s_k`. Rezultat izvajanja definicije funkcije je, da se koda funkcije priredi imenu (`f`). Novosestavljeno funkcijo vrnemo kot rezultat. Rezultat vsakega klica `mnozenje_s_k` je nova in ne vedno ista funkcija `f`.

Zanimivo vprašanje je, kam se shrani `k`. Nanj ne bomo odgovarjali, radovednejši pa naj poškilijo, ali ima rezultat (npr. `f8` iz opisa naloge) polje `func_closure` in kaj piše v njem.

Gornjo rešitev lahko poenostavimo z uporabo lambda-funkcije.

```
def mnozenje_s_k(k):
    return lambda x: x*k
```

6. V poglavju o nizih smo že napisali funkcijo, ki iz EMŠO naredi terko z rojstnim datumom. Uporabili bomo podoben trik, le da bomo postavili leto pred mesec in tega pred dan v mesecu. Da bo koda preglednejša, bomo iz tega sestavili niz, lahko pa bi delčke zložili tudi v terko. Dobljena niza bomo primerjali s `cmp`.

```
emso.sort(lambda x,y: cmp(x[4:7]+x[2:4]+x[:2],
y[4:7]+y[2:4]+y[:2]))
```

7. Funkcija mora sestaviti in vrniti novo funkcijo, za kar bomo uporabili lambdo. Nova funkcija prejme neznano število mestnih in poimenskih argumentov ter zanje pokliče `g`, nad rezultatom `g`-ja pa še `f`.

```
def kompozitum(f, g):
    return lambda *a, **b: f(g(*a, **b)
```

Moduli

O modulih v Pythonu ni povedati skoraj ničesar razen tega, da jih uvozimo z `import`, kot smo že nekajkrat storili.

Uvoz – izvoz. Kako "ni povedati skoraj ničesar"? Kaj pa, recimo, kako jih napišemo?

Tako kot doslej. Saj smo jih že pisali. Da ne? No, prav. Funkcijo našega zvestega spremljevalca Fibonaccija shranite v datoteko z imenom *fibonacci.py*. Zdaj pa napišite `import fibonacci`. Deluje? No, in v modulu, ki smo ga ravnokar uvozili, se nahaja funkcija `fibonacci`, ki jo vidimo pod imenom `fibonacci.fibonacci`. Ko bi dali modulu `vaja.py`, bi ga uvozili z `import vaja` in dobili funkcijo `vaja.fibonacci`. Pa druge stvari? Spremenljivke, razredi? Karkoli definiramo v modulu, je pač vidno v modulu. Če v modul `fibonacci` dopišemo še `a = 12`, nam bo pač na razpolago še `fibonacci.a` in njegova vrednost bo 12.

Ob ukazu `import` se izvede celotna koda modula, čisto tako, kot če bi modul izvedli kot samostojen program. Če modul poleg tega, da definira funkcije, razrede in karsižebodi, tudi kaj izpisuje, se bo ob uvozu to tudi izpisalo. Razlika pa je v tem, da po izvajanju skripte ostanejo funkcije, spremenljivke, razredi in vse ostalo, kar skripta definira, vidni kot globalne funkcije, spremenljivke, razredi in vse ostalo. Ko uvozimo modul, pa ostanejo definirane v modulu. Modul v Pythonu je tako le neke vrste imenski prostor (*namespace*).

Modul smemo uvoziti kjerkoli. Pogosto ga uvozimo na začetku skripte, nihče pa nam ne brani uvažanja znotraj funkcije, ali celo znotraj kakega stavka `if` (lahko tudi znotraj `for`, vendar bi bilo to nekoliko neobičajno) .

Kljub zatrjevanju, da o modulih ni kaj reči: par trikov moramo le omeniti.

Uvažanje v lasten imenski prostor. Prvi trik je tale:

```
from fibonacci import *
```

Če naredimo tako, vsebino modula uvozimo kar v "svoj" imenski prostor. Funkcije `fibonacci` in spremenljivke `a` tedaj ne bomo klicali `fibonacci.fibonacci` in `fibonacci.a`, temveč kar `fibonacci` in `a`. Dasiravno to zveni privlačno, je takšno uvažanje toplo odsvetovano. Če ne zaradi drugega, zato, ker se zlahka primeri, da dva modula z istim imenom (npr. `a`) poimenujeta dve različni stvari.

Manj grdo je tole:

```
from fibonacci import fibonacci
```

Resda spet uvažamo v svoj imenski prostor, vendar smo uvozili le funkcijo `fibonacci`. Takšno uvažanje je najbolj v modi pri matematičnih funkcijah, kjer lahko rečemo

```
from math import sqrt, sin, cos, exp
```

Za modul math ni neobičajno ali grdo niti, če v svoj imenski prostor uvozimo vse funkcije.

```
from math import *
```

Izrazi, kot je `(r * cos(phi), r * sin(phi))` so pač videti lepše brez `math` pred imeni funkcij.

Ponovno uvažanje. Vsak modul se *navadno* uvozi le enkrat, četudi večkrat izvedemo `import`. Tako je smiselno iz vsaj dveh vzrokov. Nekateri moduli imajo inicializacijo, ki se sme izvesti le enkrat ali pa traja dolgo časa. Večina modulov uvaža druge module in ko bi ob vsakem `import` ponovno sprožili celotno verigo, to ne bi bilo dobro. Drugi razlog je, da nas tako ni strah naložiti modul le tam, kjer ga dejansko potrebujemo. Morda pišemo funkcijo, ki dela na lokalnih podatkih, kadar je potrebno, pa se odpravi po nove na internet. Ustrezni modul (`urllib`, `httplib`, `poplib`...) bomo tedaj pač naložili tam, kjer vemo, da bo nujno potreben in nič prej. Če ga bomo zaradi tega morda naložili ob vsakem klicu funkcije – komu mar? Po prvem vsi nadaljnji uvozi ne bodo vzeli praktično nič časa, saj ne bodo storili ničesar.

Zapomniti si torej velja, da uvažanje modula ni prav nič posebnega in ni niti približno podobno ukazu `include` v kakem Cju. Za primer: če je potrebno, lahko `import` zapremo v blok `try-except`:

```
try:
        import nekModul
except:
        nekModul = None
```

Če modul obstaja (in če med uvažanjem ne pride do napake) se bo uvozil, sicer pač ne. V nadaljevanju programa bi ga potem uporabljali takole:

```
if nekModul:
    nekModul.nekaFunkcija(a, b)
else:
    znajdiSeDrugace(a, b)
```

Za prikaz, kako so moduli in funkcije čisto običajni objekti: elegantnejša rešitev za gornjo situacijo je, takšna

```
(nekModul.nekaFunkcija if nekModul else znajdiSeDrugace)(a, b)
```

Lahko pa bi si funkcijo kar spravili v spremenljivko

```
nekaFunkcija =  nekModul.nekaFunkcija if nekModul else
znajdiSeDrugace
```

in jo potem kasneje klicali z

```
nekaFunkcija(a, b)
```

A vrnimo se k modulom. Zakaj smo rekli, da se modul *navadno* uvozi le enkrat? Zato, ker ga lahko, če res želimo, uvozimo znova. Navadno to počnemo med razvojem in testiranjem modula: v funkciji iz modula odkrijemo napako, zato jo popravimo in zdaj bi ga radi ponovno uvozili. To storimo z

```
reload(fibonacci)
```

Modul `fibonacci` mora biti predtem že uvožen. Da nas to ne bi preveč motilo (prvič `import`, drugič `reload`), lahko med razvojem modula `fibonacci` v skripto, ki ga testira, zapišemo kar

```
import fibonacci
reload(fibonacci)
```

Ko jo poženemo prvič, bo resda modul najprej naložila in ga nato naložila ponovno, vsakič naslednjič pa bo preskočila `import` in izvedla le `reload`.

Globalne spremenljivke. Modul ne vidi iz lastnega imenskega prostora (razen, če si pomagamo s kakim trikom, vendar se to ne dela). Ko v modulu delamo z globalnimi spremenljivkami, gre v resnici za spremenljivke znotraj modulovega imenskega prostora. Če v modul `fibonacci` dodamo funkcijo

```
def f():
        global aaa
        aaa = 12
```

se bo po klicu `fibonacci.f()` v modulu `fibonacci` pojavila spremenljivka `fibonacci.aaa`.

Ime modula. Ime modula se skriva v njegovi spremenljivki `__name__`. Tako bo `fibonacci.__name__` enak `"fibonacci"`. V praksi se to uporablja v en sam namen: če modul vpraša samega sebe, kako mu je ime (namreč tako, da preveri vrednost globalne spremenljivke `__name__` – globalne seveda le zase) in izve, da mu je ime `"__main__"`, potem se izvaja kot "glavni program", ne kot modul, ki se ravnokar uvaža. Na ta način pogosto pišemo module, ki vsebujejo kodo za lastno testiranje. Na konec fibonaccijevega modula lahko tako dopišemo

```
if __name__ == "__main__":
    if fibonacci(2) != 2:
        print "nekaj je narobe"
```

Ko se ob uvažanju modula izvede koda, ki jo vsebuje (definicije funkcij ipd.), bo na koncu prišel na vrsto še `if`. Ker gre za modul, bo ime enako imenu modula (`fibonacci`) in test se ne bo izvedel. Če pa taisto datoteko poženemo (`python fibonacci.py`, ali pa jo naložimo v okolje in poženemo), pa se bo test izvedel.

Prevedeni moduli. Po prvem uvažanju modula boste na disku, v direktoriju z modulom opazili datoteko z enakim imenom, a končnico .pyc. To je datoteka z modulom, prevedenim v bajtno kodo za Pythonov navidezni računalnik. Ob naslednjem uvažanju bo Python preveril, ali je datoteka .pyc še mlajša od .py in v tem primeru uporabil kar že prevedeni .pyc, kar bo pospešilo uvoz. Za datoteke .pyc vam v resnici ni treba niti vedeti, edina potencialna težava je ta, da Python uvozi .pyc tudi, kadar .py sploh ne obstaja. Če torej želimo pobrisati nek modul (morda zato, ker smo ga prestavili v nek drug direktorij), je potrebno pobrisati tako .py kot tudi .pyc.

Kje Python išče module? To določa spremenljivka `sys.path`.

```
>>> import sys
>>> sys.path
['', 'C:\\Python25\\lib\\site-packages\\setuptools-0.6c6-
py2.5.egg',  'c:\\d\\ai\\orange',
'c:\\d\\ai\\orange\\orangecanvas',
'c:\\d\\ai\\orange\\orangewidgets',
'C:\\WINDOWS\\system32\\python25.zip', 'C:\\Python25\\DLLs',
'C:\\Python25\\lib', 'C:\\Python25\\lib\\plat-win',
'C:\\Python25\\Lib\\site-packages\\pythonwin', 'C:\\Python25',
'C:\\Python25\\lib\\site-packages', 'C:\\Python25\\lib\\site-
packages\\win32']
```

Ta pot ni povezana s sistemsko spremenljivko `PATH`.

Kako poskrbeti, da bo Python našel naš modul? Kako na pot dodamo še kak imenik? Spremenljivka `sys.path` je najobičajnejši seznam, torej lahko z `append`, `insert` ali kako drugače dodamo (ali, če želimo, odstranimo) kak imenik. A ta rešitev je neobičajna in jo uporabljajo – pa še to redko – le večje aplikacije, ki želijo po zagonu Pythona nastaviti poti do svojih modulov.

Najpreprosteje je modul dodati v direktorij *site-packages*. Na MS Windows je to običajno *c:\python25\lib\site-packages*, na Linuxu pa */usr/lib/python2.5/site-packages*. Številka v poti, 25 oziroma 2.5, je seveda odvisna od različice Pythona.

Sicer pa Python sestavi `sys.path` tako, da prebere imenike, naštete v okoljski spremenljivki `PYTHONPATH`, poleg tega pa prebere vse datoteke s končnico .pth v imeniku *site-packages*. V vsaki datoteki .pth je en ali več imenikov, v vsaki vrstici po eden. V MS Windows Python poleg tega doda v `sys.path` še poti iz določenih ključev v sistemskem registru, vendar to ni ravno svetovani in univerzalni način dodajanja poti do svojih modulov.

Izjeme in opozorila

Izjemo (*exception*) v Pythonu sprožamo z ukazom `raise`, ki mu sledi objekt, ki bo predstavljal izjemo. Včasih je bilo v modi za takšen objekt uporabiti kar niz z opisom napake (`raise "list index out of range"`), po novem pa mora to biti razred ali objekt razreda, izpeljanega iz razreda `Exception`. Tule je preprosta funkcija za računanje poprečij iz števil v seznamu.

```
def poprecje(l):
    if not l:
        raise ZeroDivisionError("seznam je prazen")
    return sum(l) / len(l)
```

Razred `ZeroDivisionError` prihaja iz modula `exceptions`, ki pa se že ob zagonu uvozi v globalni imenski prostor, zato ga ni potrebno posebej uvažati

Za lovljenje izjem uporabimo blok *try-except*.

```
def izpisiPoprecje(l):
    try:
        p = poprecje(l)
        print p
    except ZeroDivisionError:
        print "Ne bo slo, seznam je prazen"
```

Blok `except` v zgornji kodi lovi le napake tipa `ZeroDivisionError`. Če pričakujemo še kak drug tip napak, dodamo še `except` za druge tipe, ali pa, preprosto, izpustimo tip, s čimer polovimo vse (preostale) izjeme.

```
def izpisiPoprecje(l):
    try:
        p = poprecje(l)
        print p
    except ZeroDivisionError:
        print "Ne bo slo, seznam je prazen"
    except:
        print "Neznana napaka."
```

K tipu napake, ki jo lovimo, lahko dodamo še ime, ki mu bo prirejen objekt, ki predstavlja izjemo in lahko – predvsem, kadar razred izjeme definiramo sami –

vsebuje dodatne informacije o tem, kakšna napaka se je zgodila.

Znotraj bloka `except` smemo sprožiti novo izjemo, `raise` brez argumentov, pa ponovno sproži isto izjemo.

```
try:
    f()
except:
    print ”Nekaj izjemnega je letelo mimo”
    raise
```

Poleg `except` lahko k `try`, tako kot v Javi in C#, dodamo še blok `finally`, ki se izvede vedno, ne glede na to, ali je v bloku, ki ga ščiti `try`, prišlo do napake ali ne. V C++ bi takšen blok prišel zelo prav zaradi pospravljanja pomnilnika, v Pythonu pa se stvari pospravljajo same, zato bloka `finally` ne potrebujemo pogosto.

Pythonova posebnost je blok `else`. Tega postavimo med `except` in `finally`, izvede pa se, če v kodi znotraj `try` ni prišlo do izjeme. Na ta način je podoben `else` v zanki `for`, ki se izvede, če se zanka ne konča z `break`.

Izjeme vedno predstavljajo objekti razredov, izpeljanih iz razreda `Exception`. Obstoječi razredi za izjeme so urejeni v hierarhijo (`DivisionByZero` je izpeljan iz `ArithmeticError`, ta pa iz `Exception`). Poleg njih pa lahko definiramo še svoje in jih umestimo na primerno mesto v hierarhiji.

Poleg izjem Python pozna še opozorila (*warnings*). Ta so manj pomembna in niso tako tesno vdelana v jezik. Prožimo jih s funkcijo `warn` v modulu `warnings`. Opozorilo je sestavljeno iz niza in kategorije opozorila, predstavljene z objektom izpeljanim iz razreda `Warning`. Izpiše se na standardni izhod za napake. To lahko spremenimo s pomočjo filtrov, do katerih prav tako dostopamo s funkcijami iz modula `warnings`. Filtru opišemo sporočilo opozorila z regularnim izrazom, dodamo kategorijo opozorila, modul, ki ga sproži, in vrstico v modulu (kaj od tega smemo tudi izpustiti). Za tako opisano opozorilo lahko določimo, naj ga tolmač prezre, izpiše na izhod za napake ali pa ga spremeni v izjemo.

Opozorila so dokaj nepraktična, saj koristijo le programerju, ko dobi program v roke končni uporabnik, pa jih moramo bodisi izključevati bodisi spreminjati v izjeme, da jih lahko primerno obravnavamo. Ker oboje zahteva nastavljanje filtrov in drugo dodatno delo, se opozarjanja pretežno izogibamo.

Razredi

Tako kot je modul zbirka funkcij, pa še kako "globalno" spremenljivko ali morda razred ima lahko, tako je tudi razred zbirka metod, pa še kako spremenljivko ali razred (znotraj sebe) ali modul ali kaj drugega lahko definira. Pythonov razred je spet bolj ali manj samo imenski prostor.

Osnovna ideja

Razredi v Pythonu delujejo praktično brez birokracije. Ni privatnosti in skrivanja, ni protekcije, ni prijateljstev.[23] Sestavimo preprost primer, razred za nakupovalno košarico. Imel bo konstruktor, metodo za dodajanje artiklov v košarico in metodo, ki izračuna, koliko stane trenutna vsebina košare. Artikli bodo zbrani v seznamu `content`, ki bo vseboval pare *(artikel, količina)*.

```
class Basket:
    def __init__(self, owner = ””):
        self.owner = owner
        self.content = []

    def add(self, item, quantity):
        self.content.append((item, quantity))

    def total(self):
        tot = 0
        for item, quantity in self.content:
            tot += quantity * item.price
        return tot
```

Način, kako definiramo razred in metode, je konsistenten z vsem, kar smo videvali doslej: ključni besedi `class` sledi ime razreda, nato dvopičje in zamik. Vse, kar bomo pisali s tem zamikom, je del razreda.

---

23 Lepe navade ne morejo biti stvar prisile (le skozi okno poškilite na cesto, pa se boste prepričali).

Bralca to menda ni presenetilo. Bolj ga muči, kaj je `self`. Je to...? Da, to je ekvivalent `this` iz C++ in Jave. Objekt je potrebno eksplicitno navesti med argumenti in tudi ob uporabi znotraj metod. Tako bi v C++ prirejali "spremenljivki" `content`, ki bi v resnici pomenila `this->content`. V Pythonu to eksplicitno imenujemo `self.content`.

Spet mi boste morali verjeti na besedo: to moti le začetnike. V resnici je eksplicitni `self`, za začetek, *pregleden*, saj tako vedno vemo, kdaj govorimo o spremenljivki ali funkciji kar tako in kdaj o polju ali metodi objekta. Nadalje je *eleganten*, saj s tem tudi razred postane nekakšen imenski prostor. Končno je *nujen* zaradi drugih lastnosti Pythona. Ko bi ne bilo `self`a in bi konstruktor izvedel `content = []`, ne bi bilo jasno, ali v resnici mislimo `self.content` ali `content` kar tako. Do še večjih zapletov bi prišlo pri nevezanih metodah, na katere bo beseda še nanesla.[24]

Metoda, ki bi dodala v košarico več artiklov, dobila pa bi jih v seznamu podobnih parov, kot je `content`, bi bila lahko[25] takšna:

```
    def add_multiple(self, aList):
        for item, quantity in aList:
            self.add(item, quantity)
```

Pravilo za dostop do metod je torej enako kot za dostop do polj: tako kot do (lastnega) polja `content` pridemo s `self.content`, tudi lastno metodo `add` pokličemo s `self.add`. Logika je, ponovno v tem, da gre za metodo `add` objekta `self` (in ne katerega drugega objekta ali celo za kako globalno funkcijo `add`).

Poljem (*field*) v Pythonu navadno rečemo atributi (*attribute*). Predpostavljajoč, da je bralcu bolj domač C++, predvsem pa, ker bomo razrede v Pythonu pogosto primerjali z onimi iz C++, bomo občasno uporabljali en izraz, občasno drugega.

---

24 Ideja sama prihaja iz Module-3. Če moje mnenje kaj šteje: po dvajset letih programiranja v C++, ki se jim je v zadnjih desetih letih pridružil Python, mi v tem pogledu bolj všeč Python kot C++. Da v slednjem iz kode ni očitno, kdaj gre za spremenljivko in kdaj za polje v objektu, me je začelo prav motiti, zato bi najraje še tam striktno uporabljal `this`.

25 Privlačni možnosti sta še `for t in aList: self.add(*t)` in, seveda, `self.content.extend(aList)`

Naslednja nenavadnost Pythonovih razredov – ki je skladna s Pythonom in ki smo jo že nakazali zgoraj – je, da ni potrebno deklarirati polj, ki jih bodo imeli objekti. Zgoraj smo uporabljali `self.owner` in `self.content`, dasiravno nismo nikjer najavili, da bo imel razred ti polji, kaj šele povedali, kakšnega tipa bosta. Polja sme dodajati – vsakemu objektu posebej ali celotnemu razredu – kdorkoli, ne le metode razreda, ki mu objekt pripada, kot bi morda kdo pomislil na prvi pogled. Če je `b` košarica, smemo kjerkoli v programu napisati `b.x = 42`, pa bo imela košarica še polje `x` z vrednostjo `42`.

Na tem mestu se mnenja krešejo. Ni prosto dodajanje polj grda programerska praksa in znak slabo načrtovanega jezika? Slab programer bo vedno našel priložnost, da piše slabo kodo. Dobremu pa bo svoboda prišla prav, ne da bi se zaradi tega opekel. V resnici lahko vsak dela, kakor želi. Izbere lahko najstrožji pristop, v katerem se obnaša, kot da so vsa polja privatna in do njih dostopa le prek metod objekta. Lahko se drži načela, da vsa polja inicializira konstruktor, nova pa je prepovedano dodajati. Lahko pa dela po pameti in, če mu je primanjkuje, posledice pripiše sebi.

V funkciji za izračun cene je pozoren bralec opazil, da ta predpostavlja, da ima artikel polje `price`, saj uporabimo `item.price`. Opazil že, presenetiti pa ga ne bi smelo. S tem ni nič drugače kot s fibonaccijem, ki je predpostavljal, da je podane argumente mogoče seštevati. Če fibonacciju namesto dveh števil pomolimo dva modula, bo javil napako, podobno pa se bo zgodilo košarici, če bomo vanjo tlačili objekte brez cene.

Prav, sestavimo torej še razred za opisovanje artiklov.

```
class Item:
    def __init__(self, name, price):
        self.name = name
        self.price = price
```

Zdaj pa je že čas, da sestavimo košarico in jo napolnimo z dvajset bananami in dvanajst litri mleka.

```
>>> b = Basket()
>>> banane = Item("banane", 1.20)
>>> b.add(banane, 20)
>>> b.add(Item("mleko", 1.40), 12)
```

Objekt sestavimo tako, da pokličemo ustrezni razred. Malenkost poenostavljeno klic razreda pomeni klic konstruktorja. `Basket` smo poklicali brez argumentov, zato je lastnik košarice prazen niz (tak je privzeti argument konstruktorja). `Item` pa nima privzetih argumentov, temveč smo pri obeh navedli njuno ime in ceno. Banane smo sestavili že pred dodajanjem v košarico, mleko pa kar mimogrede, ob klicu. Tu torej vse teče praktično enako kot v C++.

Zdaj se lahko vprašamo, koliko nas bo vse skupaj stalo.

```
>>> print b.total()
40.8
```

In zdaj bi morda še enkrat pogledali v košarico...

```
>>> b.content
[(<__main__.Item instance at 0x012F9E68>, 20), (<__main__.Item
instance at 0x012F9E90>, 12)]
```

Tole pač ni videti posebej lepo in tudi nič ne pove, vendar se bomo s tem pozabavali kasneje.

Navidezne metode, dedovanje

Kot bi lahko pričakovali, so vse metode navidezne (*virtual*), abstraktnih razredov, ki bi vsebovali nedefinirane navidezne metode (*pure virtual method*), pa Python ne pozna. Metodo lahko vedno poskusimo poklicati. Če je ni, bo tolmač pač javil napako, ko in kjer bo treba.

Za začetek dodajmo v razred metodo, ki iz košarice odstrani več artiklov; kot argument dobi seznam artiklov, ki jih želimo odstraniti.

```
    def remove_multiple(self, aList):
        for item in aList:
            self.remove(item)
```

Če bi kaj podobnega storili v C++, bi se prevajalnik razjezil, da metode `remove` ni. V Pythonu se ne zgodi nič takega. Vse, kar smo z razredom počeli zgoraj, bi delovalo tudi s tako definiranim razredom, do katastrofe bi prišlo šele ob prvem klicu `remove_multiple`.

```
>>> b.remove_multiple([banane])
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "c:\d\basket.py", line 21, in remove_multiple
    self.remove(item)
AttributeError: Basket instance has no attribute 'remove'
```

Običajne trgovinske košarice (`Basket`) torej kupcem žal ne omogočajo vračanja izdelkov na police (ko se dotaknete jogurta, ga morate kupiti, tako kot morajo šahisti premakniti figuro, ki jo primejo). Obstajajo pa košarice (`EnhancedBasket`), pri katerih je to dovoljeno, razvite pa so iz osnovne košarice (`Basket`).

```
class EnhancedBasket(Basket):
    def remove(self, item):
        for art in reversed(self.content):
            if art[0] == item:
                self.content.remove(art)
```

Spreglejmo neposrečenost kode funkcije. Bralec naj bo, najprej, pozoren na to, kako smo povedali, da je razred `EnhancedBasket` izpeljan iz `Basket` – z oklepajem za imenom razreda `EnhancedBasket`. Python podpira tudi večkratno dedovanje; v tem primeru prednike razreda naštejemo v oklepaju in jih ločimo z vejicami.

Izpeljani razred, `EnhancedBasket`, kot je v navadi, podeduje vse metode prednika `Basket`, lahko pa katero doda ali spremeni. V našem primeru smo le dodali metodo `remove`. Če košarico sestavimo kot objekt razreda `EnhancedBasket`, bo zato začela delovati tudi prej nedelujoča metoda `remove_multiple`.

```
b = EnhancedBasket()
banane = Item("banane", 1.20)
b.add(banane, 20)
b.add(Item("mleko", 1.40), 12)
print b.total()

b.remove_multiple([banane])
print b.total()
```

Tole bo najprej izpisalo `40.8` in nato `16.8`.

Z vidika C++ je takšno delovanje metod čudno. Kako `remove_multiple` ve, katero metodo poklicati? Saj se ta pojavi šele pri naslednikih! Preprosto. Argument `self` v `remove_multiple` je v zadnjem primeru tipa `EnhancedBasket`. Ko `remove_multiple` poskusi poklicati `remove`, preveri, ali objekti tipa `EnhancedBasket` imajo metodo takšnega imena, in ko vidi, da jo imajo, jo zadovoljno žvižgaje pokliče.

V `EnhancedBasket` dodajmo še diagnostični izpis: ob vsakem dodajanju naj najprej pove, kaj smo dodali in za koliko bo to povečalo vrednost košarice, nato pa kliče podedovano metodo za dodajanje.

```
    def add(self, item, quantity):
        print "izdelek '%s', kolicina %s: cena %.2f" % \
          (item.name, quantity, item.price*quantity)
        Basket.add(self, item, quantity)
```

Kaj se zgodi, ko pokličemo `add_multiple`, ki je definirana v `Basket`? Kateri `add` kliče? Onega iz `Basket` ali tistega iz `EnhancedBasket`? V C++ je to odvisno od tega, ali je `add` definiran kot virtualna metoda ali ne. V Pythonu pa kliče, kar smo naročili, namreč `self.add`. Če je `self` tipa `EnhancedBasket`, bo poklicala metodo `add` razreda `EnhancedBasket`. Metoda `add` (in vse druge metode v Pythonu) se vedejo, kot bi se v C++, če bi bile deklarirane kot navidezne (*virtual*).

Tudi klic podedovane metode je podoben onemu v C++, le namesto dvojnega dvopičja imamo kar piko, kot je v navadi za Pythonove imenske prostore (npr. module). Nekoliko zmedeno deluje le `self`. Čemu ga je pri takem klicu metode potrebno dodajati? Odgovor na to vprašanje je vreden svojega razdelka.

Vezane in nevezane metode

C pozna kazalce na funkcije: če želimo funkcijo podati kot argument (ali pa jo vrniti kot rezultat, ali pa shraniti v neko strukturo...) to storimo tako, da uporabimo kazalec nanjo. Kot smo videli, je v Pythonu funkcija objekt kot katerikoli drugi. Funkcijo lahko podamo kot argument, jo vrnemo kot rezultat ali porinemo v seznam ali množico, kot bi to počeli s katerimkoli drugim objektom, recimo številom ali nizom.

Kako pa se v tem pogledu vedejo metode? C++ pozna kazalce na metode, ki jih spremlja nebeško lepi operator `::*`. (Sarkazem, seveda. Konstrukta `::*` si ne bi mogel domisliti niti Kosovel, tega se lahko spomni samo kdo, ki se sozemljanom predstavlja kot Stroupstroup ali kaj podobnega). Pa Python?

Kot vse drugo je tudi metoda objekt.

```
>>> b.add
<bound method EnhancedBasket.add of <__main__.EnhancedBasket
instance at 0x012FBAF8>>
>>> type(b.add)
<type 'instancemethod'>
```

Tole nam pove, da je `b.add` objekt tipa `instancemethod`, sicer pa gre za vezano metodo z imenom `EnhancedBasket.add`, nanaša pa se na objekt `<__main__.EnhancedBasket instance at 0x012FBAF8>`. Slednji seveda ni nihče drug kot naša košarica, kot se lahko nemudoma prepričamo.

```
>>> b
<__main__.EnhancedBasket instance at 0x012FBAF8>
```

Ker je `b.add` objekt kot katerikoli drugi in ker lahko objekte shranjujemo v spremenljivke (oziroma jim, kot bi se natančneje izrazili, dajemo imena), lahko to storimo tudi z njim.

```
>>> plenice = Item("plenice", 3.80)
>>> dodaj_v_b = b.add
>>> dodaj_v_b(plenice, 4)
```

Spodnji klic deluje, ker sta `dodaj_v_b` in `b.add` en in isti objekt, namreč metoda `add`.

```
>>> dodaj_v_b
<bound method EnhancedBasket.add of <__main__.EnhancedBasket
instance at 0x012FBAF8>>
```

Kaj pomeni, da je `dodaj_v_b` (oziroma `b.add`) *vezana* metoda? Vezana ... na kaj? Na objekt `b`. Ko pokličemo `b.add`, dodajamo, seveda, v košarico `b`. Enako pa, kajpa, ko pokličemo `dodaj_v_b`, ki je ena in ista reč. Kako pa so videti nevezane metode?

Po prejšnjem razdelku bi morali že uganiti, ne?

```
>>> EnhancedBasket.add
<unbound method EnhancedBasket.add>
```

Tudi takšna metoda je objekt, torej ga lahko, na primer, prirejamo drugim spremenljivkam.

```
>>> dodaj_v = EnhancedBasket.add
>>> dodaj_v
<unbound method EnhancedBasket.add>
```

Ga lahko tudi pokličemo? Da, vendar...

```
>>> EnhancedBasket.add(plenice, 4)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: unbound method add() must be called with
EnhancedBasket instance as first argument (got Item instance
instead)
```

`EnhancedBasket` ni objekt, temveč razred, zato `add` ne ve, v katero košarico naj doda te štiri pakete pampersk. Zato ga lahko, kot pove tudi sporočilo o napaki, pokličemo le tako, da mu priskrbimo košarico.

```
>>> EnhancedBasket.add(b, Item("plenice", 3.80), 4))
```

Tako je postal razumljiv tudi nenavadni klic s konca prejšnjega razdelka,

```
Basket.add(self, item, quantity)
```

Imeli smo objekt `self`, poklicati pa je bilo potrebno podedovano metodo `add`. Klic `self.add` bi vodil v neskončno rekurzijo, zato je bilo potrebno (enako, kot bi storili v drugih jezikih) posebej povedati razred, katerega metodo kličemo.[26] Ker pa je `Basket.add` nevezana metoda, mora dobiti kot argument še objekt, nad katerim naj deluje, torej `self`.

---

26 V resnici to od Pythona 2.2 ni več potrebno, niti ne priporočeno. Gornji klic naj bi pisali takole: `super(EnhancedBasket, b).add(item, quantity)`, vendar bi zahteval tudi nekoliko drugačno definiranje razredov. V to smer v tej knjigi ne bomo rinili, radovednejši bralec pa lahko prebere besedilo Guida van Rossuma "Unifying types and classes in Python 2.2" (`http://www.python.org/download/releases/2.2.3/descrintro/`)

Čemu obstajajo vezane in nevezane metode? Razloga sta dva. Ker je vse objekt, mora tudi metoda biti objekt. Obrnjeno drugače, ko smo zapisali `b.add(item, quantity)`, smo, formalno gledano, uporabili operator `()`. V zgledno predmetno usmerjenem jeziku operator ne more biti operator kar tako, temveč mora pripadati nekemu objektu – v tem primeru očitno *objektu* `b.add`. Že zaradi tega metoda mora biti objekt.

Drugi razlog je, da je to praktično. Vedno je koristno imeti možnost shraniti metodo v spremenljivko, ali jo poslati kot argument. Vsaj Python to interno redno počne.

Pravzaprav je razlog še eden. O njem govori naslednji razdelek.

Razredne spremenljivke

Večkrat smo se že pohvalili, da je v Pythonu tudi razred objekt. Torej: tudi `Basket` je objekt. Objektom, kot zdaj vemo, lahko prosto dodajamo nova polja: `b.x = 42` objektu `b` doda polje `x` z vrednostjo 42. Če je `Basket` objekt in če objektu lahko dodajamo polja, to pomeni, da lahko dodajamo polja tudi objektu `Basket`? Takšne reči je najboljše poskusiti.

```
>>> Basket.zz=42
```

Napake ni javilo, torej je videti, da deluje. Deluje že, kaj pa naredi? Kdo ve za `zz`? Kaj lahko počnemo z njim? Zanj ve, očitno, razred `Basket`, poleg tega pa tudi vsi objekti tega razreda – vključno s tistimi, ki so bili konstruirani, še preden smo razredu dodali `zz`.

Prvi primer rabe je klasičen. Vsaki košarici želimo dodeliti identifikacijsko številko. Lahko bi storili tole.

```
class Basket(object):
    bid = 0

    def __init__(self, owner = ""):
        Basket.bid += 1
        self.id = Basket.bid
```

```
        self.owner = owner
        self.content = []
```

Zdaj je `Basket.bid` razredna spremenljivka, nekaj takšnega kot statično polje razreda (*static class member*) v C++. V inicializaciji na začetku, `bid = 0`, smo razred izpustili, saj se vse, kar je zamaknjeno znotraj `class`, nanaša na razred. (Bralec naj poskusi namesto tega pisati `Basket.bid = 0`, pa ne bo delovalo. Če bi rad odkril, zakaj ne, pa naj zapre Python, ga ponovno odpre in še enkrat požene skripto. Potem bo razumel.)

Če zaženemo gornje primere na tako definiranem razredu, dobi vsak objekt še unikatno polje `id`. Poleg `b.id` bo obstajal tudi `b.bid`, ki seveda pomeni oni, edinstveni, razredni `bid`.

Narediti bi smeli tudi takole.

```
class Basket(object):
    id = 0

    def __init__(self, owner = ""):
        Basket.id += 1
        self.id = Basket.id
        self.owner = owner
        self.content = []
```

Ker pred `id` vedno uporabljamo objekt (`self` oz. `Basket`), ni potrebe po različnih imenih. Poleg tega objektov `id` zdaj "zasenči" razredov `id` (če želi do njega, mora eksplicitno povedati, da hoče `Basket.id`), kar je prav tako dobro.

Je bralec ob tem doživel kako malo razsvetljenje? So se kaki koščki sestavljanke začeli ujemati? Še ne? Takole: če v programu (torej, ne v razredu, temveč v skripti kar tako) rečemo `id = 0` in če v programu rečemo `def add(item, quantity):` (in tako naprej), bo imel program spremenljivko `id` in funkcijo `add`. Če to naredimo v modulu, bo imel modul spremenljivko `id` in funkcijo `add`. Če pa to naredimo v razredu, bo imel spremenljivko `id` in funkcijo `add` razred. Ker objekt vidi vse, kar je v razredu, vidi tudi razredne spremenljivke in funkcije – ki jim v tem kontekstu pravimo metode. In, z druge strani, metode so najobičajnejše razredne spremenljivke. Se razumemo? Tako kot je `id` razredna spremenljivka tipa

`int`, je `add` razredna spremenljivka tipa `unbound method`.[27] Tako kot je modul nekakšen imenski prostor, je tudi razred le nekakšen (in nekoliko poseben) imenski prostor.

Na koncu prejšnjega razdelka smo omenili tretji razlog, zakaj morajo biti metode objekti. Zdaj je jasno, ne? Razred ni nič drugega kot zbirka različnih objektov, nekateri so tipa `int` (recimo `id`), nekateri, recimo `add`, pa metode. Ko bi metode ne bile objekti, na kateri objekt iz razredovega imenskega prostora bi se nanašalo ime `self.add`?!

Hm, hm. Ko smo zgoraj rekli `self.id = Basket.id`, je objektov `id` "zasenčil" razrednega. Lahko na podoben način povozimo tudi metodo?

```
def f(item, quantity):
    print ”Danes bo izdelek ’%s’ ostal v trgovini...” %
item.name

b.add = f
```

Tako je. V košarico `b` zdaj ni več mogoče dati ničesar. Niti z metodo `add_multiple` ne, saj ta kliče `self.add` in `self.add` (oziroma `b.add`) je zdaj `f`, ta pa zavrača vsakršen nakup.

```
>>> b.add(banane, 20)
Danes bo izdelek ’banane’ ostal v trgovini...
>>> b.add_multiple([(banane, 20)])
Danes bo izdelek ’banane’ ostal v trgovini...
```

Ob takšnem programiranju seveda upravičeno pogodrnjamo. V opravičilo jeziku povejmo, da to dopušča ponesreči. Pač, da se, zato ker je takšna logika razredov. To uporabljati, po možnosti celo brez utemeljenega razloga, pa ne bi bilo lepo.

Posebne metode

V dosedanjih zgledih se nismo posebej spotikali ob konstruktor. Poimenovali smo ga `__init__` in tisti bralec, ki je predpostavil, da mora biti tako pač ime

27 Eleganco nekoliko ruši le vezava metod.

konstruktorju, je predpostavil popolnoma pravilno. Ko konstruiramo objekt, denimo s klicem `b = Basket()`, Python preveri, ali ima ta razred definirano metodo `__init__`. Če jo ima, jo pokliče, sicer sestavi objekt kar brez nje, kot da bi bil `__init__` prazen.

Podobnih metod s posebnimi imeni je še veliko. Poskrbimo, najprej, za lepši izpis naše košarice. V `EnhancedBasket` dodajmo naslednjo metodo.

```
    def __str__(self):
        names = []
        for item, quantity in self.content:
            names.append("%s (%s)" % (item.name, quantity))
        return "<" + ", ".join(names) + ">"
```

Iz seznama artiklov najprej naredimo seznam nizov, sestavljenih iz imena artikla in, v oklepaju, količine. Nato seznam združimo z vejicami in zapremo v oklepaje. (Tako zapisana koda je okorna. Nekoč bomo vedeli, da jo lahko zapišemo veliko preprosteje in učinkoviteje.)

```
>>> print b
<banane (20), mleko (12)>
```

Za ta, lepi izpis moramo uporabiti `print`. Če ga izpustimo, se košarica izpiše tako kot prej.

```
>>> b
<__main__.EnhancedBasket instance at 0x01309EE0>
```

Razlog je v tem, da obstajata dve metodi za izpis objekta. Ena, `__str__`, pokaže uporabniku prijaznejši izpis, druga, `__repr__` pa bolj tehničnega. Ukaz `print` uporabi prvega, lupine pa ob izpisovanju objektov navadno pokličejo drugega.

Seveda lahko na enak način, kot smo definirali `__str__`, definiramo še `__repr__`. Vendar je enostavneje, če v definicijo razreda dodamo kar `__repr__ = __str__`. Podobno kot smo v prejšnjem razdelku (zgražanja vredno) povozili metodo `add` nekega objekta, tako za ves razred povozimo metodo `__repr__`, da se nanaša na natanko isto funkcijo kot `__str__`. Takih reči sicer ne počnemo pogosto, še največkrat takrat, ko kako funkcijo preimenujemo, a ji zavoljo združljivosti pustimo tudi staro ime.

Za vajo sestavimo še eno metodo: takšno, s katero bomo lahko ugotovili, ali smo

kupili določen izdelek, na primer, banane. Natančneje, želeli bi si pisati pogoje v takšni obliki:

```
if not "banane" in b:
    print "Mar bomo danes kar brez banan?"
```

Tako, kot je objekt definiran zdaj, gornje seveda ne bi delovalo.

```
>>> "banane" in b
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: argument of type 'EnhancedBasket' is not iterable
```

Sporočilo o napaki je sicer nekoliko kriptično in ga za zdaj še ne razumemo, v načelu pa pravi, da je Python poskušal iti prek elementov `b` in jih primerjati z nizom `"banane"`, a tega ni mogel storiti, ker ni metode, s katero bi lahko šel prek elementov `b`.

Košarici bi lahko sicer dodali metodo za prehod prek artiklov, a tudi to nam ne bi pomagalo, saj naši artikli niso opisani z nizi in jih ni tudi ni mogoče kar tako primerjati z njimi. Najpreprostejša pot je definirati metodo `__contains__`.

```
    def __contains__(self, s):
        for item, quantity in self.content:
            if item.name == s:
                return True
```

Kako metoda deluje, je očitno in kaj mora vrniti, je bralec uganil. Pa če iskanega artikla ni v košarici, zakaj tedaj ne vrnemo `False`? Ker nam ga ni treba. Če ne vrnemo ničesar, vrnemo `None` in `None` je neresničen, torej ima, da ne vrnemo ničesar, enak učinek, kot če bi vrnili `False`.

Še zadnja metoda: želeli bi, da `len(b)` vrne število elementov v košarici. Trivialno.

```
    def __len__(self):
        return len(self.content)
```

V knjigi se izogibamo tabelam in seznamom, ki sodijo v bolj tehnično dokumentacijo, tokrat pa naredimo izjemo in naštejmo vse (ali vsaj večino) imen posebnih metod.

`__new__(cls[, ...])`, `__init__(self[, ...])`, `__del__(self)`

Konstruktor, inicializator in destruktor. Ob sestavljanju objekta se najprej pokliče prvi, ki zanj rezervira pomnilnik in postavi nujna polja, `__init__` pa konča inicializacijo. Pri programiranju v Pythonu skoraj vedno definiramo `__init__`, `__new__` pa je bolj pomemben pri pisanju razširitev v Cju. Kadar `__new__` definiramo v Pythonu, bomo vedno poklicali podedovani `__new__`, da bo rezerviral pomnilnik za objekt (v čistem Pythonu to očitno ni mogoče). Podobno je z `__del__`, ki se kliče, ko se objekt poslavlja. V Pythonu ga sem ter tja definiramo, da pospravimo kako malenkost, vedno pa moramo poklicati podedovani `__del__`, da sprosti pomnilnik. Obenem ne pozabimo, da metoda `__del__` nima veliko skupnega z operatorjem `del`. Ta namreč le pobriše eno od imen za objekt, medtem ko se `__del__` pokliče, ko za objekt ne ve nihče več.

`__repr__(self)`, `__str__(self)`

Ta par smo že spoznali: metodi s tema imenoma morata vrniti niz, ki predstavlja objekt. Zaželeno je, da prvi, če je le mogoče, vrne veljaven Pythonov izraz – torej niz, ki bi ga lahko skopirali v program in bi sestavil natanko enak objekt. `__str__` naj bi izpisal kaj prijaznejšega. V tem pogledu smo dobro definirali `__str__`, `__repr__` pa bi moral vrniti niz v slogu `[Item("banane", 1.20)]`.

`__lt__(self, other)`, `__le__(self, other)`, `__eq__(self, other)`, `__ne__(self, other)`, `__ge__(self, other)`, `__gt__(self, other)`

Operatorji za različne enakosti in neenakosti (*less-than*, *less-or-equal*, ...). Vrnejo lahko `True` ali `False`, ali pa tudi kaj drugega, podobno kot smo pri definiciji `__contains__` vrnili `None`, ki ga Python razume kot neresnično. Med seboj so povsem neodvisni – če dva objekta nista različna, to še ne pomeni, da sta enaka.

`__cmp__(self, other)`

Starejša različica metode za primerjanje. Vrniti mora `-1`, če je prvi objekt manjši, `1`, če je večji in `0`, če sta enaka. Kadar sta definirana tako

`__cmp__` kot ustrezna gornja metoda, se kliče gornja metoda.

`__hash__(self)`

Razpršitvena funkcija, ki jo objekt potrebuje, da je lahko shranjen v slovarjih in množicah. Rezultat mora biti tipa `int` ali `long`. Ko smo rekli, da slovarji in množice lahko vsebujejo le nespremenljive objekte, smo malo poenostavili: v resnici lahko vsebujejo poljubne objekte, ki imajo definirano metodo `__hash__`. V praksi to navadno pomeni nespremenljive objekte, saj se od `__hash__` zahteva, da za isti objekt vedno vrne isto vrednost. Dodatna zahteva je, da pri paru objektov, ki sta enaka (torej, pri katerih bi `__cmp__(a, b)` vrnil `0` oziroma `__eq__(a, b)` vrnil `True`) tudi `__hash__` vrne enako vrednost.

`__nonzero__(self)`

Metoda, ki določa, ali je objekt resničen ali ne. Vrniti mora `True` ali `False` oziroma `1` ali `0`. Če bi želeli, naj bo naša košarica neresnična, ko je prazna, bi vrnili `False`, ko je dolžina `self.content` enaka `0` in `True` sicer. Če razred te metode ne definira, Python namesto nje uporabi `__len__`; objekt je neresničen, če `__len__` vrne `0`. Če razred ne definira ne `__nonzero__` ne `__len__`, so vsi objekti tega razreda resnični.

`__getattr__(self, name), __setattr__(self, name, value), __delattr__(self, name)`

Metode za branje in postavljanje vrednosti atributov. Ko napišemo `b.id`, Python dobi želeno vrednost tako, da pokliče `b.__getattr__("id")`. Podobno `b.id = 30` pokliče `b.__setattr__("id", 30)`. Metoda `b.__delattr__("id")` se pokliče ob brisanju atributa (`del b.id` – tega doslej nismo posebej opisovali, ker se nam ni zdelo posebej pomembno). Zanimivo je prirejanje `b.x.y.z = 3`, ki se izvede kot `b.__getattr__("x").__getattr__("y").__setattr__("z", 3)`.

Razredi novega tipa poznajo poleg teh treh metod še drugačen mehanizem za nastavljanje in branje vrednosti atributov, deskriptorje, ki pa jih bomo prezrli.

`__getitem__(self, key)`, `__setitem__(self, key, value)`, `__delitem__(self, key)`

Podobna reč, a nad seznami: te tri metode definirajo operator []. Prva se kliče ob branju (`l[42]` se izvede tako, da Python pokliče `l.__getitem__(42)` in `d["Miran"]` tako, da pokliče `d.__getitem__("Miran")`), druga ob prirejanju (`l[42] = "x"` se izvede kot `l.__setitem__(42, "x")`) in zadnja ob brisanju (`del l[42]` pomeni `l.__delitem__(42)`). Metode z istim imenom se uporabljajo za razrede, ki se obnašajo podobno kot seznami, terke in nizi ter za razrede podobne slovarjem. Razlika je le v tem, da prvi sprejemajo argumente tipov `int` in `SliceType` (ta se uporabi ob indeksiranju z rezinami, vsebuje pa polja `start`, `step` in `stop`), drugi pa objekte poljubnih tipov.

`__iter__(self)`

Operator, ki vrne iterator prek objekta. Posvečen jim bo poseben razdelek, za zdaj le nakažimo: `__iter__` je osnova za delovanje zanke `for`. Zanko `for` lahko poženemo nad objekti, ki imajo definirano metodo `__iter__`. Pri tistih, ki je nimajo, si bo Python poskušal pomagati z `__getitem__`, če ni na voljo niti ta, pa prek objekta ne bo mogel iterirati.

`__contains__(self, obj)`

Metoda, ki se skriva za operatorjem `in`. Kako jo definiramo, smo že videli. Kot vemo zdaj, si bo Python, če `__contains__` ne obstaja, poskušal pomagati z `__iter__`, če ni niti tega, z `__getitem__`, v skrajnem primeru pa bo obupano javil napako.

`__call__(self[, ...])`

Operator za klic. Poklicati je mogoče tiste objekte, pri katerih je definiran ta operator – med njimi so očitno vse funkcije, metode in, zanimivo, *razredi*. Seveda, ko smo napisali `Basket()`, smo poklicali razred, torej mora imeti tudi razred definiran `__call__`. (A kaj dela? I, no, objekt tega razreda sestavi in ga vrne, ne?)

`__add__(self, other), __radd__(self, other), __iadd__(self, other)`

Operatorji za seštevanje. Izraz `a+b` se izračuna s klicem `a.__add__(b)`. Če ta javi napako (torej: `a`-jev `__add__` ne zna sešteti `a` in `b`, zato javi napako `NotImplemented`), Python poskusi srečo pri `b`-ju, tako da pokliče `b.__radd__(a)`. Metoda `__radd__` se vedno pokliče za desni operand, da se ga vpraša, ali lahko prišteje levega. Pri komutativnih operacijah, na primer seštevanju števil, sta `__add__` in `__radd__` enaka, pri nekomutativnih, kot je seštevanje nizov, pa je razlika očitno pomembna. Če tudi to ne uspe, objektov ni mogoče sešteti.

`__iadd__` je operator, ki se uporablja pri prištevanju, torej `a += b`. Definirati ga smejo le spremenljivi razredi. Pri nespremenljivih, kot sta niz ali število, Python opazi, da `__iadd__` manjka, zato namesto njega kliče `__add__` oz. `__radd__`, rezultat katerega pa je seveda nov objekt.

`__sub__, __rsub__, __isub__, ..., __mod__, __rmod__, __imod__, ..., __or__, __ror__, __ior__, ...`

Operatorji za vse ostale aritmetične in logične operacije. Ker jih je precej, ne bomo našteli vseh, bralec bo med programiranjem tako ali tako raje škilil na splet kot v knjigo.

Kot zanimivost omenimo le operator za ostanek po deljenju, `__mod__`, ki ni definiran le nad števili temveč, nenavadno, tudi nizi. Kot argument sprejme niz na levi (očitno) in poljuben objekt, pogosto terko, na desni. Zveni nenavadno? Kako izračunamo ostanek po deljenju niza s čemerkoli že? Kdo pa pravi, da operator % *vedno* računa ostanek po deljenju? Metoda `__mod__` nad nizi se pokliče pri oblikovanju nizov, torej v izrazih kot je `"%s (%.3f)" % (item, quantity)`.

`__neg__, __pos__, __abs__, __invert__, __int__, __float__, __hex__`

Operatorji za razne druge številske pretvorbe. Bralec bo spet sam poiskal detajle, če jih bo potreboval.

Izpeljevanje iz vdelanih razredov

O dedovanju nismo govorili posebej veliko, saj o njem niti ni kaj posebej veliko govoriti: izpeljani razred pač podeduje metode svojih prednikov. Do zoprnih podrobnosti pride pri večkratnem dedovanju. Ker se avtorju knjige to zdi slaba praksa, se vanje ne bo spuščal.

Zanimivost Pythona (sicer ne edinstvena, vendar kar redka med splošno uporabljanimi jeziki) je, da lahko izpeljujemo nove razrede tudi iz vdelanih razredov, kot sta `int` ali `list`. Ne iz čisto vseh – iz razreda `types.FunctionType`, ki mu pripadajo funkcije, že ne moremo izpeljevati – iz teh, pri katerih je to smiselno, pa.

Tip `bool` je, kot že vemo, izpeljan iz `int`. Približno tako:

```
class bool(int):
    def __str__(self):
        if self==0:
            return "False"
        elif self==1:
            return "True"
        else:
            return int.__str__(self)

    __repr__ = __str__

True = bool(0)
False = bool(1)
```

Kako? Razred `bool` je enak `int`, samo z drugačnim izpisom za `0` in `1`? Natanko tako.

```
>>> True+2
3
```

Ker se `int` že od začetka obnaša enako kot `bool`, ko so tip dodali v Python, je bilo potrebno le popraviti izpis in dodati konstanti `True` in `False`, kot smo to storili v gornji kodi. V resnici je razred definiran v Cju, vendar bi tudi gornja rešitev v Python delovala enako.

Za vajo popravimo še našo košarico tako, da bo izpeljana kar iz seznama, `list`, namesto da vsebino košarice shranjuje v `self.content`. Sprememb je bore malo. Konstruktor mora poklicati podedovani konstruktor, `list.__init__(self)`. Vse `self.content` v programu nato zamenjamo s `self`. Poleg tega odstranimo metodo `__len__` saj ni več potrebna – ona, ki smo jo podedovali, je natanko enaka. Pisati

```
def __len__(self):
    return len(self)
```

bi bilo zelo narobe, saj bi metoda klicala samo sebe, pisati

```
def __len__(self):
    return list.__len__(self)
```

pa nesmiselno, saj bi tako metoda le poklicala podedovano metodo. Opustili bi lahko tudi metodi `add` in `add_multiple`, saj se ne razlikujeta bistveno od `append` in `extend`.

Takšna je celotna koda razreda:

```
class Basket(list):
    bid = 0

    def __init__(self, owner = ""):
        Basket.bid += 1
        self.id = Basket.bid
        self.owner = owner
        list.__init__(self)

    def __str__(self):
        names = []
        for item, quantity in self:
            names.append("%s (%s)" % (item.name, quantity))
        return "[" + ", ".join(names) + "]"

    __repr__ = __str__

    def add(self, item, quantity):
        self.append((item, quantity))

    def total(self):
        tot = 0
```

```
        for item, quantity in self:
            tot += quantity * item.price
        return tot

    def add_multiple(self, aList):
        for item, quantity in aList:
            self.add(item, quantity)

    def remove_multiple(self, aList):
        for item in aList:
            self.remove(item)

    def __contains__(self, s):
        for item, quantity in self:
            if item.name == s:
                return True
```

Konstruktorji in pretvarjanje tipov

Pretvarjanje med tipi (*type casting*) dosežemo s klicem konstruktorja.

Kot smo videli, nov objekt ustvarimo tako, da "pokličemo" razred, na primer,

```
Item("plenice", 3.80)
```

Sintaksa je sicer videti podobna kot v C++ in drugih objektnih jezikih, ozadje pa povsem drugačno. Razred `Item` je v resnici objekt, ki ima definiran operator `__call__`, ki poskrbi, da se ustvari nov objekt. To stori tako, da pokliče najprej `__new__` in nato `__init__`, vmes pa po potrebi še malo manevrira z argumenti.[28,29]

28 Bralec naj ne spregleda razlike: ko pokličemo `b(7)`, kjer je `b` objekt razreda `Basket`, za klic poskrbi metoda `type(b).__call__`, se pravi `Basket.__call__` (ki je ni, zato klic ne uspe). Ko kličemo `Basket()`, pa se pokliče `type(Basket).__call__`, se pravi `type.__call__`. Tip objekta `Basket` je namreč `type`. Metoda `type.__call__` pa je vdelana metoda, ki kliče `__new__` in `__init__`, kot smo opisali.

29 Prejšnja opomba velja le za Python 3.0, v starejših različicah pa samo za razrede novega sloga. V razredih starega sloga `type(Basket)` ni `type`, v grobem pa je mehanizem vseeno praktično enak.

Konstruktor razreda lahko sprejema argumente različnih tipov. Vzemimo cela števila. Cela števila so tipa `int`, torej nova števila sestavljamo tako, da kličemo tip, `int`. Kot argument lahko podamo število (celo ali ne) ali niz, rezultat pa bo vedno celo število.

```
>>> int(42)
42
>>> int(3.14)
3
>>> int("2")
2
```

Kadar podamo niz, lahko kot dodatni argument povemo še številsko osnovo, ki jo uporabi namesto privzete desetiške.

```
>>> int("2a", 16)
42
>>> int("101010", 2)
42
>>> int("60", 7)
42
```

Drug vdelan tip s privlačnim konstruktorjem je `list`. Ta kot argument prejme poljuben razred, po katerem je mogoče iterirati, ga "preiterira" in vse element zloži v novi seznam. Tule je nekaj primerov.

```
>>> t = [1, 2, 3]
>>> list(t)
[1, 2, 3]
>>> list("Miran")
['M', 'i', 'r', 'a', 'n']
>>> list((1, 2, 3))
[1, 2, 3]
>>> s = set([1, 2, 3])
>>> list(s)
[1, 2, 3]
```

Podobno se vede tudi množica. Kot argument smo ji doslej vedno podtikali le sezname, v resnici pa ji lahko damo tudi kaj drugega.

```
>>> set("tudi kaj drugega")
set(['a', ' ', 'e', 'd', 'g', 'i', 'k', 'j', 'r', 'u', 't'])
```

Tole se da tako lepo nadaljevati, da bi bilo škoda, če ne bi.

```
>>> sorted(list(set("tudi kaj drugega")))
[' ', 'a', 'd', 'e', 'g', 'i', 'j', 'k', 'r', 't', 'u']
>>> "".join(sorted(list(set("tudi kaj drugega"))))
' adegijkrtu'
```

Funkcijo `sorted` bomo resneje pogledali kasneje, za zdaj povejmo le, da ji podamo reči, ki jih je mogoče urediti, vrne pa urejen seznam teh reči.

Razredi kot objekti

Tako kot definicija funkcije priredi kodo imenu, tako tudi definicija razreda samo priredi objekt, ki predstavlja razred, imenu razreda. Za ilustracijo si najprej oglejmo tole čudno (in neuporabno) kodo.

```
for i in range(5):
    class A:
        def f(self):
            print i

    a = A()
    a.f()
```

Program izpiše številke od `0` do `4` (kar bi se dalo seveda doseči tudi na manj sofisticiran način). Ker je definicija razreda v zanki, smo petkrat definirali razred `A`, vendar vsakič na novo, z novim `i`. Še bolj nenavadno je tole.

```
l = []
for i in range(5):
    class A:
        def f(self, i=i):
            print i
    l.append(A())

for e in l:
    e.f()
```

Po zanki seznam `l` vsebuje pet objektov, vsi so objekti razreda, ki se je nekoč imenoval `A`. Ko je zanke konec, sicer obstaja samo en razred z imenom `A`, oni, zadnji, ostali so pač brez imena, a še vedno živi, ker obstajajo objekti tega razreda.

In zanka spet izpiše števila od `0` do `4`.

Torej: tako kot so spremenljivke v Pythonu za razliko od spremenljivk v Cju samo imena za objekte, je tudi gornji `A` samo ime za razred. Tako kot gre "običajna" spremenljivka (točneje, objekt) v smeti, ko nihče več ne ve zanjo, gre tudi razred v smeti, ko nima več ne imena ne objektov, ki bi mu pripadali. Če bi gornji program nadaljevali z `del l`, bi z objekti v tem seznamu izginili tudi njihovi razredi, preživel bi le zadnji, saj je znan pod imenom `A`, poleg tega pa bi bil `e` še vedno tudi objekt tega razreda.

`A` je, na nek način, spremenljivka, namreč spremenljivka tipa razred.

Tile primeri niso praktično uporabni, ogledali smo si jih zgolj, da bi bralec razumel mehaniko Pythonovih razredov.

Vsakdanji primer, ko pride tako znanje prav, je tale. Sprogramiramo razred, nato v ukazni vrstici sestavimo objekt tega razreda, preskusimo neko metodo, vidimo, da ne dela, jo popravimo in ponovno poženemo skripto z definicijo razreda. Tako sicer dobimo novo definicijo tega razreda,[30] a objekt, ki ga imamo v ukazni vrstici, je še vedno objekt starega razreda, onega, z metodo, ki ne deluje. Da preskusimo popravljeni razred, je potrebno sestaviti nov objekt.

Čemu skrivnostni `i=i` v definiciji metode `f`? S tem smo naredili kopijo spremenljivke-objekta `i`. Poskusite brez tega, pa boste videli, kaj se zgodi.

Razredi novega sloga

Do Pythona 2.2 so bili razredi, ki smo jih s `class` definirali v Pythonu, in tipi, ki so definirani v Cju (vsi vdelani tipi, npr. `int` in `dict`, pa tudi vsi tipi, ki jih definirajo dodatni moduli v Cju) ločeni. Nekoliko različno so se vedli, predvsem pa iz tipov v Cju ni bilo mogoče izpeljevati novih razredov. Tako je bil Python sicer popolnoma predmetno usmerjen, vendar so bili razredi nekako dveh vrst, oni, iz

30 Že tole je pravzaprav povedano narobe. Nikakor nismo mogli dobiti nove definicije *istega* razreda, temveč zgolj nov razred z istim imenom! Drugačen razred pač ne more biti *isti* razred.

Pythona in oni iz Cja. Python 2.2 je delno poenotil obe vrsti razredov, dokončno pa bo zaradi (nevelikih) nezdružljivosti to opravljeno v Pythonu 3.0.

Razredi, ki uporabljajo spremembe, uvedene v Pythonu 2.2, se imenujejo razredi novega sloga (*new style classes*). Večina kode v Pythonu uporablja nekakšno mešanico obeh vrst, ne da bi programer posebej razmišljal o tem, kaj dela. Tega pa ne počne zato, ker mu niti ni treba. Sami smo košarico sprva definirali kot razred starega sloga, druga definicija, tista, kjer smo jo izpeljali iz `list`, pa je razred novega sloga.

V Pythonih 2.X dobimo razred novega sloga, če ga izpeljemo iz katerega od vdelanih razredov. Če ne čutimo potrebe po tem, da bi kaj dedovali, ga izpeljemo iz `object`. V Pythonu 3.0 razredov starega sloga ne bo več, temveč bodo vsi razredi razredi novega sloga, ne glede na način definicije.

Kaj omogočajo razredi novega sloga, česar stari niso? Predvsem omogočajo definiranje nekaj dodatnih metod, kot so deskriptorji, s katerimi lahko določimo metode za branje in postavljanje (*getter*, *setter*) posameznih atributov. Tako se lahko izognemo `__getattr__` in `__setattr__`, če se želimo. Določimo lahko, naj ima razred le vnaprej definirane atribute. Uporabljamo lahko metodo `super`, ki vrne objekt, s pomočjo katerega lažje kličemo podedovano metodo.

Vse razlike so minimalne in programerja, ki se je šele začel učiti Python, še dolgo ne bodo zadevale.

Več o razredih

Ker je namen knjige bralca, veščega programiranja v kakem drugem jeziku, čim hitreje priučiti Pythona, se pri temi razredov v Pythonu, ne moremo zadržati sto strani – kolikor bi jih lahko napolnili brez težav. V poglavju smo se potrudili le nakazati, kaj obstaja in motivirati bralca, da se po potrebi zakoplje v obilje literature, ki mu je na voljo na spletu.

K razredom pa se bomo zdaj, ko jih poznamo, seveda vračali še ves preostanek knjige. Če je bil Python v prejšnjih poglavjih videti bolj ali manj ne-predmetno usmerjen jezik, saj nas razen občasnih pik (`l.sort`...) na predmete ni spominjalo

prav nič, zdaj vemo, da se v resnici za vsako operacijo skrivajo razni razredi in metode. V prihodnje jih zato ne bomo več ignorirali, temveč vedno – do smiselne globine, seveda – spoznali tudi ozadje reči, o katerih bomo govorili.

Naloge

1. Sestavi *seznam* Fibonaccijevih števil, ki števila računa sproti. Vsa naračunana števila naj shrani, da mu jih naslednjič ne bo več potrebno računati. Uporabljali ga bomo takole.

```
>>> fibo[5]
8
>>> fibo[100]
573147844013817084101L
>>> fibo[50]
20365011074L
```

   Ko smo zahtevali peti element, zaporedje izračuna do petega elementa. Ko smo zahtevali stoti, ga izračuna do stotega. Ko zahtevamo petdesetega, pa vrne že izračunani in shranjeni petdeseti element.

2. S čim manj programiranja sestavi razred za sklad, `stack`, ki bo imel metodi `push(o)` in `pop()` za postavljanje elementa na sklad in jemanje z njega. Poleg tega sme imeti razred še kako drugo metodo. (Namig: izpelji ga iz seznama, `list`.)

Rešitve

1. Definirati moramo razred z metodo `__getitem__`, ki ob vsakem klicu podaljša seznam, kolikor je treba (če je treba) in nato vrne zahtevani člen. Nato sestavimo objekt tega razreda in ta se bo vedel, kot zahteva naloga.

```
class Fibonacci:
    def __init__(self):
        self.shranjena = [1, 1]

    def __getitem__(self, n):
        shr = self.shranjena
        for i in range(n+1 - len(shr)):
            shr.append(shr[-1] + shr[-2])
        return shr[n]
```

```
fibo = Fibonacci()
```

V funkciji smo `self.shranjena` shranili v lokalno spremenljivko `shr`, da je funkcija preglednejša. Zanka `for` se bo izvedla le, če zahtevanega števila še ni v seznamu: (`n+1-len(shr)` je število členov, ki jih je potrebno še naračunati, in če je negativno, ne računamo ničesar).

2. Razred `list` že ima metodo `pop`, metoda `append` pa dela, kar zahtevamo od `push`. Storiti nam je le tole.

```
class stack(list):
    push = list.append
```

Generatorji in operacije nad zaporedji

Kot smo se naučili doslej, je Python predmetno usmerjen jezik, pri čemer moremo predmete tudi bolj ali manj zanemariti. Oba sloga programiranja sodita v "običajno" imperativno programiranje, kot ga prakticira večina danes uporabljanih programskih jezikov. Iz nekaterih bolj eksotičnih programskih jezikov, predvsem Hasklla, pa si je Python sposodil še nekoliko drugačen slog programiranja: funkcijsko programiranje.

Kar se bomo naučili, vodi v kratke in hitre programe, kaj lahko pa tudi v programe, ki jih je nemogoče brati in vzdrževati, ne da bi ob tem kaj posebnega pridobili. Izposojenke iz funkcijskega programiranja moramo zato uporabljati razumno in sem ter tja premagati skušnjavo ter napisati običajno zanko `for` in kak `if` namesto (sicer nedvomno bistroumnega) generatorskega izraza.

Iteratorji

Videli smo, da moremo z zanko `for` prek najrazličnejših tipov objektov, od *seznamov*, *terk* in *nizov*, kjer do elementov pridemo s celoštevilskimi indeksi, pa *datotek*, kjer zanka bere vrstico za vrstico do *slovarjev* in *množic*, kjer ni ne številskih indeksov, ne česa drugega, kar bi se dalo po vrsti brati. Kako pravzaprav v resnici deluje zanka `for`? S katerimi tipi zna delati? Če definiramo svoj razred, ki vsebuje več elementov, kaj moramo storiti, da ga bomo lahko uporabili v zanki?

Kot smo mimogrede omenili v poglavju o razredih: zanka `for` zahteva metodo `__iter__` (če je ni, pa si pomaga z `__getitem__`, a to v tem poglavju ni pomembno). V resnici `__iter__` še ne naredi ničesar, `__iter__` le vrne objekt, ki je zmožen iterirati prek objekta. Razred `list` (seznam) ima metodo `__iter__`, ki vrne iterator, ki zna iterirati prek seznama, za katerega smo jo poklicali. Razred `file` ima metodo `__iter__`, ki vrne iterator, ki zna vrstico za vrstico brati datoteko. Razred `dict` ima `__iter__`, ki vrne iterator, ki vrača

ključe, ki se pojavljajo v slovarju.

Kako pa je videti takšen iterator? Kot objekt, ki ima funkcijo `next`, ki ob vsakem klicu vrne nov element, ko jih zmanjka, pa sproži izjemo `StopIteration`. Ta, ki kliče iterator, mora razumeti, da `StopIteration` ne pomeni napake, temveč le konec iteracije, zato mora to izjemo ujeti in nehati klicati `next`.

Poglejmo si iterator v akciji. Na roko poskusimo, kar sicer za nas počne `for`.

```
>>> l = [2, 5, 3]
>>> i = l.__iter__()
>>> i.next()
2
>>> i.next()
5
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
StopIteration
```

Enako bi lahko počeli z datoteko, nizom... Če ima razred funkcijo `__iter__`, ta vrne objekt, ki ima metodo `next`. To je vsa skrivnost, zaradi katere je `for` zmožen iterirati prek toliko različnih objektov.

Generatorji

Napišimo funkcijo, ki bo vračala Fibonaccijeva števila. Emmm, mar nismo tega že storili? Nismo, nismo. Napisali smo funkcijo, ki vrne Fibonaccijevo število. Eno. Znali bi napisati tudi funkcijo, ki vrne seznam Fibonaccijevih števil, recimo prvih petdeset. Kaj pa bi torej rad zdaj? Funkcijo, s katero bom lahko napisal tole:

```
for i in fibonacci(40):
    print i
```

pa se bo izpisalo prvih 40 Fibonaccijevih števil. Poleg tega pa nočem, da funkcija dejansko sestavi spisek. Hočem, da vsakič vrne po eno število.

Funkcija bi torej morala izgledati *nekako tako*

```
# Tole ne deluje!
def fibonacci(n):
    a = b = 1
    for i in range(n):
        return a
        a, b = b, a+b
```

Pri tem pa želim, da `return` ne pomeni konca izvajanja funkcije, temveč le "tule vrni rezultat, ko te naslednjič pokličejo, pa se izvajaj od tod naprej.

Prava rešitev je zelo podobna tej *nekako-tako*-rešitvi.

```
def fibonacci(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a+b
```

Ukaz `yield` dela točno, kakor smo opisali zaželeno vlogo gornjega `return`. Funkcijam, ki vsebujejo `yield`, pravimo generatorske funkcije. Običajno jih uporabljamo tako, da jih pokličemo v zanki `for`, tako kot smo storili zgoraj.

Tule je, za spremembo, še zaporedje Fibonaccijevih števil, ki so deljiva s 7.

```
def fibonacci7(n):
    a = b = 1
    for i in range(n):
        if a % 7 == 0:
            yield a
        a, b = b, a+b
```

Kakšne so prednosti generatorskih funkcij pred običajnimi funkcijami, ki bi preprosto sestavile seznam? No, prednost je pač v tem, da ne sestavijo seznama, ki ga ne potrebujemo. To je koristno predvsem, kadar bi bil seznam dolg. Vas pri pisanju zanke `for` ni nikoli motilo, da z `range` sestavimo seznam števil, prek katerih iteriramo? Da `for i in range(100)` v resnici sestavi seznam števil od `0` do `99`, da `i` potem po vrsti zavzema vrednosti s seznama? Namesto `range` *bi lahko* zdaj definirali svojo funkcijo, imenovali bi jo lahko `xrange`, ki bi le vračala števila, ne da bi vnaprej sestavila kak seznam.

```
def xrange(start, stop, step):
    while start < stop:
        yield start
        start += step
```

Preprosto, ne? Tole sicer žal zahteva, da klicatelj poda vse tri argumente, pa še z negativnim korakom ima težave, saj je potrebno tam pogoj `start < stop` obrniti. Čisto zares bi šlo takole:

```
def xrange(start, stop = None, step = 1):
    if stop is None:
        start, stop = 0, start
    while start < stop if step > 0 else start > stop:
        yield start
        start += step
```

Če si odpustimo rokohitrstvo v pogoju v `while`, je tudi ta definicija še vedno kar preprosta. Zakaj pa smo potem postavili "bi lahko" v kurziv? Zato, ker Python to funkcijo že ima in se v kontekstu zanke `for` v resnici vede natanko tako kot `range`, le seznama ne sestavi vnaprej. Če je to tako imenitno, čemu potem `range` dela, kot dela? Zaradi združljivosti. V Pythonu 3.0 pa bo `range` enak trenutnemu `xrange`.

Je mogoče generirati neskončen seznam? Seveda, čemu ne? Tule je funkcija (no, generator), ki vrne *vsa* Fibonaccijeva števila.

```
def fibonacci():
    a = b = 1
    while 1:
        yield a
        a, b = b, a+b
```

Pri uporabi takšnega, neskončnega generatorja moramo seveda paziti, da iteracijo ustavimo s primernim `break` v zanki ali kako drugače.

V generatorju se `yield` lahko pojavi tudi večkrat. Naslednji (ne preveč smiselni) generator sestavi zaporedje 1, 2, 0, 1, 2, 3, 4.

```
def gen():
    yield 1
    yield 2
    for i in range(5):
        yield i
```

Ne moremo pa kombinirati `yield` in `return`. Funkcija je generatorska ali običajna, ne pa oboje.

Za tiste, ki jih muči radovednost, pokličimo generator, recimo `fibonacci`ja, kot da bi bil funkcija (saj nekako tako smo ga klicali tudi doslej, vendar, smo rezultat vedno prepustili zanki, da dela z njim, kar pač mora) in poglejmo, kaj je rezultat takšnega klica.

```
>>> g = fibonacci()
>>> g
<generator object at 0x012EF148>
```

Posebej veliko nismo izvedeli. Kaj je *generator object*, kaj je mogoče početi z njim?

Pa pomislimo: kaj pa počne zanka `for`? To smo se pač ravnokar naučili. Zanka `for` preveri, ali tisto, prek česar smo ji dali lesti, ima metodo `__iter__` in če jo ima, jo pokliče. Kot rezultat dobi iterator, iterator pa je objekt, ki ima metodo `next` in zanka jo kliče, dokler `next` ne pove, da je konec.

Prav. Zanka dobi `g` in pokliče njegov `__iter__`.

```
>>> g
<generator object at 0x012EF148>
>>> g.__iter__()
<generator object at 0x012EF148>
```

Hec, kar samega sebe je vrnil! (V resnici gre za pravilo. Vsi iteratorji imajo metodo `__iter__`, ki vrača njih same.) Dobro, potemtakem mora imeti tudi `next`.

```
>>> g.next()
1
>>> g.next()
1
>>> g.next()
2
>>> g.next()
```

```
3
>>> g.next()
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
StopIteration
```

Zdaj vemo (skoraj) vse. Ko pokličemo generatorsko funkcijo, kot rezultat dobimo tisto, kar smo že spoznali pod imenom iterator. Objekt torej, ki ima metodo `next`, ki (navadno sproti) generira in vrača elemente zaporedja.

Preslikave, filtri in redukcije

Med opisovanjem razredov smo ob izpisovanju vsebine košarice spremenili seznam artiklov v seznam njihovih imen in količin. Da nekoliko poenostavimo primere, sedaj sestavimo le seznam imen artiklov v košarici. Po starem bi to storili takole.

```
names = []
for item, quantity in b.content:
    names.append(item.name)
```

Ker moramo iz seznama elementov pogosto sestaviti drug seznam elementov, izračunan iz prvega, so si v ta namen izmislili posebno funkcijo, `map`. Ta kot argument prejme funkcijo in seznam; funkcijo pokliče za vsak element seznama, kot rezultat pa vrne seznam rezultatov.

```
>>> l = ["Ana", "Berta", "Cilka", "Dani"]
>>> map(len, l)
[3, 5, 5, 4]
```

Gornja koda za vsakega izmed nizov v seznamu pokliče `len` in dobljene rezultate – dolžine nizov – zloži v nov seznam.

Na artiklih bi lahko naredili tole (predpostavimo, da uporabljamo prvo definicijo razreda, kjer so artikli shranjeni v seznamu `content`):

```
def opisArtikla(t):
    return t[0].name
seznamImen = map(opisArtikla, b.content)
```

Vsebina košarice je podana kot seznam parov (terk) `(item, quantity)`. Funkcija `map` bo poklicala `opisArtikla` za vsak element seznama, torej bo argument, ki ga bo dobil `opisArtikla`, terka, katere prvi element bo artikel. `opisArtikla` vrne niz in `map` bo vrnil seznam takšnih nizov.

Pa je to kaj posebej elegantno? Ali preprosteje od zanke `for`? Ne, to pa predvsem zato, ker smo morali *mimogrede definirati kratko funkcijo, ki je samo nekaj malega poračunala in si pravzaprav ni zaslužila niti imena*... Ha, lambda-funkcija! Z njo se gornja koda precej poenostavi.

```
seznamImen = map(lambda t: t[0].name, b.content)
```

Estetskost `map` je stvar okusa, a če že ne drugega, je `map` vsaj hitrejši od `for`. Vendar je `map` prišel v nemilost in odšel iz mode. Kmalu bomo namreč spoznali nekaj boljšega.

Mimogrede, funkciji `map` lahko damo tudi več kot le en seznam. Če ji damo tri, mora funkcija, ki smo jo podali kot argument, sprejemati tri argumente. Funkcija `map` bo šla istočasno, vzporedno, prek vseh treh seznamov in funkciji kot argumente dala po en ("istoležni") element iz vsakega seznama.

Druga pogosta operacija je sejanje (filtriranje) seznama. Pripravimo seznam vseh artiklov v košarici, ki so (posamezen kos oz. kilogram oz. liter...) dražji kot 10.

```
drageReci = filter(lambda t: t[0].price > 10, b.content)
```

Funkcijo smemo tudi izpustiti: če namesto funkcije kot argument podamo `None`, bo `filter` vrnil vse resnične elemente podanega seznama. Če gre, denimo, za seznam nizov, bo sestavil seznam vseh nepraznih nizov.

Tudi `filter` je zelo uporabna reč, a spodnesla ga je ista skrivnostna nova operacija kot `map`.

Tretja funkcija iz te skupine pa je še vedno cenjena. Za razliko od `map` in `filter`, ki dobita sezname in sestavljata sezname, `reduce` iz seznama izračuna en sam objekt. Uporabimo ga lahko, na primer, za to, da seštejemo elemente v danem seznamu.[31]

---

31 Python ima sicer že vdelano funkcijo `sum`.

```
vsota = reduce(lambda x, y: x+y, l)
```

Pa združimo `map` in `reduce` in izračunajmo vrednost košarice.

```
vsota = reduce(lambda x, y: x+y,
               map(lambda t: t[0].price * t[1], b.content))
```

Lambda, ki smo jo dali funkciji `map`, pomnoži ceno vsakega artikla z njegovo količino. Iz tega `map` sestavi seznam vrednosti posameznih artiklov, `reduce` pa to lepo sešteje.

Funkciji `reduce` lahko podamo še tretji argument, začetno vrednost. Če ima seznam `l` štiri elemente, bo `reduce(f, l)` izračunal `f(f(f(l[0], l[1]), l[2]), l[3])` (če je `f` vsota, bomo torej dobili `(((l[0] + l[1]) + l[2]) + l[3])`). Če mu podamo še začetno vrednost, `reduce(f, s)`, pa računa `f(f(f(f(l[0], s), l[1]), l[2]), l[3])`. Začetna vrednost pri vsoti ni potrebna, včasih, kadar združujemo na kak drugačen način, pa utegne biti potrebno inicializirati objekt, v katerega potem "združujemo" druge.

Vse naštete funkcije ne delujejo le nad seznami, temveč nad vsakim objektom, prek katerega je mogoče iterirati. Če želimo seznam nepraznih vrstic datoteke, ga dobimo s

```
filter(None, file("nekaDatoteka.txt"))
```

Pretvorba niza v seznam kod ASCII je takšna:

```
>>> map(ord, "Miran")
[77, 105, 114, 97, 110]
```

Je vse to pregledno? Začetniku gotovo ne in tudi kasneje se preglednost klobas iz reduceov, filtrov, mapov in lambd ne poveča. Še dobro, da se danes to dela tudi drugače. A še preden izvemo, kako, si oglejmo

Druge operacije nad zaporedji

Funkcije `map`, `filter` in `reduce` predstavljajo osnovne operacije, ki jih lahko opravljamo nad zaporedji. Funkcijo za preslikovanje, izbiranje ali sestavljanje moramo podati sami in jo, najpogosteje, tudi sami definirati, s pomočjo lambde ali

kar s pravo funkcijo. Obstaja pa kopica že pripravljenih funkcij.

Funkcija `all(l)` vrne `True`, če so vsi elementi seznama `l` resnični, `any(l)` pa, če je resničen vsaj en element seznama. (Bi znali to zapisati s pomočjo `reduce`?[32])

Funkciji `max(l)` in `min(l)` vrneta najmanjši in največji element seznama. Obe je mogoče klicati tudi drugače, `max(a, b, c, d)` bo vrnil največjega izmed argumentov. Bi lahko prvo različico `max`, to, ki dela nad seznami, definirali s pomočjo druge?[33]

Vsoto elementov seznama dobimo s `sum`, vendar le za sezname števil. (Bi znali, recimo s pomočjo `reduce`, definirati svoj `sum`, ki bi delal tudi nad objekti drugačnih tipov, na primer nad seznami nizov?[34])

Funkciji `sorted` in `reversed` kot argument dobita seznam, vrneta pa urejen seznam oz. seznam z obratnim vrstnim redom argumentov.

Najpomembnejši funkciji smo prihranili za konec. Funkciji `zip` (zadrga) podamo dva ali tri ali več seznamov in vrne seznam parov, trojk oziroma terk z elementi iz podanih seznamov.

```
>>> l1 = ["a", "b", "c"]
>>> l2 = [1, 2, 3]
>>> zip(l1, l2)
[('a', 1), ('b', 2), ('c', 3)]
```

Kot ostale funkcije iz tega razdelka tudi `zip` deluje nad zaporedji poljubnih, morda celo različnih vrst.

```
>>> s = "abc"
>>> zip(s, l2)
[('a', 1), ('b', 2), ('c', 3)]
```

---

32 `reduce(operator.and_, l)`, vendar to ni učinkovito, saj vedno pregleda vse elemente seznama, namesto da bi se ustavil ob prvem neresničnem.

33 Poleg očitne rešitve, klica z argumenti v terki, lahko uporabimo `reduce(max, l)`, ki izračuna natanko to, kar potrebujemo, namreč `max(max(max(l[0], l[1]), l[2]), l[3])`.

34 Saj smo ga že! Večji izziv je napisati `sum`, ki tega ne bo znal...

Zadrgo pogosto uporabljamo v zanki `for`, kjer bi radi šli prek dveh seznamov hkrati.

```
>>> l1 = ["Zoran", "Vlado", "Peter"]
>>> l2 = ["Predin", "Kreslin", "Lovsin"]
>>> for ime, priimek in zip(l1, l2):
...     print ime, priimek
...
Zoran Predin
Vlado Kreslin
Peter Lovsin
```

Če seznami niso enako dolgi, se zadrga ustavi pri najkrajšem (tako kot se zadrge, predvsem one na starih bundah, vedejo tudi v resničnem svetu).

Zdaj pa še zadnja funkcija. Pogosto želimo iterirati prek seznama, obenem pa poleg elementov poznati tudi njihove indekse. Doslej smo morali to reševati takole:

```
>>> for i in range(len(l2)):
...     print "Pevec %i: %s" % (i+1, l2[i])
...
Pevec 1: Predin
Pevec 2: Kreslin
Pevec 3: Lovsin
```

Zdaj, ko poznamo `zip`, nam je na voljo ... no, priznajmo, nič preveč elegantna alternativa.

```
>>> for i, pevec in zip(range(len(l2)), l2):
...     print "Pevec %i: %s" % (i+1, pevec)
...
Pevec 1: Predin
Pevec 2: Kreslin
Pevec 3: Lovsin
```

Priznam, brez zadrge je bilo lepše. Zato pa imamo že pripravljeno funkcijo `enumerate`, ki naredi natanko to, v kar smo zgoraj prisilili `zip`.

```
>>> for i, pevec in enumerate(l2):
...     print "Pevec %i: %s" % (i+1, pevec)
...
Pevec 1: Predin
```

```
Pevec 2: Kreslin
Pevec 3: Lovsin
```

Mimogrede, tole lahko luštneje zapišemo brez razkopavanja terke, ki smo jo v `print` tako ali tako morali ponovno sestavljati:

```
for t in enumerate(l2):
    print "Pevec %i: %s" % t
```

Za še en primer napišimo funkcijo, ki za seznam točk, na primer `[(0, 0), (1, 1), (2, 3), (2, 4), (-1, 0)]`, izračuna razdaljo med najbolj oddaljenima točkama.

```
from math import sqrt

def najdalj(s):
    naj2 = -1
    for i, t in enumerate(s):
        for u in s[:i]:
            d2 = (t[0]-u[0])**2 + (t[1]-u[1])**2
            if d2 > naj2:
                naj2 = d2
    return sqrt(naj2)
```

Prva zanka teče prek celega seznama, druga pa preteče le vse elemente pred onim iz prve zanke. V prvi zanki zato potrebujemo `enumerate`, s katerim poleg elementa dobimo še indeks, `i`, ki ga uporabimo, da v drugi zanki skrajšamo seznam (`s[:i]`).

V zanki bi lahko razpakirali tudi koordinate točke, tako da bi `t` zamenjali z `(x1, y1)`.

```
def najdalj(s):
    naj2 = -1
    for i, (x1, y1) in enumerate(s):
        for (x2, y2) in s[:i]:
            d2 = (x1-x2)**2 + (y1-y2)**2
            if d2 > naj2:
                naj2 = d2
    return sqrt(naj2)
```

Izpeljevanje seznamov

Zdaj pa končno poglejmo, kaj je boljše od `map` in `filter`.

Matematiki uporabljajo eleganten način opisovanja množic. Imejmo neko množico M, iz katere bi radi sestavili množico K, ki bo vsebovala kvadrate vseh števil iz M, vendar le tistih, ki so manjša od deset. Opisali bi jo takole.

$K = \{ x^2 \mid x \in M, x < 10 \}$

Tako iz ene množice izpeljemo drugo. Python pozna podobno operacijo, vendar nad seznami. Sintaksa je precej podobna.

```
k = [x**2 for x in m if x < 10]
```

Temu rečemo izpeljevanje seznama[35] (*list comprehension*).

Izpeljevanje seznama zna očitno vse, kar znata `map` in `filter`. Ko smo predstavili `map`, smo pisali

```
seznamImen = map(lambda t: t[0].name, b.content)
```

Zdaj lahko rečemo

```
seznamImen = [t[0].name for t in b.content]
```

Podobno opravimo s filtrom,

```
drageReci = filter(lambda t: t[0].price > 10, b.content)
```

se spremeni v

```
drageReci = [t for t in b.content if t[0].price > 10]
```

Slednji nemara ni najbolj prepričljiv, sintaksa `t for t in ...` je videti okorno. Kdor noče, pa naj uporabi `filter`. Celo, če bi ga v Pythonu 3.0 ukinili, kot se je nekaj časa šušljalo, bi si lahko preprosto definirali svojega.

---

35 Pri iskanju prevoda je pomagal Andrej Bauer, končne izbire je kriv avtor sam.

```
def filter(f, l):
    if f is None:
        return [x for x in l if x]
    else:
        return [x for x in l if f(x)]
```

Tudi `map` je včasih videti privlačneje od izpeljevanja. Lepota izpeljevanja je, da ne zahteva lambda-funkcije, saj je izraz del sintakse izpeljevanja. Kadar je funkcija že na voljo, pa je `map` krajši. Ob razlagi funkcije `map` smo napisali:

```
map(len, l)
```

Različica z izpeljanim seznamom je nekoliko daljša.

```
[len(x) for x in l]
```

Kaj od tega je prijetneje brati, je stvar navade in okusa. Komur je bližji `map`, mu bo prišel prav modul `operator`, v katerem so običajni operatorji (seštevanje, množenje, enakost...), dostopni kot funkcije (`operator.add`, `operator.mul`, `operator.eq`...).

Vsekakor pa so izpeljani izrazi najbolj priročni, ko potrebujemo obe operaciji, `map` in `filter`. Denimo takrat, ko bi radi imena vseh dragih artiklov.

```
drageReci = [t[0].name for t in b.content if t[0].price > 10]
```

Lepše je pravzaprav

```
drageReci = [item.name for item, quantity in b.content if
item.price > 10]
```

Ob izpeljevanju smemo sestavljati tudi terke ali druge strukture. Tule je seznam imen dragih rečin in njihovih količin.

```
drageReci = [(item.name, quantity) for item, quantity in
b.content
               if item.price > 10]
```

Nekoč davno smo govorili o tem, da seznama praznih seznamov ne smemo sestaviti takole.

```
prazni = [[]]*5
```

Pač pa navadno storimo tako.

```
prazni = [[] for i in range(5)]
```

Izraz je zanimiv, ker `i`-ja niti ne uporabimo. Isti učinek bi dosegli s sicer neobičajnim

```
prazni = [[] for i in "abcde"]
```

Rezultat izpeljevanja seznamov je vedno seznam, tisto, iz česar ga izpeljujemo, pa je lahko tudi kaj drugega – karkoli pač, s čimer zna delati zanka `for` – nizi, slovarji, terke. Datoteke? No, seveda. Če želimo iz datoteke pobrati vse neprazne vrstice in jih olupiti belega prostora, to storimo v enem zamahu.

```
[line.strip() for line in file("besedilo.txt") if line]
```

(Tole deluje le v ideji, v praksi pa ne. Popravek sledi v naslednjem razdelku...)

Pa generatorji? Smemo pri izpeljevanju uporabiti tudi generatorje ali pa je to že prezapleteno? Seveda, predvsem generatorje. (Medklic: v čem pomembnem pa se datoteka pravzaprav sploh razlikuje od generatorja?! Datoteka je generator vrstic...) Tole so kvadrati vseh tistih fibonaccijevih števil med prvimi stotimi, ki so deljiva s 7.

```
>>> [x**2 for x in fibonacci(100) if x % 7 == 0]
[441, 974169, 2149991424L, 4745030099481L, 10472279279564025L,
2311231562496770457 6L, 5100887011202444436089L,
11257655322492232390 2744729L, 24845640195853345682891318169 6L,
54834316654593011429908748925922 5L,
12101931201104658037246292598819287 61L,
26708956677406314828901424774719275171 84L]
```

Še več primerov bomo videli v rešitvah nalog, ki jih v tem poglavju ne manjka.

Generatorski izrazi

Nad seznami smo se pritoževali, da jih pogosto ne potrebujemo. Potarnali smo, da `range(1000000)` sestavi seznam z milijon števili, pa četudi ga kanimo uporabiti v zanki `for`, kjer potrebujemo le zaporedna števila, po eno naenkrat, in ne vseh hkrati. V izogib nepotrebnim seznamom smo spoznali generatorje, ki dejansko vračajo po en element naenkrat.

Nato smo spoznali izpeljevanje seznamov. Če potrebujemo kvadrate tisoč fibonaccijevih števil, nam jih priskrbi

```
[x**2 for x in fibonacci(1000)]
```

Nam je tole všeč? Kaj pa, če ta seznam potrebujemo le zato, da bomo prek njega iterirali?

```
for i in [x**2 for x in fibonacci(1000)]:
    print i
```

Pa smo spet na začetku, ne? Da bi števila izpisali, smo sestavili celoten seznam, čeprav bi nam zadoščalo eno število naenkrat.

Težava je v tem, da izpeljevanje seznamov naredi natanko to, kar ime pravi, da naredi. Seznam.

Kaj pa "izpeljevanje generatorjev"? Nekaj, kar bi izgledalo podobno kot izpeljevanje seznama, vendar bi vrnilo generator, ne seznama? Ideja je dobra, le ime je drugačno. Temu pravimo generatorski izrazi. Dobimo jih preprosto tako, da pri izpeljevanju seznama izpustimo oklepaje.

```
>>> x**2 for x in fibonacci(1000)
Traceback (  File "<interactive input>", line 1
    x**2 for x in fibonacci(1000)
           ^
SyntaxError: invalid syntax
```

Hm, no, skoraj tako. Včasih gre brez, večinoma pa zaradi sintaktičnih razlogov dodamo okrogle oklepaje.

```
>>> (x**2 for x in fibonacci(1000))
<generator object at 0x012EF198>
```

Kar smo dobili, je generator, objekt, ki ga lahko uporabimo povsod, kjer lahko sicer uporabljamo generatorje. Denimo v zanki.

```
>>> for i in (x**2 for x in fibonacci(10)):
...     print i
...
1
1
4
9
```

```
25
64
169
441
1156
3025
```

Ali pri izpeljevanju seznamov.

```
>>> ["f%i" % i for i in (x**2 for x in fibonacci(10))]
['f1', 'f1', 'f4', 'f9', 'f25', 'f64', 'f169', 'f441', 'f1156',
'f3025']
```

Tale primer je bil nekoliko umeten, naslednji pa ne bo. Se spomnite, kako iz datoteke prebrati vse neprazne vrstice in odstraniti beli prostor?

```
[line.strip() for line in file("besedilo.txt") if line]
```

Pogoj `if line` je resničen, čim je vrstica `line` neprazna, pa čeprav je tisto, kar vsebuje, morda samo beli prostor – kak presledek ali celo samo znak za novo vrsto. Pravilno bi bilo

```
[line.strip() for line in file("besedilo.txt") if line.strip()]
```

To pa nam ni všeč, ker dvakrat pokličemo `strip`. Tega se znebimo z dodatnim generatorjem (oklepaji okrog generatorja so izpisani malo večji, da ne preveč čitljivi izraz lažje preberemo).

```
[line for line in (l.strip() for l in file("besedilo.txt")) if
line]
```

Generator vrne "olupljene vrstice", pri izpeljevanju seznama pa izberemo le neprazne. Če to še ni končna postaja, pa lahko vse skupaj spremenimo v nov generator.

```
(line for line in (l.strip() for l in file("besedilo.txt")) if
line)
```

Čeprav generatorje na koncu koncev vedno uporabljamo v zankah, nam je zanka včasih tudi skrita. Moremo z vdelano funkcijo `sum` izračunati vsoto kvadratov prvih tisoč Fibonaccijevih števil?

```
>>> sum(x**2 for x in fibonacci(1000))
3057018985272083280891769550770709134443022741973377352285622728485339924629663085880556303646279364980699180369468530938703716323
2003437472967819927007152115225052849839294525464545244408527529080083391004460907758759027766765632329201613696317437463342788779
1182797696864935655412157466783380476967806614030314962244622315773287342597409177188867302433121412464029059694406425161745696115
8954609924765369132325291375L
```

Funkciji nismo dali seznama tisoč števil, temveč le generator. Se je morala kaj posebej potruditi, da ga je prebavila? Ne, sploh ne, mirno je lahko definirana takole.

```
def sum(l):
    s = 0
    for i in l:
        s += i
    return s
```

To deluje tako s seznami kot z generatorji. Če želimo izračunati podobno vsoto le za Fibonaccijeva števila deljiva s 7, pač dodamo še pogoj.

```
>>> sum(x**2 for x in fibonacci(1000) if x % 7 == 0)
1890198093192303838201340574885918651635253862285094001406592976274721363232198414361512642793564090489604546668245878513736424450460683891593091363506778235700094634740093681198438759858345658022813605155755582585478600151408437730190982725790440418346518149189056237947169245379610112620561074427028098882616568956502928442139290658908120095542345514851048924517580951503328092264400010505850044241613701985725408500L
```

Funkcija, pri kateri nam je še posebej všeč, da sprejema tudi generatorje, je metoda `str.join`. Omogoča nam namreč sestavljanje nizov iz reči, ki jih generiramo kar sproti.

```
>>> ", ".join(str(x**2) for x in fibonacci(10))
'1, 1, 4, 9, 25, 64, 169, 441, 1156, 3025'
```

Generatorji v Pythonu 3.0

Python v začetku ni imel generatorjev, zato so funkcije, kot so `range`,

`enumerate`, `filter`, `map`, ter metode, kot so `keys`, `values` in `items`, vračale sezname. Po prihodu generatorjev bi bilo smiselno vse te funkcije spremeniti v generatorske funkcije, vendar tega niso storili zaradi ohranjanja združljvosti.

Najbolj očiten primer je `enumerate`, ki ga uporabljamo skoraj samo v zankah (še `range` je sem ter tja uporaben še kje drugje, `enumerate` pa res skoraj ne), vendar vedno sestavi nov, oštevilčen seznam.

Ker je namen Pythona 3.0 opraviti z ostanki preteklosti, tudi za ceno nezdružljivosti, bodo vse funkcije in metode, ki vračajo sezname, čeprav bi bilo bolj smiselno, da bi vračale generatorje, v Pythonu 3.0 tudi dejansko vračale generatorje.

Najpogostejša nezdružljivost, ki jo bo to povzročilo, bo povezana z metodo `keys`. Denimo, da v slovar shranimo osebne podatke, pri čemer je ključ priimek in ime osebe. Ko je potrebno podatke izpisati, s `keys` dobimo seznam ključev, ga s `sort` uredimo po abecedi in nato izpišemo. Ko bo `keys` postal generator, ga ne bo mogoče več *urediti*, saj urejamo le sezname. Pač pa bi morali, če bi želeli delati enako kot prej, iz generatorja najprej dobiti v seznam (s klicem konstruktorja `list`, ki mu kot argument damo generator), tega pa lahko urejamo, kakor želimo. Še preprostje pa je vse skupaj zaupati funkciji `sorted`.

Bralec knjige naj bo torej pozoren, da to, kar smo povedali v tem poglavju v novejših verzijah Pythona ... ne *ne velja več*, temveč bo postalo še bolj res in še bolj pomembno. Le to in ono se bo napisalo enostavneje ali izvajalo hitreje in z manj pomnilnika.

Naloge

1. Sestavi množico vseh končnic datotek v trenutnem direktoriju.

2. Številka ISBN ima deset števk. Zadnja števka je kontrolna. Dobimo jo tako, da prvo števko pomnožimo z ena, drugo z dve, tretjo s tri in tako do devete. Vse skupaj seštejemo in izračunamo ostanek po deljenju z enajst. Rezultat je deseta števka. V primeru, da je ostanek po deljenju z 11 enak 10, je deseta "števka" X. Napiši funkcijo,

ki ji podamo prvih devet števk (kot niz) in vrne deseto. Za primer pravilne ISBN poglejte na svojo knjižno polico ali kam na splet.

3. Napiši funkciji za pretvorbo niza v niz s šestnajstiškimi kodami ASCII in nazaj. Takole naj delujeta.

```
>>> toHex("Tine")
'54 69 6E 65'
>>> toString('54 69 6E 65')
'Tine'
```

4. Napiši binarni števec s podanim številom števk, ki ga bo mogoče uporabljati takole.

```
>>> for e in BoolCounter(3):
...       e
...
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
```

Nalogo reši enkrat z generatorjem, enkrat pa tako, da je `BoolCounter` razred.

5. Napiši generator, ki kot argument prejme množico, podano kot seznam, in vrne vse njene podmnožice.

```
>>> for e in subsets([1, 2, 3]):
...       e
...
[]
[3]
[2]
[2, 3]
[1]
[1, 3]
[1, 2]
[1, 2, 3]
```

Pomagaj si z generatorjem, ki si ga napisal v prejšnji nalogi.

6. Denimo, da Benevolentnemu diktatorju pade na glavo opeka in v Pythonu 4.0 ukine funkcije `map`, `filter` in `reduce` (vse ostalo nam je pustil). Kako bi jih napisali? Kaj pa, če bi izgubili tudi izpeljevanje seznamov?

7. Zdaj pa napiši funkciji `map`, `filter` kot generatorja, torej tako, da seznama ne sestavita, temveč vračata elemente zaporedja.

8. Za seznam pravokotnikov, podanih s koordinatama levega spodnjega in desnega zgornjega oglišča, se pravi terko ((x1, y1), (x2, y2)), poišči njihov presek. Kako pa bi bila videti rešitev za poljubno število dimenzij?

9. Napiši funkcijo, ki izračuna fakulteto *n*.

Rešitve

1. Trivialno.

```
>>> import set, os
>>> set(os.path.splitext(f)[1].lower() for f in os.listdir(".."))
set(['.png', '.bak', '.odg', '.odt', '.py', '.txt', '.pdf'])
```

2. Funkcijo bi lahko z vso pravico napisali v eni vrstici, a jo dajmo v dveh, da jo bo lažje prebrati.

```
def isbn(s):
    vsota9 = sum((i+1)*int(st) for i, st in enumerate(s))
    return "0123456789X"[vsota9 % 11]
```

   Komentar ni potreben, le pozorno preberite, kar piše.

3. Takšne stvari je najboljše sestavljati po korakih, da sproti preverjamo in popravljamo izraz.

```
>>> s = "Tine"
>>> [ord(c) for c in s]
[84, 105, 110, 101]
>>> [hex(ord(c)) for c in s]
['0x54', '0x69', '0x6e', '0x65']
>>> [hex(ord(c))[2:] for c in s]
['54', '69', '6e', '65']
>>> [hex(ord(c))[2:].upper() for c in s]
['54', '69', '6E', '65']
>>> " ".join([hex(ord(c))[2:].upper() for c in s])
'54 69 6E 65'
>>> " ".join(hex(ord(c))[2:].upper() for c in s)
'54 69 6E 65'
```

V eno stran smo že zmagali. Zdaj pa še nazaj.

```
>>> s = '54 69 6E 65'
>>> s.split()
['54', '69', '6E', '65']
>>> [int(h, 16) for h in s.split()]
[84, 105, 110, 101]
>>> [chr(int(h, 16)) for h in s.split()]
['T', 'i', 'n', 'e']
>>> "".join([chr(int(h, 16)) for h in s.split()])
'Tine'
>>> "".join(chr(int(h, 16)) for h in s.split())
'Tine'
```

Zahtevani funkciji sta torej

```
def toHex(s):
    return " ".join(hex(ord(c))[2:].upper() for c in s)

def toString(s):
    return "".join(chr(int(h, 16)) for h in s.split())
```

4. Rešitev s funkcijo je kratka, vendar nekoliko zavozlana.

```
def BoolCounter(n):
    state = [0]*n
    while True:
        yield state[:]
        for i in range(n-1, -1, -1):
            state[i] = 1 - state[i]
            if state[i]:
                break
        else:
            break
```

Najprej premislimo zanko `for`, katere naloga je naračunati naslednje stanje števca (`state`). Z leve proti desni pregleduje trenutno stanje in spreminja enke v ničle. Ko prvič spremeni ničlo v enko, je opravila svoje, zato izskoči iz zanke z `break`.

Zdaj pa `while`. Najprej vrne trenutno stanje. V naslednjem krogu naračuna naslednje stanje in ga vrne. Razen, če `for` ni uspel spremeniti nobene ničle v enko (kar se zgodi, ko so ostale le še enke) in se zato ni končal z `break`. V tem primeru se izvede, kar je napisano pod `else`: drugi `break` prekine `while` in generator se izteče.

Zdaj pa še umazani detajl: `state[:]`. Čemu?! To naredi kopijo objekta `state`. Brez kopiranja bi stalno vračali en in isti objekt, ki pa ga v funkciji kasneje spreminjamo. Če bi kličoča funkcija vrnjeni objekt shranjevala, bi shranila več kopij istega objekta, ne pa različna stanja.

```
>>> list(BoolCounter(2))
[[0, 0], [0, 0], [0, 0], [0, 0]]
```

Naj bralca ne zavede: tule `[0, 0]` ni začetno stanje, temveč končno, tisto, ki ga dobimo, ko zanka `for` postavi vse enice nazaj na ničlo, zato se ne izvede `break` v zanki `for`, temveč oni v `while`.

Rešitev z razredom je zanimiva predvsem, ker njena dolžina in zapletenost pokažeta, koliko dela nam prihranijo generatorji. Potrebovali bomo dva razreda: tistega, po katerem iteriramo in onega, ki bo predstavljal iterator.

```
class BoolCounter:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return BoolCounterIterator(self.n)

class BoolCounterIterator:
    def __init__(self, n):
        self.n = n
        self.state = None

    def __iter__(self):
        return self

    def next(self):
        if not self.state:
            self.state = [0]*self.n
            return self.state

        for i in range(self.n-1, -1, -1):
            self.state[i] = 1 - self.state[i]
            if self.state[i]:
                return self.state

        raise StopIteration
```

Prvi razred, `BoolCounter`, si zapomni, koliko števk potrebujemo, in njegova metoda `__iter__` vrne primerno inicializirano instanco `BoolCounterIterator`. Ta ima metodo `__iter__`, ki, kot velevajo pravila, vrne sam objekt. Resnično delo opravlja `next`. Ko ga pokličemo prvič, sestavi seznam ničel, prvo stanje, in ga vrne. Naslednjič pa s podobno zanko `for`, kot smo jo napisali v prejšnji rešitvi, izračuna naslednje stanje, le da ga tokrat, ko opravi delo (torej, ko spremeni ničlo v enko), vrne. Če mu to ne uspe, sproži izjemo `StopIteration`, ki sporoči, da je zaporedja konec.

5. Tole je pa res elegantno.

```
def subsets(s):
    for bc in BoolCounter(len(s)):
        yield [e for e, i in zip(s, bc) if i]
```

Rešitev je generator, ki temelji na drugem generatorju. Binarni števec iz prejšnje naloge uporabimo tako, da vsakemu elementu podane množice odgovarja ena števka. V vsakem koraku vrnemo tiste elemente, katerih pripadajoče števke so enake 1. Za izbor poskrbi izpeljani seznam v stavku `yield`. Z njim istočasno iteriramo prek množice in števca, `zip(s, bc)` ter element `e` dodamo, ko je števka `i` resnična (se pravi 1).

6. Najprej z izpeljevanjem seznamov.

```
def map(f, *s):
    return [f(*x) for x in zip(*s)]

def filter(f, s):
    return [x for x in s if f(x)]

def reduce(f, s, *d):
    if d:
        r = f(d[0], s[0])
    else:
        r = s[0]

    for x in s[1:]:
        r = f(r, x)
    return r
```

V `map` smo predvideli, da jo lahko pokličemo s poljubnim številom seznamov; argument `S` bo terka seznamov. Funkcija `f` mora sprejeti toliko argumentov, kolikor je dolg `s`. Z `zip` poskrbimo, da iz vsakega seznama vzamemo po en element; `zip` tule naredi nekakšno transpozicijo.

```
>>> s = [[1, 2, 3], [4, 5, 6]]
>>> zip(*s)
[(1, 4), (2, 5), (3, 6)]
```

Spremenljivka `x` bo torej terka z vsemi prvimi, drugimi, tretjimi (in tako naprej) elementi seznamov podanih v `s`. Terko uporabimo kot argumente (ne argument, argumente!) za `f`, ki jo moramo zato poklicati z `f(*s)`.

Funkcija `reduce` je nerodna, ker sprejema še tretji argument, začetno vrednost, ki je lahko poljubnega tipa. Argument zato prejmemo kot terko s poljubnim številom dodatnih argumentov (koliko jih je, ne preverjamo – to bi bilo potrebno v funkcijo še dodati). Če je terka neprazna, vzamemo za začetno vrednost njen prvi element,

ki mu že takoj dodamo prvi element seznama, sicer pa je začetni element pač prvi element seznama. Ker smo znotraj `if-else` tako v obeh primerih že uporabili prvi element, v zanki dodajamo elemente od drugega naprej.

Funkciji `map` in `filter` brez izpeljevanja seznamov sta seveda nekoliko daljši, pa tudi počasnejši.

```
def map(f, *s):
    l = []
    for x in zip(*s):
        l.append(f(*x))
    return l

def filter(f, s):
    l = []
    for x in s:
        if f(x):
            l.append(x)
    return l
```

7. Kode je manj kot prej.

```
def map(f, *s):
    for x in zip(*s):
        yield f(*x)

def filter(f, s):
    for x in s:
        if f(x):
            yield x
```

V resnici pa je še ta prezapletena. Kaj vračata gornji funkciji, vemo. Generatorja.

```
>>> filter(lambda x:x % 2, range(10))
<generator object at 0x013003A0>
```

Če je tako, pa generatorja raje vrnimo kar eksplicitno.

```
def map(f, *s):
    return (f(*e) for e in zip(*s))

def filter(f, s):
    return (e for e in s if f(e))
```

Rešitev je, ne po naključju, enaka prvi rešitvi prejšnje naloge, le z drugačnimi oklepaji.

8. Poiskati moramo največjo koordinato x levega spodnjega oglišča, največji y levega spodnjega, ter najmanjši koordinati x in y desnega zgornjega. Rešitev sestavimo postopno. Vzemimo seznam s tremi kvadrati.

```
kvadrati = [((1, 2), (10, 13)), ((3, 1), (11, 8)), ((1, 4), (9,
10))]
```

Iz njega poberimo vsa leva spodnja oglišča.

```
>>> [k[0] for k in kvadrati]
[(1, 2), (3, 1), (1, 4)]
```

Ločimo jih v seznam koordinat x in seznam koordinat y.

```
>>> zip(*[k[0] for k in kvadrati])
[(1, 3, 1), (2, 1, 4)]
```

Zdaj pa izračunajmo maksimum vsakega od seznamov.

```
>>> [max(x) for x in zip(*[k[0] for k in kvadrati])]
[3, 4]
```

Slednje bi bilo pravzaprav videti lepše z `map` kot z izpeljevanjem seznamov.

```
>>> map(max, zip(*[k[0] for k in kvadrati]))
[3, 4]
```

Za zgornje desno vozlišče iščemo minimum, torej

```
>>> map(min, zip(*[k[1] for k in kvadrati]))
[9, 8]
```

Potrebno je le še pretvoriti oba rezultata v terko in ju združiti v novo terko. Poleg tega bi morali še preveriti, da presek morda ni prazen: če je katera od koordinat levega spodnjega oglišča preseka večja od koordinate desnega zgornjega, je presek prazen.

Kako pa nalogo rešimo za poljubno število dimenzij? Smo je že: naša rešitev je čisto splošna, saj je map čisto vseeno, ali dobi le dva seznama (za x in y), kot sedaj, ali več.

9. To nalogo smo že reševali, čisto na začetku, vendar z zanko `for`. Zdaj jo znamo rešiti preprosteje.

```
def fakulteta(n):
    from operator import mul
    return reduce(mul, range(2, n+1), 1)
```

Introspekcija

Python vidi vase. Med izvajanjem programa lahko dobi slovar spremenljivk. Za vsak objekt lahko ugotovi kakšnega tipa je. Če je razred, lahko poizve o njegovih metodah, poljih in prednikih ter o tem, kakšne operatorje podpira. Če funkcija, se lahko pozanima o številu argumentov, privzetih vrednostih in še čem. Če modul, lahko izve, kaj vsebuje.

Še več. Program ima dostop do samodejnega smetarja (*garbage collection*), do lastnega sklada in, pravzaprav do vsega drugega, povezanega s sabo. Pythonov tolmač je skoraj tako dostopen, kot da bi bil pisan v Pythonu, to pa zato, ker dejansko shranjuje vse, kar je povezano z izvajanjem programa (spremenljivke, sklad...), kar v Pythonovih strukturah. Tako mu jih ni težko napraviti vidne programu.

Branje poglavja večini ne bo v posebno korist, torej ga lahko bralec brez škode preskoči. Všeč pa bo tistim, ki jih zanima, kako so stvari v resnici narejene.

Spremenljivke. Kako dobimo slovar spremenljivk, smo že videli. Funkcija `globals` vrne slovar vseh globalnih spremenljivk, s tem da gre pri modulu za "globalne spremenljivke" modula, ne "glavnega programa". Ključi slovarja so imena spremenljivk, pripadajoče vrednosti pa objekti, ki jih imena imenujejo. Funkcija `locals` vrne podoben slovar s trenutnimi lokalnimi spremenljivkami.

Objekti in razredi. Objekt ima polje `__dict__`, v katerem so shranjena vsa njegova polja. `__dict__` igra podobno vlogo kot `locals` in `globals`, le da se nanaša na posamezen objekt. S pisanjem v slovar spreminjamo ali dodajamo polja objektu. `a.__dict__["y"]=42` pomeni isto kot `a.y = 42`, druga oblika je le sintaktični sladkorček.

```
>>> class A:
...     pass
...
>>> a = A()
>>> a.x = 12
>>> a.__dict__
{'x': 12}
```

```
>>> a.__dict__["y"]=42
>>> a.y
42
```

Polje `__class__` vrne razred objekta. Kot smo sestavili razrede in objekte zgoraj, bi `a.__class__` vrnil `A`.

```
>>> a.__class__
<class __main__.A at 0x013B6FC0>
>>> A
<class __main__.A at 0x013B6FC0>
```

Polja `__class__` ne smemo uporabljati za preverjanje tipa objekta. Izraz `a.__class__ == A` bo resničen le, če je `a` objekt razreda A, ne pa tudi, če gre za objekt razreda, izpeljanega iz `A`. Ker pri preverjanju tipov navadno mislimo slednje, uporabimo funkcijo `isinstance(a, A)`, ki vrne `True`, če je `a` objekt razreda `A` ali katerega od njegovih potomcev.

Ker je `A` in `a.__class__` eno in isto in ker nov objekt razreda `A` konstruiramo tako, da pokličemo `A`, bomo dosegli enak učinek, če pokličemo `a.__class__`.

```
>>> b = a.__class__()
>>> b
<__main__.A instance at 0x013C7558>
```

Če bi vas kdo vprašal, kako napisati funkcijo, ki iz danega objekta naredi še en objekt istega razreda – ne da bi vedeli, kateri razred je to – bi znali, ne?

```
>>> t = 42
>>> u = t.__class__()
>>> u
0
>>> t = "xy"
>>> u = t.__class__()
>>> u
''
```

Trik vžge posebno dobro, če konstruktor razreda dopušča, da mu podamo originalni objekt. Tako lahko kloniramo objekte.

```
>>> u = [1, 2, 3]
>>> t = u.__class__(u)
>>> u.append(4)
```

```
>>> u
[1, 2, 3, 4]
>>> t
[1, 2, 3]
```

S podobnimi triki si pomaga tudi serializacija preprostejših tipov.

Razred je, vemo, objekt. Kot vsi objekti ima tudi razred polje `__dict__`, katerega vsebina so vsa polja in metode *razreda*, pa še nekaj vidnih in nekaj skritih atributov.

```
>>> class A:
...     def f():
...             pass
...
>>> A.__dict__
{'__module__': '__main__', '__doc__': None, 'f': <function f at
0x013C5230>}
```

Med skritimi atributi je ime razreda (`__name__`), in terka z razredi, iz katerih je razred izpeljan (`__bases__`). Tako lahko v programu izvemo vso hierarhijo razredov (nekaj, česar si programerji v C++ lahko le zaman želijo od njegovega omejenega RTTI).

V zvezi z objekti in razredi moramo omeniti še dve vdelani funkciji. Klic `dir(obj)` vrne približno isto reč, kot bi jo dobili z `obj.__dict__`, klic `type(obj)` včasih vrne `obj.__class__`. Včasih?!

```
>>> u = 42
>>> u.__class__
<type 'int'>
>>> type(u)
<type 'int'>
>>>
>>> a.__class__
<class __main__.A at 0x013B6FC0>
>>> type(a)
<type 'instance'>
```

Tole je žal nekaj, kar se vleče iz starejših verzij Pythona. Da bi bil tudi odgovor na zadnje vprašanje `<class __main__.A at 0x013B6FC0>`, moramo razred `A` izpeljati iz razreda `object`, s čimer dobimo razred novega sloga. Python 3.0 pa

razredov starega sloga nima več, zato je odgovor na zadnje vprašanje takšen, kot mora biti.

Kakšnega tipa je tip?

```
>>> type(42)
<type 'int'>
>>> int
<type 'int'>
>>> type(int)
<type 'type'>
>>> type(type)
<type 'type'>
```

Razumete? Objekt `42` je tipa `int`. Tudi tip `int` je objekt, in sicer tipa `type`. Objekt `type` pa je tipa `type`.

Moduli. Kot smo že povedali, so moduli dolgočasne reči. Modul je objekt tipa `module` in nima skoraj nič drugega kot slovar, `__dict__`, v katerem so shranjeni vsi objekti, ki jih modul vsebuje – spremenljivke, razredi, funkcije, drugi moduli...

Funkcije. O funkcijah je mogoče izvedeti kar precej. Vzemimo takšnole funkcijo.

```
>>> def f(a, b, c=2, d="X"):
...     print a, b, c, d
...
```

Najprej, funkcija je objekt. Objekt tipa `function`. Kaj pa takšen objekt zna?

```
>>> dir(f.__class__)
['__call__', '__class__', '__delattr__', '__dict__', '__doc__',
'__get__', '__getattribute__', '__hash__', '__init__',
'__module__', '__name__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__str__',
'func_closure', 'func_code', 'func_defaults', 'func_dict',
'func_doc', 'func_globals', 'func_name']
```

Najprej opazimo, da ima definiran operator `__call__`. Ko bi ga ne imel, funkcije ne bi mogli poklicati. Ima `__hash__`, torej jo lahko uporabimo kot ključ v slovarju in jo damo v množico. Res?

```
>>> {f: "x"}
{<function f at 0x013C50B0>: 'x'}
```

Res. Posebej zanimiva so polja, ki se začno s `func`. Polje `func_code` vsebuje kodo funkcije. Koda funkcije mora biti nekje shranjena, ne? In ni razloga, da je tolmač ne bi shranil nekam, kjer jo vidi tudi program, medtem ko se izvaja. `func_globals` je slovar globalnih spremenljivk, torej tisto, kar dobimo, če znotraj funkcije pokličemo `globals`. To je, seveda, slovar modula, v katerem je funkcija definirana, oziroma slovar globalnih spremenljivk, če je funkcija definirana v glavnem programu. Zanimivo je tudi polje `func__defaults`, ki vsebuje privzete argumente. Te lahko vidimo in celo spreminjamo.

```
>>> f(1, 2)
1 2 2 X
>>> f(1)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: f() takes at least 2 arguments (1 given)
>>> f.func_defaults = (3, 4, 5)
>>> f(1)
1 3 4 5
```

Ne pravim, da je to posebej koristno. Kot smo napovedali v uvodu v poglavje, se tule bolj ukvarjamo z zanimivim kot z uporabnim.

Sklad. Do tolmačevega sklada pridemo s funkcijo `stack`, ki se nahaja v modulu `inspect`. Rezultat klica je seznam objektov tipa `frame`. Vsak od njih vsebuje slovar lokalnih spremenljivk pripadajoče funkcije in slovar globalnih spremenljivk (ta je seveda isti za vse funkcije istega modula), prevedeno kodo funkcije, indeks ukaza, ki se izvaja v tem trenutku, številko vrstice, ki se trenutno izvaja, in nekaj polj povezanih z izjemami.

S pomočjo sklada lahko program izve vse o sebi, kar ni posebej zanimivo, ker pač nikogar ne zanima. Razen razhroščevalnikov. Razhroščevalniki za Python so pogosto pisani kar v Pythonu in temeljijo na opazovanju sklada, kakor ga vrne `inspect.stack`. No, včasih pa pride prav tudi pri "ročnem iskanju" kakih napak, na primer, ko želimo odkriti, kdo vse je poklical neko funkcijo, za katero se nam zdi, da je bila poklicana prevečkrat.

```
import inspect, random

def f(x):
    kdo_klice = inspect.stack()[1]
    print "%s:%s, funkcija %s" % kdo_klice[1:4]
```

```
    print "Koda: %s" % kdo_klice[4][0].strip()
    print "Lokalne spr: %s" % kdo_klice[0].f_locals
    print

def g(x, l=0):
    a = random.randint(0, 10)
    if l>5 or l>=2 and a>8:
        return f(x)
    elif a<3:
        h(x)
    g(x, l+1)

def h(x):
    return f(x), g(x, 3)

g(12)
```

Rezultat je lahko videti takole:

```
C:\D\x.py:20, funkcija h
Koda: return f(x), g(x, 3)
Lokalne spr: {'x': 12}

C:\D\x.py:14, funkcija g
Koda: return f(x)
Lokalne spr: {'a': 9, 'x': 12, 'l': 4}

C:\D\x.py:14, funkcija g
Koda: return f(x)
Lokalne spr: {'a': 9, 'x': 12, 'l': 3}
```

Nekateri zanimivejši moduli

Ker Python ni prav hiter jezik, je njegova uporabnost odvisna od modulov napisanih v Cju. Zato Python dobimo skupaj z "baterijami" (*batteries included*): kupom uporabnih modulov za najrazličnejše splošne namene. Takšne module bomo (nekoliko nenatančno) imenovali vdelani moduli. V tem poglavju bomo opisali najpomembnejše in najzanimivejše.

Za specifične naloge pogosto potrebujemo module, ki jih v distribuciji ni. Veliko uporabnikov poleg Pythona namesti vsaj še `numpy`, modul za premetavanje matrik. Odvisno od tega, kaj bomo z jezikom počeli, bomo nemara namestili še knjižnice za obdelavo slik, za video in zvok, enega ali več sistemov za uporabniške vmesnike in še kaj. Knjižnic, ki so brezplačno na voljo na spletu, je ogromno; karkoli se namenite početi, prijazna duša je verjetno že napisala ustrezno knjižnico in vam jo dala v brezplačno rabo.

Najprej pa si bomo ogledali nekaj vdelanih funkcij. Večino smo jih mimogrede že spoznali, tule omenjamo le tiste, ki jih še nismo, a bi bilo dobro, če bi jih.

Vdelane funkcije

Funkcija `callable(obj)` vrne `True`, če je objekt mogoče poklicati. Funkcijo `isinstance(obj, klass)` smo že spoznali, vendar ne bo škodilo, če ponovimo: z njo preverjamo, ali je `obj` objekt razreda `klass` ali katerega njegovih potomcev. Tipe preverjamo tako in samo tako. Kdor uporablja `obj.__class__`, kliče nadse prekletstvo. Funkcija `issubclass(klass1, klass2)` počne nekaj podobnega: `True` vrne, če je `klass1` posredni ali neposredni potomec `klass2`, ali pa je `klass1` enak `klass2`. Drugi argument je lahko tudi terka razredov. V tem primeru funkcija vrne `True`, če je `klass1` potomec kateregakoli izmed njih.

Z `execfile` izvedemo program, ki se nahaja v datoteki. Za razliko od izvajanja modulov, ki dobijo svoj imenski prostor, se ob klicu `execfile` koda izvede v imenskem prostoru klicatelja – kodi so na voljo vsi obstoječi razredi in tudi vse, kar stori `execfile`, ostane vidno. Z `eval` izračunamo vrednost izraza. Izraz je

lahko poljuben izraz v Pythonu, vsebuje lahko klice funkcij in podobno, ne more pa vsebovati ukazov, kot je `print`, ali definicij. S `compile` prevedemo niz v bajtno kodo za Pythonov navidezni računalnik. Takšen niz lahko vsebuje tudi celotne programe. V praksi funkcije `compile` ne uporabljamo pogosto; še celo `eval` je v običajnih programih bolj redek.

Funkcije `hasattr(obj, name)`, `getattr(obj, name[, default])`, `setattr(obj, name, value)`, `delattr(obj, name)` preverijo, ali ima objekt `obj` polje z imenom `name`, ga preberejo, nastavijo in pobrišejo.

Vdelani moduli

Med vdelanimi moduli je, za začetek, vse, kar je pričakovati od vsakega jezika. Funkcije za delo z nizi (veliko jih je sicer lažje dostopnih v obliki metod razreda `str`, ki mu pripadajo nizi), funkcije za delo s časom (trenutni čas, pretvarjanje sem in tja, štoparica), nekaj dodatnih tipov, ki temeljijo na vdelanih (npr. prednostna vrsta), modula z matematičnimi funkcijami za realna in kompleksna števila, moduli za nižjenivojsko delo z datotekami, procesi, nitmi... Vseh ne bomo naštevali, bralec jih bo že našel sam.

Regularni izrazi

Regularni izrazi v Pythonu niso vdelani v sam jezik, tako kot v Perlu in PHPju, saj gre vendarle za splošnonamenski jezik. Pač pa vdelani modul `re` vsebuje implementacijo regularnih izrazov po zgledu Perla s kar nekaj dodatki.

Regularne izraze lahko uporabljamo na dva načina. Modul vsebuje funkcije, kot je `re.search`, ki ji kot argument podamo regularni izraz in niz, v katerem naj ga išče, ali pa, denimo, `re.split`, ki dobi regularni izraz in niz, ki ga razbije na podnize, kjer je podani regularni izraz ločilo med njimi (podobno torej, kot metoda `str.split`, ki jo že poznamo, vendar z bolj zapletenimi ločili). Kadar uporabimo takšno funkcijo, mora Python prevesti regularni izraz v končni avtomat in ga uporabiti.

Če en in isti izraz uporabimo večkrat, se nam ga splača prevesti enkrat samkrat.

Funkciji `re.compile` podamo kot argument regularni izraz, pa ga prevede. Objekt, ki ga dobimo kot rezultat, ima enake metode kot modul (`search`, `split` in deset drugih), vendar z enim argumentom, namreč regularnim izrazom, manj.

Sintakse regularnih izrazov ne bomo opisovali, saj je taka kot drugod, omenimo le hvaležnosti vreden dodatek: poimenovanje skupin. V naslednjem regularnem izrazu nas zanimata dela, ki smo ju označili s krepkim tiskom.

```
r'<((span)|(div)) class="tv_clock">(\d\d?):(\d\d?)</span>\s*'
```

Večina drugih jezikov bi nas obsodila na preštevanje skupin: označena dela predstavljata četrto in peto skupino. Po morebitnih spremembah v prvem delu izraza, s katerimi bi dodali ali odvzeli kako skupino, bi se indeksa spremenila in skupine bi morali preštevati ponovno (ali pa bi na to celo pozabili). V Pythonovih regularnih izrazih lahko skupine poimenujemo.

```
r'<((span)|(div)) class="tv_clock">(?P<hour>\d\d?):(?
P<min>\d\d?)</span>\s*'
```

Ko kasneje zahtevamo vsebino teh skupin, lahko namesto indeksa uporabimo kar simbolično ime.

Ne spreglejte `r`-ja pred nizom. Z njegovo pomočjo se izognemo dvojnim levim poševnicam. Regularni izrazi so dovolj zapleteni tudi brez njih.

Regularne izraze lahko uporabljamo s funkcijama `search` in `match`, prva išče poljubno pojavitev podniza, druga zahteva, da se iskani podniz pojavi na začetku niza, po katerem iščemo. Rezultat iskanja je `None`, če iskanega podniza ni, sicer pa dobimo objekt tipa `MatchObject`. Tega lahko vprašamo po vsebini posameznih skupin ali celotnega podniza, ki je ustrezal regularnemu izrazu, in kje v preiskovanem nizu se nahajajo.

Funkcija/metoda `sub` zamenja vse podnize, ki ustrezajo regularnemu izrazu, s podanim podnizom. Ta se lahko sklicuje tudi na posamezne dele podniza, ki so ustrezale skupinam v regularnem izrazu.

Funkcija `split` razbije niz na podnize, ki jih ločuje podani regularni izraz.

Funkcija `findall` vrne seznam vseh podnizov danega niza, ki ustrezajo regularnemu izrazu. Če je regularni izraz zapisan tako, da vsebuje skupine, so

elementi vrnjenega seznama terke z deli podniza, ki ustrezajo posameznim podskupinam.

Lepša alternativa je `finditer`. Ta vrne generator, ki vrača objekte tipa `MatchObject`, kakršne smo opisali zgoraj.

Oglejmo si le en primer uporabe regularnih izrazov. Na enem od državnih tekmovanj iz računalništva je neka naloga od tekmovalca zahtevala poiskati vse "grbave" besede, ki se pojavijo v nekem besedilu. Grbava beseda je beseda, v katerih se vsaj dvakrat pojavi kombinacija velike črke, ki jih sledi mala črka.

```
import re
re_grbav = re.compile("([A-Za-z]*[A-Z][a-z]){2}[A-Za-z]*")
for w in re_grbav.finditer(besedilo):
    print w.group(0)
```

Regularni izraz razumemo iz drugih jezikov in orodij (ali pa tudi ne), bistvo pa je v `finditer`, ki sestavi generator. Generirani objekti `w` so tipa `MatchObject` in imajo metodo `group`, ki zna na različne načine vrniti posamezno skupino najdenega niza. V našem primeru smo z `group(0)` zahtevali celoten niz, ki ustreza regularnemu izrazu.

Če regularni izraz uporabimo le enkrat, ga ni smiselno prevajati. Tedaj kličemo modulovo funkcijo `finditer` in celotna rešitev naloge je

```
import re
for w in re.finditer("([A-Za-z]*[A-Z][a-z]){2}[A-Za-z]*",
besedilo):
    print w.group(0)
```

Serializacija

Python omogoča "serializacijo" objektov, se pravi shranjevanje v niz oziroma v datoteko, od koder ga lahko pozneje ponovno preberemo in skonstruiramo. Temu sta namenjena dva modula, `pickle` in `cPickle`. Prvi je pisan v Pythonu, drugi v Cju, zato je veliko hitrejši in podpira nekatere dodatne protokole. Python 3.0 ima le še drugega (a pod imenom `pickle`).

Serializacija ni preprosta zadeva. Python bo poskrbel, da se bo vsak objekt shranil le enkrat, tudi če ga "vsebuje" več različnih objektov. Tudi cikli – objekt `a` vsebuje

`b`, ta pa vsebuje `a`, ne bi smeli biti problem. Shranjevanje objektov razredov definiranih v Pythonu tako ne povzroča težav. Protokol, ki ga morajo podpirati objekti, ki jih je mogoče serializirati, pa je dokaj zapleten, zato razredi iz dodatnih modulov v Cju serializacijo podpirajo bolj redko. Preveč dela.

Naključje, kriptografija, razprševanje

Modul, ki dela naključne reči, temelji na algoritmu Mersenne twister, ki generira 53-bitna naključna števila s ciklom 2^{19937}-1. Poleg običajnih funkcij – `random` vrne naključno število med `0` in `1`, `randint(a, b)` pa naključno celo število med `a` in `b` vključno z `b` (tole pač ni indeksiranje!) - ima še par posrečenih dodatnih funkcij.

Funkciji `choice` podamo zaporedje in vrne naključno izbrani element. Funkcija `shuffle` naključno premeče elemente podanega zaporedja, `sample` pa naključno izbere podano število elementov podanega zaporedja.

Resnejšim uporabnikom bodo nemara prišle prav tudi funkcije za žrebanje naključnih spremenljivk iz kupa različnih porazdelitev – enakomerne, normalne, log-normalne, beta, gama, Gaussove, Weibullove ...

Naštete funkcije so funkcije modula, ki uporabljajo globalni generator naključnih števil. Včasih pa si želimo pripraviti svoj, privatni generator. V tem primeru sestavimo objekt razreda `Random`, ki ima enake metode kot modul.

S Pythonom dobimo tudi nekaj modulov povezanih z varnim prenosom podatkov: modula za izračun md5 in sha-1, pa modul za algoritem hmac, pa še enega s kupom različnih drugih razpršitvenih funkcij. Kriptografske funkcije (DES, RSA...) si moramo poiskati sami.

Iteratorji

Modul `itertool` vsebuje kup generatorskih verzij vdelanih funkcij. Tipična je `ifilter`, ki dela podobno kot `filter`, le da namesto seznama vrne generator. (Da, tako kot funkcija, ki smo jo v eni od nalog napisali tudi sami.) Podobni so `imap`, `islice` in `izip`. Katere funkcije nadomeščajo, je očitno. V Pythonu 3.0 nobene od teh funkcij ni več, saj tako delujejo že "osnovni" `filter`, `map` in `zip`.

Poleg teh vsebuje modul še nekaj drugih iteratorjev: takšnega, ki šteje do podanega števila, pa takšnega, ki vrača en in isti objekt in podobne.

Internet, splet, XML

Pythonu je priložen kup modulov za delo z različnimi internetnimi protokoli. Tako je trivialno pobirati elektronsko pošto (`poplib`, `imaplib`), pošiljati pošto (`smtplib`), brati datoteke s spleta (`urllib`, `urllib2`, `httplib`), uporabljati telnet (`telnetlib`) ali, za one iz prejšnjega stoletja, gopher (`gopherlib`).

Gre pa tudi v drugo smer: običajna distribucija že vsebuje različno zmogljive preproste strežnike za http (`BaseHTTPServer`, `SimpleHTTPServer`, `CGIHTTPServer`) in smtp (`smtpd`). Za pisanje CGI je na voljo še preprost modul za branje zahtev in modul za delo s piškotki.

Pri pisanju strežnikov si običajno namestimo boljše module, kot je razširitev za Apache (ApacheMod) ali strežnik CherryPy, PythonWin pa vsebuje tudi modul, s katerim lahko v Pythonu pišemo aktivne spletne strani (.asp) za strežnik IIS. S temi razširitvami postane programiranje spletnih strani v Pythonu podobno (le udobnejše) kot v PHP.

Za branje datotek XML sta na voljo oba razširjena standarda, DOM in SAX. Prvi prebere celotni XML v drevesno strukturo, drugega pa uporabljamo tako, da podamo funkcije, ki se prožijo ob posameznih dogodkih (začetek in konec oznake in podobno). Vse funkcije modulov DOM in SAX so takšne, kot jih zahteva standard in kot so na voljo tudi v drugih programskih jezikih, zato se jih bo bralec, ki se je s tem že ukvarjal, hitro navadil.

Dodatni moduli

Za vsako stvar, ki se je boste lotili, najverjetneje obstaja že pripravljen modul, pa najsi gre za branje oznak ID3 v datotekah .mp3 ali pa za tridimenzionalno grafiko. Module si bo znal poiskati vsak sam, povezava do precejšnjega repozitorija pa vodi z domače strani jezika. Tule jih bomo spet omenili le nekaj.

Numpy

Modul `numpy` služi delu z matrikami in linearni algebri (iskanje lastnih vrednosti in podobno). Modul je tako razširjen, da so za enega njegovih prednikov predlagali, naj se uvrsti kar med standardne, vdelane module. Benevolentni diktator se je temu uprl, češ da je modul prezapleteno vzdrževati, a dovolil, da so za udobje numpyja nekoliko prilagajali način, na katerega Python dela z rezinami.

Diktatorjeva previdnost se je izkazala za upravičeno. Omenjeni prednik se je imenoval `Numeric`. Ko je njegov razvoj zastal, ga je zamenjal `numarray`, ki je bil z njim samo delno združljiv. Čez nekaj let se je pojavil današnji `numpy`, ki je vseboval lastnosti obeh, tako da je bil z obema skoraj združljiv (zares pa z nobenim). Toliko, da boste, če naletite na koga iz te klape, vedeli, za kaj gre.

Prva prednost numpyjevih matrik pred seznami seznamov (seznamov seznamov) je v tem, da eno število v numpyjevi matriki zasede le toliko pomnilnika, kot ga mora. Če so v matriki števila dvojne natančnosti (numpy sicer podpira najrazličnejše matrike, od booleovih in enobajtnih do `long int` in `double`) bo vsako zasedlo le toliko, kolikor je na danem sistemu `sizeof(double)`.

Temu primerno večje matrike lahko shranimo v pomnilnik in temu primerno hitrejše so operacije z njimi. Matrike so lahko poljubno dimenzionalne. Njihovo rezanje je preprosto, hitro in ne zasede nič dodatnega pomnilnika (če sami ne zahtevamo drugače). Če v dani matriki izberemo le stolpce od drugega do petega in vrstice od sedme do dvanajste, bo `numpy` novo matriko sestavil kar tako, da bo "kazala" na prvotno, ne da bi prepisovala podatke.

Matrike lahko preobračamo, preračunavamo, množimo, računamo vsote po vrsticah/stolpcih v poljubni dimenziji... Z modulom za linearno algebro lahko računamo inverze in determinante, rešujemo sisteme enačb in različne dekompozicije, iščemo lastne vrednosti, pa še marsikaj, česar avtor knjige, nematematik, ne razume.

Numpy lahko dobimo tudi kot del širšega paketa SciPy (izg. *saj paj*). Ta vsebuje vse, česar si poželi srce znanstvenika: statistiko, različne optimizacije, numerično integracijo, Fourierovo transformacijo in procesiranje signalov in slik...

Grafika, video, zvok

Knjižnica PIL (Python Imaging Library) sicer ni tako vseprisotna kot numpy, vendar si je pridobila mesto standardne knjižnice za obdelavo slik. Osnovni objekt knjižnice je `Image`, slika, ki jo bodisi ustvarimo na novo bodisi preberemo iz datoteke. Podprte so mnoge različne vrste datotek s slikami, pa tudi različni zapisi slik (RGB, RGBA, CMYK ...). Slike lahko obdelujemo tako, da spreminjamo barve (npr. več zelene in manj modre), ločujemo slike na posamezne barvne kanale, prekrivamo različne slike ali nad njimi izvajamo različne druge operacije (iz dveh slik sestavimo novo tako, da jemljemo, denimo, svetlejšo točko iz vsake), po njih rišemo in pišemo... na koncu pa jo ponovno shranimo v datoteko.

PIL ve tudi za druge popularne module. Sliko je preprosto prenesti v `numpy`, kjer jo obdelujemo kot tridimenzionalno matriko, ki vsebuje jakost vsake barvne komponente (npr. R, G in B) za vsako točko slike. Ali obratno, matriko iz `numpyja` lahko spremenimo nazaj v sliko.

Poleg tega vsebuje PIL module za pretvorbo slike v objekte, s kakršnimi predstavljata slike priljubljena modula za sestavljanje uporabniških vmesnikov Tk in Qt (glej spodaj). V okolju MS Windows nam je na voljo še možnost pretvorbe slike v standardno bitno sliko in shranjevanje in branje slike z odložišča.

Impresivna knjižnica matplotlib služi preprostemu risanju zapletenih grafov po zgledu teh iz komercialnega programa Matlab. Matplotlib pozna klasične krivuljne grafe, grafe v polarnih koordinatah, histograme, tortne diagrame (*pie chart*), razsevne diagrame (*scatter plot*), grafe nivojnice (*contour plot*). Graf si je mogoče ogledati na zaslonu ali ga shraniti v formatu png ali kakem vektorskem formatu (eps, pdf, svg).

Igranju z zvokom in videom sta namenjeni knjižnici PyMedia in PyGame. Drugi dobesedno (čeprav vsebuje tudi koristne stvari za delo z videom), prvi malo manj. Dobiti je mogoče tudi druge pripomočke za delo z videom, denimo modul VideoCapture za zajemanje slike s kamere. Rezultat dobimo v obliki objekta `Image` modula PIL, ki ga lahko tam še malo obdelamo, nato spremenimo v Qt-jev `QImage` ... pa smo si pripravili preprosto aplikacijo, s katero, recimo, zajemamo sliko z varnostne kamere.

Grafični uporabniški vmesniki

Skriptni programski jeziki so zelo praktični za pisanje grafičnih uporabniških vmesnikov (GUI), saj ti navadno ne zahtevajo hitrosti, temveč fleksibilen jezik. Python je za to kot nalašč, zato knjižnic za sestavljanje uporabniških vmesnikov zanj kar mrgoli.

Ob Pythonu se nam navadno namesti Tcl/Tk, ki je predstavljen kot "standardni" sistem za pisanje GUIjev v Pythonu. Zakaj, pravzaprav ne vem: *de facto* standard je wxPython, ki ovija wxWidgets (nekdaj wxWindows, a preimenovan zaradi Microsoftovih groženj s tožbo zaradi kraje imena(!)).

Po mnenju avtorja se nič ne more kosati s Qt oziroma pripadajočim modulom za Python, PyQt. Qt je v resnici knjižnica za C++ in pisanje uporabniških vmesnikov s PyQt je komaj kaj manj duhamorno, kot če bi jih pisali v C++. (Pretiravam. Ampak prijetno pa res ni.) Če pa si pripravimo primeren vmesni nivo, v katerem ovijemo PyQtjeve razrede, ki jih najpogosteje uporabljamo, v bolj "pythonovske" razrede, pa dobimo zelo močno okolje, s katerim dobimo aplikacije, ki v MS Windows izgledajo, kot aplikacije v MS Windows, na Mac OS X pa tako kot v Mac OS X.

Poleg naštetih okolij, ki delujejo na različnih platformah, so na voljo tudi knjižnice za specifične platforme, kot so GnomePython in PyKDE za Gnome in KDE ter Win32All za MS Windows. Win32All je bolj znan pod drugim imenom, imenom razvojnega okolja, ki je povezano z njim: PythonWin. Ker večina bralcev uporablja ta operacijski sistem, je Win32All vreden posebne obravnave.

Win32All

Win32All bi lahko uporabljali kot okolje za sestavljanje uporabniških vmesnikov. A tega navadno ne počnemo – druga okolja so veliko prijetnejša. Zato pa nam Win32All omogoča še veliko drugega: sistemske klice MS Windows in klice različnih z njim povezanih podsistemov in knjižnic (direct sound, exchange, ActiveX...)

Modul `win32api` omogoča klicanje funkcij iz, kot pove ime, APIja operacijskega sistema. Zbirka je pestra. S `ChangeDisplaySettings` nastavljamo ločljivost

zaslona, `CopyFile` prepiše datoteko, `InitiateSystemShutdown` ustavi sistem (`AbortSystemShutdown` pa ustavi ustavljanje), `FindExecutable` poišče program, ki je povezan s podano datoteko (običajno glede na končnico datoteke), `LoadLibrary` naloži dinamično knjižnico (dll), `RegCreateKey` piše v sistemski register, `TerminateProcess` ustavi proces...

Ostali moduli so specializirani za posamezne dele sistema. Modul `win32file` vsebuje kup funkcij za delo z datotekami, z `win32clipboard` lahko beremo iz in pišemo v odložišče, `win32gui` omogoča odpiranje oken in delo z njimi (denimo nastavljanje menujskih vrstic), `win32help` je namenjen delu s sistemom WinHelp in starejšim HtmlHelp, `win32inet` podpira internetne protokole http, ftp in gopher, z `win32process` upravljamo s procesi (odpiramo nove procese in niti, nastavljamo prioritete in podobno) in tako naprej.

Za mnoge izmed gornjih modulov Python že nudi svoje, boljše module (tipičen primer modula, ki ga ne potrebujemo je `win32inet`), tako da nam ni potrebno uporabljati sistemskih. Druge uporabljamo redko – skripte, ki znajo pisati v odložišče, tipičnemu programerju v Pythonu niso vsakdanji kruh. Zato pa je zelo zanimiv in uporaben modul `win32com`, ki omogoča delo z ActiveX. V obe smeri. Takole bere podatke iz Excelove datoteke.

```
import win32com.client
app = win32com.client.Dispatch("Excel.Application")
app.Visible = True
wb = app.Workbooks.Add("c:\\d\\zbirka.xls")
ws = app.Worksheets("Dokumentarci")
for row in ws.UsedRange.Rows:
        print row.Cells(1, 2).Value
```

Tule pa je primer strežnika. Ko zaženemo spodnjo skripto, v sistemu registrira nov objekt COM z imenom *Python.Fibonacci*. Objekt vsebuje eno samo funkcijo, `f`, ki vrne n-to Fibonaccijevo število (vem, prav pogrešali ste jo že).

```
import win32com.server

class Fibonacci:
    _public_methods_ = ["f"]
    _reg_clsid_ = "{732509D9-B57D-44AB-A0F0-C4FDBFCB8869}"
    _reg_progid_ = "Python.Fibonacci"
    _reg_desc_ = "Python Fibonacci Computer"
```

```
    def f(self, n, a=1, b=1):
        for i in range(n):
            a, b = b, a+b
        return a

if __name__=="__main__":
    import win32com.server, win32com.server.register
    win32com.server.register.UseCommandLine(Fibonacci)
```

Takšen objekt lahko nato uporabljamo v vseh drugih jezikih, ki podpirajo Active X. Tule, recimo, je makro za Excel, ki pokliče našo funkcijo v Pythonu, da s Fibonaccijevimi števili zapolni prvih deset vrstic prvega stolpca.

```
Sub zapolni()
    Set fibo = CreateObject("Python.Fibonacci")
    For Row = 1 To 10
        Cells(Row, 1) = fibo.f(Row)
    Next
End Sub
```

Kdor je kdaj programiral kaj podobnega – namreč odjemalec ali strežnik za ActiveX – v C, bo znal ceniti preprostost gornjih programov.

Python lahko registriramo tudi kot enega od jezikov, v katerem lahko pišemo strani ASP za strežnik IIS. Za registracijo je potrebno na spletnem strežniku pognati skripto, ki se pri privzeti namestitvi Pythona 2.5 nahaja v imeniku *c:\python25\lib\site-packages\win32comext\axscript\client\pyscript.py*. Ko to storimo, v straneh ASP v glavo dodamo oznako `<%@ LANGUAGE=Python%>` in že lahko strani pišemo v Pythonu, na podoben način kot jih ob privzetih nastavitvah pišemo v Visual Basicu. Zaradi lastnosti Pythona, denimo obveznega zamikanja, ASP v Pythonu sicer ne podpira vsega, kar lahko počnemo z ASP v Visual Basicu (omejitve so pri uporabi `if` in `for` za vključevanje in ponavljanje delov dokumenta, napisanih v HTML), vendar utegne biti programiranje ASP v Pythonu vendarle bistveno preprostejše kot v Visual Basicu.

Filozofija in oblikovanje kode

Začnimo poglavje s pirhom (*easter egg*).

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do
it.
Although that way may not be obvious at first unless you're
Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Osnovno spoznanje, na katerem temelji jezik, je, da programsko kodo večkrat beremo kot pišemo, branje kode pa je težje kot pisanje (nekateri s tem razlagajo dejstvo, da večina programerjev o večini svoje kode meni, da je skrpucalo, ki bi ga bilo treba napisati znova: ni, le koda se zdi, ko jo pišemo, preprostejša kot takrat, ko jo beremo).

Odtod geslo "*There should be one-- and preferably only one --obvious way to do it.*" Tako bodo vsi programerji pisali podobno kodo in jim tudi branje tuje kode ne bo povzročalo težav. Ker je "edini" način tudi očiten in eleganten, se ob tem nihče ne bo čutil utesnjenega. To načelo je posebej dobrodošlo pri delu v skupini, kjer lahko

isto funkcijo programira in dopolnjuje več ljudi, pa med njihovim pisanjem ne bo večjih slogovnih razlik.[36]

Poleg teh načel je van Rossum, avtor Pythona, sestavil tudi priporočila za oblikovanje kode.

- Zamiki naj bodo veliki štiri presledke. Tabulator je odsvetovan, mešanje tabulatorjev in presledkov pa sploh.
- Vrstice naj ne bodo daljše od 79 znakov. Kadar lomimo vrstico v več vrstic, uporabimo, če je le mogoče, oklepaje. Če je vrstica v resnici, denimo, seznam, ga lahko brez težav pišemo v več vrsticah, saj bo tolmač opazil odprte oklepaje. Če oklepajev ni, jih lahko navadno brez škode dodamo, namreč okrogle.
- Med definicijami razredov izpustimo dve vrsti, med funkcijami pa po eno. S praznimi vrsticami znotraj funkcij varčujemo.
- Module praviloma uvažamo na vrhu datoteke, vsakega v novi vrstici. (Avtor knjige se bolj ali manj drži vseh pravil razen tega. Ne gre.)
- Raba presledkov:
    - Presledkov ne pišemo za oklepaji in pred zaklepaji, torej `f(1, 2)` in ne `f( 1, 2 )` ali `f (1, 2)`.
    - Presledke pišemo za vejico, pred njo pa ne.
    - Presledke pišemo pred in za operatorji (=, +=, ==, <, > ...), razen v poimenskih argumentih, kjer pišemo `f(a=1)` in ne `f(a = 1)`
    - Za dvopičjem gremo v novo vrstico. Python sicer dovoljuje pisanje v eni vrstici, če, denimo, `if`u sledi en sam stavek, npr. `if a < 0: a = 0`, vendar se temu pisanju izogibamo, ker je nepregledno.
- Imena:
    - Imena spremenljivk in funkcij pišemo z malimi začetnicami. Če je sestavljeno iz več besed, jih lahko ločimo s podčrtaji ali pa

36 Nasprotni temu so jeziki po načelu Tim Toady (TIMTOWTDI, There is more than one way to do it), kjer je kodo mogoče napisati na sto in en način, cena za to pa je, da se moramo pri branju zato pogosto spopasti s kodo, ki je napisana bistveno drugače, kot bi jo pisali sami – pač na enega od stotih ostalih načinov.

nadaljnje besede začenjamo z veliko začetnico, ne pa oboje.

- Imena modulov pišemo z malimi črkami.
- Imena razredov pišemo z veliko začetnico. Če je sestavljeno iz več besed, tudi vse naslednje besede začenjamo z veliko začetnico, npr. `EnhancedBasket`.
- Podčrtaj na začetku imena nakazuje, da gre za lokalni objekt, ki naj ga drugi puste pri miru.

Namen teh pravil je pomagati programerjem pisati čitljive programe, ki bodo poleg tega videti podobno kot programi njihovih kolegov. Vsakemu posebej pa je prepuščeno, ali se jih bo držal ali ne.

Kaj vse smo izpustili

Knjigo, ki bi povedala vse o Pythonu, bi bilo seveda mogoče napisati. Ne bi pa bi je bilo mogoče prebrati. Niti ne bi imelo smisla. Za konec pa vendarle omenimo pomembne teme, ki smo jih prezli.

Za razrede novega sloga smo povedali, kaj prida pa bralec ni izvedel o njih. Omenili smo, da omogočajo drugačne mehanizme nastavljanja in branja atributov (*getter*, *setter*). Podobnih detajlov je še veliko, vendar se jim nismo posvečali.

Funkcije in metode imajo lahko dekoratorje, nekakšne ovojnice, ki prestrezajo njihove klice. Čeprav so nedvomno uporabne, po avtorjevem mnenju ne prispevajo ravno h konceptualni preprostosti jezika. Iz podobnih razlogov smo zamolčali tudi stavek `with`, s katerim lahko ustvarjamo kontekste.

Tema, ki bi bila vredna obravnave, ker bo bralcu gotovo prišla prav, je povezovanje s funkcijami, napisanimi v C oz. C++. Za to lahko uporabimo eno od mnogih temu namenjenih orodij (swig, sip...), ki s pomočjo datoteke z definicijami, ki jo moramo pripraviti, sestavijo vmesno kodo za izvoz Cjevskih funkcij ali celo C++ovskih razredov v Python. Vsaj za začetek pa bralcu priporočam, da takšno kodo sestavi ročno, saj se bo ob tem veliko naučil o tem, kako Python deluje.

Resnično skopi smo bili v poglavju o dodatnih modulih. A drugače ni šlo, saj je zgolj te teme za knjigo ali dve. Ali osem.

O tem, kako zapakirati program v Pythonu, nismo govorili. Tolmača je mogoče vključiti v program v Cju, tako da končni uporabnik dobi le datoteko .exe, ki pa v resnici izvaja program, ki smo ga delno ali v celoti napisali v Pythonu. Prav tako bralec ni izvedel ničesar o tem, kako shranimo module, ki smo jih napisali, v pakete, primerne za shranjevanje v repozitorijih po zgledu drugih jezikov.

Pa tudi vse, kar smo povedali, smo skoraj brez izjeme povedali površno. Najboljše bralca šele čaka.