Univerza v Ljubljani

Fakulteta za računalništvo in informatiko

Janez Demšar

Python za programerje

Druga, dopolnjena izdaja

Ljubljana, 2012

CIP - Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

004.43(075.8)

DEMŠAR, Janez, 1971-
Python za programerje / Janez Demšar. - 2. popravljena in dopolnjena izd. - Ljubljana : Fakulteta za računalništvo in informatiko, 2012

ISBN 978-961-6209-79-3

260785920

Recenzenta: prof. dr. Blaž Zupan, doc. dr. Tomaž Dobravec
Založnik: Založba FE in FRI, Ljubljana
Izdajatelj: UL Fakulteta za računalništvo in informatiko, Ljubljana
Urednik: mag. Peter Šega

Natisnil: KOPIJA Mavrič, Ljubljana
Naklada: 150 izvodov
2. popravljena in dopolnjena izdaja

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 samo, ko to želimo, sicer pa se le tiho sukajo v ozadju.

Slabost Pythona je hitrost (bolj pošteno: počasnost) izvajanja. Ker se (neopazno) prevaja v kodo za navidezni stroj, je hitrejši od tistih 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. 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 na 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. Python je zato široko uporaben: v njem so pisane mnoge sistemske skripte za Linux, v njem je vedno več spletnih aplikacij (med njimi je najbrž najopaznejši YouTube), Python je bil dolgo 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. Ko je nekdo na nekem forumu napisal, da je jezik (zaradi dinamične tipiziranosti) videti nezanesljivo in v njem ne bi programiral kode za krmiljenje satelitov, mu je nekdo odgovoril, da je že prepozno, saj v Pythonu programira kodo za krmiljenje satelitov.

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 z ukazom `import` uvozili modul, ne da bi prej vsaj z besedico nakazali, da moduli v Pythonu sploh obstajajo. Od bralca pričakujemo, da bo take reči v osnovi razumel, 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 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) ali preklinjati (če niso). Pa naj.

Ob tem, kar naj bi bralec že znal, se torej ne bomo dolgo zadrževali. 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.

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 bolj zabavno.

Ob drugi izdaji

Druga izdaja knjige temelji na Pythonu 3.2. V dvajset letih, kar jezik obstaja, so bile nove različice Pythona kvečjemu minimalno nezdružljive s predhodnimi, različica 3.0 pa je pometla vse, kar se je v jeziku izkazalo za zgrešeno ali pa je bilo že nadomeščeno z boljšim, vendar je staro zaradi združljivosti še vedno kazilo jezik. Veliko sprememb je začetniku neopaznih, resnejši programer pa jih je opazil, a tudi razumel in pozdravil. V knjigi bomo razlike omenili, kadar se nam bo zdelo potrebno, vedno pa ne.

Čeprav spremembe v jeziku niti niso tako velike, kot smo se bali, ko so se pripravljale, je knjiga napisana praktično znova. Če česa nisem spremenil zato, ker je bilo potrebno spremeniti, sem spremenil zato, ker mi ni bilo (več) všeč.

Že v prvem poglavju je bilo potrebno dodati nova razvojna okolja in napraviti knjigo manj obrnjeno k MS Windows, saj se s klopi predavalnic sveti vedno več obgriženih jabolk, ki skupaj z Linuxom odrivajo MS Windows v manjšino. Poglavji o osnovnih podatkovnih tipih in kontroli toka sta ostali dokaj nedotaknjeni. Sestavljene podatkovne strukture so doživele revolucijo; namesto tipov `str` in `unicode` imamo zdaj `bytes` in `str`, ki ju je bilo potrebno opisati na novo. Rezultat te spremembe je tudi bistveno lažje delo z datotekami in vrstni red tem v prejšnji knjigi je postal neustrezen … in tako se podre celotno poglavje. Poglavje o funkcijah je bilo žrtev mojega nezadovoljstva s tistim iz prejšnje knjige. Podobno se je zgodilo naslednjemu: kaj je spremenljivka, znam zdaj povedati boljše, zato sem tudi moral povedati boljše in celo s slikami (za katere sem bil v prvi izdaji odločno prelen). S poglavjem o razredih že prej nisem bil zadovoljen, novi Python pa jih je dovolj prečistil, da se je dalo vse povedati veliko jasneje kot prej. Poglavje o funkcijskem programiranju je temeljilo na zastarelih `map`, `filter` in `reduce`, ki jih v besedilu o novem Pythonu skoraj ni več spodobno omenjati, zato je bilo potrebno snov predstaviti iz smeri izpeljanih seznamov oziroma množic. V poglavju o dodatnih modulih je bilo potrebno vsaj približno posneti, kar je novega v zadnjih petih letih. Poglavje o introspekciji pa je postalo preprosto odveč; vse, kar je vsebovalo uporabnega, se je preselilo v druga poglavja.

Od stare knjige sta, z malo pretiravanja, ostala le ideja in naslov.

Hvala

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

In Ireni, ki potrpežljivo prenaša, da je njen mož stalno na službeni poti v delovni sobi, za računalnikom. (Tako sem napisal ob prvi izdaji; še vedno velja.) 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 hkrati s sistemom. Preverite le, da imate različico 3.0 ali novejšo, saj s starejšo nekateri primeri iz knjige delujejo drugače, nekateri pa sploh ne. 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. Dobimo ga vsaj 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 z imenom *mojprogram.py* najlažje poženemo s `python mojprogram.py`.

Nekoliko udobnejša alternativa je ipython. Ta ima lupino, podobno tisti v programu Mathematica, ki oštevilčuje vhodne ukaze in shranjuje rezultate izračunov, tako da jih lahko uporabljamo tudi kasneje. Poleg tega ponuja pomoč pri vpisovanju ukazov, kot je dopolnjevanje s pritiskom na tabulator in podobno. Med njegovimi posebnostmi je vdelana podpora za vzporedno računanje. Okolje je zelo uporabno, kot razvojno okolje pa še vedno prešpartansko; za začetnika še toliko bolj.

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.

PyCharm je podoben Eclipsu, vendar že v osnovi napisan za Python in ponuja vse, kar ponujajo sodobna razvojna okolja, od orodij za predelavo (*refactoring*), do orodij za testiranje (*unit test*), merjenje pokritosti kode s testi (*coverage*) in povezave s sistemi za nadzor različic. Žal ga lahko brezplačno uporabljamo le trideset dni, vendar razvijalci ne skoparijo z licencami za razvijalce odprtokodnih

programov in za izobraževalne ustanove. Kdor s Pythonom misli resno, naj si ga vsekakor ogleda.

Za začetnike je primernejše preprostejše okolje. 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 je prikladen 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.

S Pythonom se je mogoče pogovarjati: v konzolo lahko vtipkavamo kodo in Python jo bo izvajal. Na ta način lahko vtipkamo in izvedemo karkoli, od preprostega `print`a do definicije funkcij ali celo razredov. To je uporabno tako za začetnika, ki lahko na ta način sproti preskuša svoje stvaritve, kot za razvijalca, ki bi rad odtipkal kak čuden kos kode, preden ga vnese v program, ali pa po kosih preskušal regularne izraze.

Razvojna okolja pogosto omogočajo tudi, da poženemo program, ki definira funkcije, potem pa se v konzoli igramo z njimi in jih kličemo. PyCharm je tu nekoliko neroden; med različnimi zasilnimi rešitvami je najboljša, da v *Run/Edit Configurations* pod *Intepreter options* za posamezen program ali kar ves projekt dodamo *–i*.

Tem, ki so se odločili za PythonWin, povejmo, da bo ta postal prijaznejši, če v *View* / *Options* / *General Options* obkljukajo *Dockable windows* in ga ponovno zaženejo.

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 razvojno okolje, ki si ga je izbral. Program bo vpisal v urejevalnik, ga shranil (to v PyCharmu ni potrebno, v ostalih pa le prvič, toliko, da mu določimo ime; kasneje se program sam shrani, ko ga poženemo) in ga pognal tako, da se bo po koncu "izvajanja" pojavila ukazna vrstica, v kateri bo lahko preskušal funkcijo.

V PythonWinu se navadite bližnjice: *Ctrl-R*. Ob tem bodite posebej pozorni še na sintaktične napake, saj jih javlja zelo diskretno: ob poskusu zagona programa s sintaktično napako se PythonWin neopazno pritoži v statusni vrstici in postavi kurzor na mesto napake.

V PyCharmu in drugih si sami poiščite bližnjici za izvajanje programa v trenutno odprtem oknu. Razpored tipk se razlikuje od sistema do sistema, pa tudi različne sheme si lahko izbiramo.

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

Ko smo že pri staroselcih in emacsu: da, tudi emacs je za začetek čisto primerno okolje za Python. Seveda le za tiste, ki so že vajeni jahati to zverino. Ostali naj se raje ne učijo dvojega hkrati.

Kakorkoli že izvedemo gornji program: program vsebuje definicijo funkcije, zato je rezultat njegovega izvajanja definirana funkcija. V večini okolij lahko dosežemo, da nam po izvajanju programov ostane konzola, v kateri lahko podebatiramo s Pythonom. V knjigi bo takšnih dialogov veliko. Prepoznali jih boste po `>>>` ali `...` na začetku vrstic, ki jih vtipkavamo; sledijo jim, seveda, Pythonovi odgovori. Konzola je tipično na dnu okna; v PythonWinu in PyScripterju je tam vedno, v PyCharmu moramo, kot smo že omenili, pod *Intepreter options* v *Run/Edit Configurations* dodati opcijo –i (po čemer ne smemo pozabiti ročno zapreti konzole po vsakem zaganjanju programa, sicer se bodo začele nabirati!).

Program smo torej zagnali, funkcija je definirana in prišli smo do ukazne vrstice. Zdaj funkcijo pokličimo.

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

Konzole so navadno narejene tako, da smemo `print` izpustiti, zato ga tudi bomo. 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 klica `print` ne smemo izpuščati, sicer se bo funkcija poklicala, izpisalo pa se ne bo nič). Če storimo tako, program ne bo le definiral funkcije, temveč jo tudi poklical in izpisal njen rezultat.

Mimogrede, s klici iz ukazne vrstice, pogosto preskušamo tudi tuje funkcije. Če se ne morem spomniti vrstnega reda argumentov funkcije, ki sem jo nazadnje uporabil predlansko pomlad, mi ni potrebno brskati po dokumentaciji, temveč jih z ugibanjem dobim 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
```

(Namig: pomaga pa tudi, če iz ukazne vrstice pokličemo `help(math.log)`).

Zdaj pa funkcijo `fibonacci` nekoliko spremenimo. Dodajmo možnost, da ob klicu nastavimo prva dva člena, 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
```

Napočil je čas, da se prvič 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 naset jejo goro diagnostičnih izpisov. Večina razvojnih 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-to 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, imamo še nekaj boljšega: ukazno vrstico. 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'
```

Osnovne poteze jezika

Knjiga je namenjena bralcu, ki že zna programirati, kar navadno pomeni, da pozna C, C++, C# ali Javo. Na začetku bo moral pozabljati kot pomniti. Za primer 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.[1] 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 kode doslej ni zamikal, ga bo Python (edino pravilno!) prisilil, da začne.

Glavi definicije funkcije sledi dvopičje in v naslednji vrstici smo povečali zamik. Podobno je z zanko `for`: glava, dvopičje, zamik. In vedno bo 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 ga človek navadi pisati.

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 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.

---

[1] Če res ne more brez, naj vtipka `from __future__ import braces`.

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 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 to v resnici storimo le redko, saj poševnica ni potrebna, če je zaradi odprtih oklepajev jasno, da se vrstica nadaljuje. In kadar je vrstica dolga, skoraj vedno vsebuje tudi nek odprt oklepaj.

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

Zamik druge in morebitnih naslednjih vrstic je lahko poljuben, nič pa ni narobe, če jih poravnamo lepo.

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 splošnejšega.

Spremenljivk ne deklariramo, niti zanje ne zahtevamo ali sproščamo prostora (`new`, `delete` in podobne nadloge). Podobno ne moremo navesti tipov argumentov funkcij. Funkcija pač 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, saj do onega mesta niti ne more vedeti, da je s tem kaj narobe. Zalomilo se je šele v tretji vrstici, kjer naj bi k številu 4 dodal niz `"a"`. Kot *močno tipiziran* se ni poskušal izmazati s `"4a"` (kot bi storil JavaScript, kjer je glede na običajno rabo 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. Se vidimo v naslednjem poglavju, prihodnji teden. Pa pazite, da se ne zmotite.

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

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

   Lepše je tako:

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

   Ljubitelji rekurzije so morda napisali tole:

```
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, je napisal preprosto

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

Komur diši funkcijsko programiranje, pa se bo razveselil naslednje rešitve.

```
from operator import mul
from functools import reduce
def fakulteta(n):
    return reduce(mul, range(1, n))
```

ki je v resnici isto kot

```
def fakulteta(n):
    return reduce(lambda x,y: x*y, range(1, n))
```

Naj tisti, ki jim to kaj pomeni, že takoj izvedo, da ima Python tudi lambda-funkcije (oziroma, priznajmo, nek slab izgovor zanje).

Besednjak

Osnovni podatkovni tipi

Python pozna cela in realna števila, `int` in `float`. Njuni imeni ne ustrezata Cjevskim: `int` je lahko poljubno veliko število, ne le $\pm 2^{31}$ ali $\pm 2^{63}$ ali kaj podobno omejenega. Pythonov `float` pa je v resnici predstavljen s tistim, čemur C pravi `double`. Tako imajo tipi v Pythonu lepa imena; nekomu, ki ne pozna Cja, ime *double* itak ne bi pomenilo ničesar in bi si ob njem verjetno predstavljal par števil.

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 modul `cmath` z osnovnimi funkcijami (`sin`, `cos`, `exp`...) na kompleksnih številih.

Podatkovni tip `bool` je izpeljan iz `int`. Tako je iz historičnih razlogov; tipa `bool` Python včasih ni poznal, zato so logične operacije vračale `0` in `1`. Pa tudi praktično je, saj ga lahko tako uporabimo tudi na vseh mestih, ki sicer zahtevajo `int`, na primer indeksiranje. V Pythonu imena konstant navadno začenjamo z velikimi začetnicami, torej se konstanti tipa `bool` imenujeta `True` in `False`.

Konstanta `None` pomeni nič. Vračajo jo, denimo, funkcije, ki ne vračajo ničesar. 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 ni prav pogosto. Pač pa navadno razpakiramo terko. Terka je, če 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: Python najprej sestavi terko (`7, 12, "Tine"`), nato pa jo razpakira v `a`, `b` in `c`.

Vedoč to se vprašajmo: kako v Pythonu zamenjamo vrednosti dveh spremenljivk?

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

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

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

Zdaj pa vemo, kaj je ta stavek v resnici pomenil.

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

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

Vendar ni, da bi to potrebovali ravno vsak dan. Dvakrat letno pa že.

Matematične operacije so takšne kot drugod: +, -, *, /, poleg teh pa imamo še //. Deljenje z `/` vedno vrača realna števila (to je, aleluja!, novost iz Pythona 3.0; prej smo se ničkolikokrat zmotili, ko smo pred kakim deljenjem pozabili spremeniti kako celo število v realno), operator `//` pa predstavlja celoštevilsko deljenje – tudi, kadar dobi kot argument realna števila.

```
>>> 1//2
0
>>> 4.5//1.4
3.0
```

Operator `**` pomeni potenciranje. Tako osnova kot potenca sta lahko neceli števili, le da mora biti pri neceli potenci osnova pozitivna (račun `(-1)**0.5` nas bi pripeljal v kompleksna števila, kadar želimo delati z njimi, pa moramo to zahtevati bolj eksplicitno). `%` računa 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 še prazno, kar je nič in kar je nikakršno na kak drug način. Vsa ostala števila, nizi, seznami in drugi objekti se štejejo za resnične.

Logični izraz se, tako kot v skoraj vseh drugih jezikih, računa od leve proti desni (pri čemer ima `and` seveda prednost pred `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 = "Miran"
>>> "Vaše spletno mesto je obiskal " + (ime or "Anonymous")
'Vaše spletno mesto je obiskal Miran'
>>>
>>> "Vaše spletno mesto je obiskal " + (ime or "Anonymous")
'Vaše spletno mesto je obiskal Anonymous'
```

To bi pa utegnilo priti kje prav, ne?

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 odlično znašli in smo namesto `a ? b : c` pisali kar `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 nekoliko zmedenih and-or izrazov se je nekoliko zmedlo veliko ljudi in

pisalo kodo, ki je samo nekoliko delovala, odkrivanje hroščev zaradi napak v takšnih izrazih pa je bil nekoliko 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 operator vendarle dodal, čeprav zanj ni bilo dobre sintakse. Zdaj je nekoliko japonska:[3]

```
b if a else c
```

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

Ob tem se `b` 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, se pravi `==` za enakost in `!=` za neenakost. Primerjamo lahko marsikaj, ne le števil, temveč tudi nize, zaporedja, tipe... 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, pač pa nizanje elegantno poenostavi preverjanje, ali je vrednost določene spremenljivke znotraj meja, npr. `if mn <= a <= mx` ali `if "a" <= c <= "z"`.

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`. Večini zanj ni mar in primeri, ko bi ga dejansko potrebovali, so navadno zgledi slabo

---

[3] Zakaj naj bi bilo to japonsko, nimam pojma. Tako so ocenili moji študenti, konkretno, eden, ki v resnici zna malo japonsko, zato mi v pomanjkanju drugih dokazov ne preostane drugega kot verjeti na besedo.

zasnovanega programa. Redno ga uporabljamo le za primerjanje z `None`, v stavkih kot `if a is None` ali `if a is not None`, ki je ekvivalentna `if not a is None`. A tudi tu bi operatorja `==` in `!=` naredila natanko isto kot `is` in `is not`. No, ne isto, 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 (`s[12]`), z zavitimi pokličemo funkcijo (`fibonacci(12)`), s piko pa dostopamo do polj oz. metod objekta (`stavek.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 ne bedasto ne 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. [4]

```
>>> 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]
```

[4] Ni pa ga treba preskusiti, saj goljufam: od Pythona 3.0 `range` ne vrača več seznama, temveč generator (več o tem kasneje). Tale pogovor s Pythonom je star dve leti.

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

Funkcija print

Funkcija `print` izpiše vse podane argumente, mednje da presledke, na koncu pa gre v novo vrstico. Če nam kaj ni prav, lahko njeno vedenje spremenimo s poimenovanima argumentoma `sep`, ki določa, kaj bo med argumenti, in `end`, ki določi, kaj bo na koncu.

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

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

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

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

```
>>> s = 'Miran'
>>> s
'Miran'
>>> print(s)
Miran
```

Če rečemo le `s`, izvemo, kaj je `s`, namreč niz. Če rečemo `print(s)`, ga izpišemo. To je pogosto isto (`42` se izpiše kot `42`), a ne vedno (`"Miran"` se izpiše kot `Miran`).

Kontrola toka

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

Pogojni stavki

Stavek `if` je videti takole:

```
if a > 12 or b == 42:
    f()
elif a < 0 or b == 2:
    g()
elif a > 10:
    h()
else:
    j()
```

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:
    f()
else:
    if a < 0 or b == 2:
        g()
    else:
        if a > 10:
(...in tako naprej...)
```

za kar pa bi seveda dobili bolj malo točk za eleganco.

Oklepaji okrog pogojev niso potrebni, razen, seveda, kadar jih potrebujemo zaradi prioritete operatorjev (`or` znotraj `and`...). Če jih kdo želi pisati tudi v primerih, kot je gornji, sme, vendar se bo s tem izdal za prišleka, ki govori z nenavadnim naglasom.

Zanka while

Zdaj, ko poznamo `if`, je tudi zanka `while` je takšna, kot bi jo pričakovali: besedi `while` sledi pogoj (ki ga (tako kot pri `if`) ni treba zapirati v (odvečne) oklepaje, ker izgledajo smešno), 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
```

Zanka for

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[5], Javascriptu...

```
>>> s = ["Miran", "Miha", "Mitja"]
>>> for e in s:
...     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,

[5] PHPja vešči bralec naj bo pozoren na drugačen besedni red v Pythonovi zanki.

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)
```

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)
```

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 še marsičemu drugemu). Če je torej `s` 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(s)):
    print(s[i])
```

drugi pa, seveda,

```
for e in s:
    print(e)
```

Druga koda je preglednejša, malenkost hitrejša, predvsem pa splošnejša, saj deluje tudi v primerih, ko dolžine `s`-ja ni mogoče vnaprej določiti in bi klic funkcije `len` vrnil napako.

Zanka `for` zna tudi kaj razpakirati. Če jo naženemo prek, recimo, seznama parov, lahko elemente teh parov priredimo dvema spremenljivkama.

```
>>> for a, b in [(5, 2), (1, 4), (3, 7)]:
...     print(a+b)
...
7
5
10
```

To je posebej uporabno, kadar imamo zanko, v kateri bi potrebovali tako element seznama kot njegov indeks. Tedaj nam pride prav funkcija `enumerate`, ki za podani seznam vrne seznam parov (indeks, element):

```
>>> enumerate([5, 42, 6, 9, -3, 12])
[(0, 5), (1, 42), (2, 6), (3, 9), (4, -3), (5, 12)]
```

No, tale Pythonov odgovor sem si izmislil, takole, za ilustracijo. V resnici `enumerate` ne naredi nič takega, temveč le nekaj, kar se obnaša tako, kot bi se obnašalo tole. Da bi izvedeli, kaj v resnici naredi `enumerate`, je v tem trenutku še prezgodaj. Za zdaj si pač predstavljajmo, da je tole zgoraj res.

In tako vzemimo, recimo, programerja, ki mora poiskati indeks največjega elementa na seznamu `s`. Storiti mu je tole:

```
naj_i = naj_e = -1
for i, e in enumerate(s):
    if i < 0 or naj_e < e:
        naj_i, naj_e = i, e
```

V zanki bo `e` element seznama, `i` pa njegov indeks. Klasična alternativa je

```
naj_i = naj_e = -1
for i in range(len(s)):
    if i < 0 or naj_e < s[i]:
        naj_i, naj_e = i, s[i]
```

kar je dolgočasno in ničkaj Pythonovsko, ali

```
naj = None
for ie in enumerate(s):
    if naj is None or naj[1] < ie[1]:
        naj = ie
```

Spremenljivki `naj` in `ie` zdaj vsebujeta kar par (indeks, element). Iskani indeks je na koncu v `naj[0]`, element pa v `naj[1]`.

Funkcija `enumerate` ne sprejema le seznamov, temveč vse, prek česar bi bilo mogoče z zanko `for`. Tudi, recimo, nize. Če bi delala, kakor sem si zgoraj izmislil, da dela, bi z nizom naredila tole:

```
>>> enumerate("Miran")
[(0, "M"), (1, "i"), (2, "r"), (3, "a"), (4, "n")]
```

Še ena podobna funkcija kot `enumerate` je `zip`, ki spremeni dva seznama v seznam parov, ali pa tri sezname v seznam trojk in *n* seznamov v seznam *n*-terk, tako da jih "zadrgne" skupaj. Poleg seznamov pa, kot `enumerate`, prebavlja tudi druge stvari, recimo nize. Zadrgniti zna tudi objekte različnih tipov, na primer seznam in niz. Če niso enako dolgi, nič ne de; rezultat je dolg toliko, kolikor je dolg najkrajši od elementov. Čisto tako, kot bi se zapenjale resnične zadrge, če zaradi tovarniške napake (ah, ti Kitajci!) ne bi imele enako dolgih krakov.

```
>>> zip("Miran", [5, 2, 78, 9, 2])
[('M', 5), ('i', 2), ('r', 78), ('a', 9), ('n', 2)]
>>> zip("Miran", "Benjamin", "Ana")
[('M', 'B', 'A'), ('i', 'e', 'n'), ('r', 'n', 'a')]
```

Tale primer je sicer spet izmišljen, vendar malo manj kot `enumerate`, saj je `zip` do različice 3.0 v resnici deloval tako.

Funkciji `zip` in `enumerate` sicer lahko uporabljamo v različnih okoliščinah, vendar ju drugje kot v zanki `for` malone nikoli ne srečamo.

Prekinjanje zank

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 rane pubertete vznemirja 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")
```

Pazimo: `else` je poravnan s `for` in ne z `if`, saj tudi spada k `for` in ne k `if`!

Malo manj uporaben primer – vsaj v primerjavi z gornjim prav gotovo – je spodnja skripta, ki poskuša prebrati spletno stran in obupa po petih neuspešnih poskusih.

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

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 kaj posebnega zgodi, pač rečemo, kot že vemo, `pass`.

Koda torej kliče funkcijo `urllib.request.urlopen`, ki prebere spletno stran. Če klic uspe, se zanka konča z `break` in `else` se ne izvede. Če ji ne uspe (ker stran ne deluje, ker je URL napačen, ker je računalnik priključen na omrežje znanega slovenskega ponudnika interneta...), pa se sproži napaka, izvajanje preskoči v except, ta ne naredi ničesar in zanka se ponovi. Če se to zgodi petkrat, se zanka ne konča z `break`, zato se izvede `else`. V njem bi v pravem programu sicer naredili kaj bolj konkretnega kot `print`, za zdaj pa še ne znamo nič boljšega.

Go to

Ukaza `goto` Python nima. Če ga kdo pogreša, naj se sramuje.

Preprosta definicija funkcije

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.

Funkcij brez rezultata (kot C-jevski `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, recimo, kadar pride do napake, 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, kazalcev in referenc Python ne pozna. 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, par 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 je c med, vključno, `minc` in `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
```

   Mimogrede, ste opazili, kako podobna je Fibonaciju?

2. Nič posebnega. Možne vrednosti c so med `minc` in `maxc`; k slednjemu moramo v klicu `range` prišteti 1. Da bi vsako trojko izpisali le enkrat, spustimo `a` le do `c` deljenem s korenom `2` (razmislite, zakaj); z `int` spremenimo količnik v celo število. Edina nenavadnost programa je ostanek po deljenju z 1. Tako preverimo ali je število celo; ostanek mora biti približno 0, le nekaj zaokrožitvene napake smo pripravljeni tolerirati.

```
import math

def trojka(minc, maxc):
    for c in range(minc, maxc+1):
        for a in range(1, 1+int(c//math.sqrt(2))):
            b = math.sqrt(c**2 - a**2)
            if b % 1 < 1e-6:
                print(a, int(b), c)
```

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

```
def binomski(r, c):
    if c == 0 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 + int(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
```

Sestavljene podatkovne strukture

Zdaj pa je že čas, da si resneje ogledamo sezname in terke, v katere smo doslej le parkrat poškilili, in nize, ki smo jih kar nekako privzeli, poleg tega pa bomo videli še nekaj drugih sestavljenih podatkovnih struktur. Vse te 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.

Seznam

Seznamov se bomo lotili temeljiteje kot ostalih podatkovnih struktur, saj se bodo metode, načini indeksiranja in podobno kasneje ponavljalo tudi pri drugih.

Seznam je objekt tipa `list`. Njegovi elementi so objekti poljubnega tipa.

```
s = [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.

```
s = ["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.

```
s = ["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.[6]

```
>>> s[0]
'Ana'
>>> s[2]
'Cilka'
```

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

```
>>> s[-1]
'Helga'
>>> s[-2]
'Fani'
```

Tole si le zapomnite, da ne boste, kot mnogi drugi prišleki, namesto `s[-3]` pisali `s[len(s)-3]` in se vam bomo hahljali. Z rezanjem (*slicing*) dobimo podseznam.

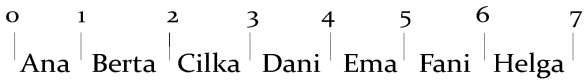
---

[6] Seznam je interno shranjen kot tabela kazalcev na objekte. Tega nam sicer ni potrebno vedeti niti, če razvijamo dodatke v Cju, saj do elementov dostopamo prek funkcij, ne neposredno.

Pravila so takšna kot pri funkciji `range`: interval vključuje začetni element, ne pa tudi končnega.

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

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

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

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

```
>>> s[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 ali razen zadnjih toliko in toliko. Tudi to se zelo elegantno opiše in ujame z različno predznačenimi indeksi in s 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 `s[:2]` prva dva elementa seznama in `s[2:]` vse od drugega elementa naprej (torej vse razen prvih dveh). `s[-2:]` sta zadnja dva elementa (vse od `-2`-gega naprej) in `s[:-2]` vsi razen zadnjih dveh – vse do `-2`-gega, a brez njega, torej manjkata minus drugi in minus prvi.

```
>>> s[:2]
['Ana', 'Berta']
>>> s[2:]
['Cilka', 'Dani', 'Ema', 'Fani', 'Helga']
>>> s[-2:]
['Fani', 'Helga']
>>> s[:-2]
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
>>> s[:]
['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, takšna kopija je plitva, saj oba seznama vsebujeta *iste*, ne le *enake* elemente.

K rezanju lahko dodamo še korak.

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

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

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

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?

```
>>> s[::-1]
```

Ob indeksiranju smo se zadržali precej časa, a ta ni bil vržen stran. Enaka pravila veljajo za indeksiranje 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.

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

```
>>> s[4] = "Erika"
>>> s
['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 (a manj uporabni) vlogi, na seznamu pa jo uporabimo tako:

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

Podoben učinek ima metoda `pop`. Tudi `s.pop(4)` bi pobrisal četrti element, a poleg tega še vrnil njegovo vrednost. Metodo `pop` lahko kličemo (in navadno tudi jo 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`.

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

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

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

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

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

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

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

Prednost `inserta` ni le (očitno) večja preglednost, temveč tudi hitrost, saj `insert` dejansko vrine element,[7] v spodnjem primeru pa sestavimo nov seznam iz treh kosov, namreč iz prvih treh elementov `s`-ja, iz seznama, ki vsebuje element `"Ester"` in iz vseh elementov `s`-ja od tretjega naprej.

K seznamu lahko dodamo tudi seznam.

```
>>> s.extend(["Jasna", "Karla"])
>>> s
['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 izjema. Ker lahko sezname seštevamo, bi bilo očitno konsistento, da bi jih lahko tudi prištevali: če imamo +, smemo pričakovati, da bomo imeli tudi +=. Namesto z `extend` bi lahko gornji seznam dodali z

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

Zadnji, manj očitni in redko uporabljeni način je prirejanje rezinam.

```
>>> s[3:3] = ["Cireja", "Creda"]
>>> s
['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.

---

[7] 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 zelo hitra – napisana je v Cju in od procesorja zahteva le, da premakne podatke v kosu pomnilnika – torej se v primerjavi s kodo, napisano v Pythonu, zgodi *praktično* "v trenutku", se pravi v času O(1).

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

Mimogrede, rezino lahko zamenjamo tudi s praznim seznamom,

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

To je ekvivalentno brisanju,

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

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

```
>>> 'Berta' in s
True
>>> 'Dani' in s
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 vrednost v seznamu najdemo le tako, da po vrsti pregledujemo njegove elemente, bomo takrat, ko nas čaka veliko takšnega iskanja, namesto seznama raje uporabili množico ali slovar, ki sta prav tako vdelana v Python.

Dolžine seznama ne dobimo s kako metodo (`len`, `length`, `size` ali kaj podobnega), temveč s funkcijo `len`. Razlogi za to so tehnični.[8]

---

[8] 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.

```
>>> len(s)
8
```

Metoda `reverse` obrne seznam. Obrniti seznam smo se sicer mimogrede naučili ob rezinah, vendar je razlika v tem, da `s[::-1]` sestavi nov seznam, `s.reverse()` pa ga obrne *na mestu* (torej `s.reverse()` obrne, spremeni sam seznam `s`). Metoda `sort` seznam uredi, prav tako na mestu. Kot argument ji lahko podamo funkcijo za računanje ključev, po katerih bo seznam urejen; podati jo moramo s ključno besedo `key`.

```
>>> s = ["Ana", "Berta", "Cilka", "Danica", "Eva", "Frančiška"]
>>> s.sort(key=len)
>>> s
['Ana', 'Eva', 'Berta', 'Cilka', 'Danica', 'Frančiška']
>>> s.sort()
>>> s
['Ana', 'Berta', 'Cilka', 'Danica', 'Eva', 'Frančiška']
```

Če funkcijo izpustimo, `sort` elemente primerja, 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
```

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.

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

Na prvi pogled smo sestavili seznam, ki vsebuje pet praznih seznamov. V resnici pa imamo seznam, ki petkrat vsebuje *isti* (prazni) seznam. Ko v prvo kopijo dodamo element `1`, se ta pojavi tudi v drugih kopijah. A več o tem bo povedalo posebno poglavje, eno najzanimivejših v knjigi.

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

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

Zaresen programer v Pythonu pa naredi, kakor se bomo naučili v nekem drugem zanimivem poglavju, namreč

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

Naloge

1. Delaj se, da Python ne pozna primerjanja seznamov in napiši funkcijo `primerjaj(s, t)`, ki vrne `-1`, če je seznam `s` (leksikografsko) manjši od `t`, `0`, če sta enaka in `1`, če je `s` večji.

2. Poišči najmanjši element v seznamu.
   a. Vrni najmanjši element.
   b. Vrni indeks, kjer se pojavi najmanjši element, in njegovo vrednost.
   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 vsebuje le male črke in nobenih drugih znakov.

Rešitve

1. Python je imel do različice 3.0 funkcijo cmp, ki je prejela dva argumenta, vrnila -1, če je prvi manjši, 0, če sta bila enaka in 1, če je prvi večji. Prišla nam bo prav za primerjanje elementov seznamov, zato si jo pripravimo.

```
def cmp(a, b):
    if a < b:
        return -1
    elif a > b:
        return 1
    else:
        return 0
```

   Ah, veste kaj – smo hackerji ali nismo? In, poleg tega, je bool izpeljan iz int ali ni? Naredimo kot se zagre.

```
def cmp(a, b):
    return (a > b) - (b > a)
```

   Razmislite in spoznali boste, da je tudi to prav.

Zdaj pa h glavni nalogi, primerjanju seznamov. Manj pozorni bralec naredi tako:

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

Zanka teče do dolžine krajšega od seznamov. Če naletimo na različna elementa, vrnemo ustrezno vrednost. Če se zanka izteče (ker je enega ali obeh seznamov zmanjkalo), je "manjši" tisti seznam, ki je krajši, za kar spet uporabimo `cmp`.

Pozorni bralec pa se spomni funkcije zip in naredi tako:

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

2. Rešitev prve različice naloge je čista klasika.

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

Drugo smo že rešili, ko smo spoznali `enumerate`. Rešimo jo najprej brez.

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

Zdaj pa še enkrat z `enumerate`, tako kot v prejšnjem poglavju, le v funkcijo jo zapakirajmo.

```
def najmanjsi_b(s):
    naj = None
    for ie in enumerate(s):
        if naj is None or naj[1] > ie[1]:
            naj = ie
    return naj
```

Za rešitev naloge pod točko c se naslonimo na prvo različico rešitve naloge b (lahko pa bi se tudi na drugo).

```
def najmanjsi_c(s):
    najm, naji = s[0], [0]
    for i in range(len(s)):
        if s[i] < najm:
            najm, naji = s[i], [i]
        elif s[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(s):
    zacetek = 0
    podseznami = []
    for i in range(1, len(s)):
        if s[i] < s[i-1]:
            podseznami.append(s[zacetek:i])
            zacetek = i
    podseznami.append(s[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 enaka.

7. Funkcijo? Je to vredno funkcije?

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

Funkcijo lahko pohvalimo, da 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)))
```

Terka

Terka (*tuple*[9]) je zelo podobna seznamu, le precej bolj omejena je. Ne moremo je spreminjati (terke 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")
```

---

[9] 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.

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
```

in `return`, ki vrača terko. To je namreč tako 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 `s.append("Lenka", "Matjaz")`, bi se Python pritožil, da metoda `append` sprejme le en argument, ne dveh. Pravilno je `s.append(("Lenka", "Matjaz"))`.

Terko lahko sestavimo tudi s klicem konstruktorja `tuple`, ki mu podamo 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 metod(e `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 tam, kjer je potrebno zagotoviti, da se njena vsebina ne bo

spreminjala. 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.

Sestavljeni so iz znakov. Python tipa, kot je Cjevski `char`, nima; "znaki", iz katerih je sestavljen niz, so v resnici le nizi dolžine 1. Za Slovence (in druge neangleško govoreče) je ena najpomembnejših prednosti različice 3.0 pred starejšimi Pythoni ta, da lahko nizi vsebujejo poljubne znake po standardu Unicode. Kako so nizi shranjeni interno, nas ne zanima (navadno, a ne nujno, so zapisani z UTF-16). V komunikaciji z operacijskim sistemom uporablja Python privzete nastavitve operacijskega sistema. Po potrebi lahko, recimo, pri branju kake datoteke določimo, naj se uporablja kako drugačno kodiranje, večinoma pa se nam s tem ni potrebno obremenjevati in reči preprosto delujejo.

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

Če nočemo niza v več vrsticah, temveč ga razbijamo na več vrstic le, da program ne bi bil preširok, pa bomo pisali bodisi

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

bodisi

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

Včasih ne želimo pisati dvojnih vzvratnih poševnic in si želimo, da le-te v nizu ne bi imele posebnega pomena. Takšne želje so posebej pogoste pri pisanju regularnih izrazov, kjer hitro pridemo do četvernih 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 = "miran"
>>> s[3]
'r'
>>> s[:3]
'mir'
>>> s[-4:]
'iran'
```

Prek njih lahko gremo 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 (`"x"*10` naredi niz desetih iksov), mu z operatorjem += dodamo drug niz...

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

```
>>> 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 posamezne znake že sestavljenega niza pravzaprav zelo redko spreminjamo.

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

```
>>> s = k = [1, 2]
>>> k += [3]
>>> s
[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 `s`, 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.

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

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

```
>>> "".join(s)
'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 po Cjevsko. Čeprav je oblikovanje nizov z oznakami, kot na primer `%5.3f`, je v Pythonu že nekaj časa iz mode, ga zavoljo združljivosti s starejšimi različicami tudi v različici 3.0 (in nadaljnjih) niso opustili. To je dobro, saj je združljivo tudi s predznanjem teh, ki se programiranja ne učite od začetka. Ko bomo končali ogled starega načina, pa le ne preskočite tudi novejšega.

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 znotraj niza so takšne kot v Cju, recimo `%5.3f` ali `%-02i`. Razlika med Cjem in Pythonom je le 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? %(kol).3f" % vars())
Koliko je 11/5? 2.200
```

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

Oblikovanje z metodo `format`. Takole pa gre po novem: niz ima metodo `format`, ki poišče vse pojavitve vejičnih oklepajev, `{}`, in jih zamenja s podanimi argumenti.

```
>>> "Koliko je {}/{}? {}".format(a, b, kol)
'Koliko je 11/5? 2.2'
```

To seveda še ni vse: oklepaji niso nujno prazni. Vanje lahko, za začetek, dodamo številke argumentov.

```
>>> "Koliko je {0}/{1} ({0} deljeno z {1})? {2}".format(a, b, kol)
'Koliko je 11/5 (11 deljeno z 5)? 2.2'
```

Če damo metodi poimenovane argumente, se lahko pri oblikovanju sklicujemo nanje.

```
>>> "Koliko je {a}/{b}? {kol}".format(a=11, b=5, kol=2.2)
'Koliko je 11/5? 2.2'
```

Če hočemo uporabiti enaka imena, kot jih imamo v programu, pišemo

```
>>> "Koliko je {a}/{b}? {kol}".format(a=a, b=b, kol=kol)
'Koliko je 11/5? 2.2'
```

To je seveda čudno, zato uporabimo prejle videni `vars()`, na še nekoliko čudnejši način,

```
>>> "Koliko je {a}/{b}? {kol}".format(**vars())
'Koliko je 11/5? 2.2'
```

ali pa pokličemo sorodno metodo `format_map`, ki pričakuje natančno to, kar dobimo od `vars()`,

```
>>> "Koliko je {a}/{b}? {kol}".format_map(vars())
'Koliko je 11/5? 2.2'
```

Če je mogoče argumente indeksirati, lahko dodamo indekse.

```
>>> "{t[0]} in {t[1]} je {r}".format(t=(4, 5), r=9)
'4 in 5 je 9'
```

Prav tako lahko dostopamo do polj objektov (o tem se še nismo pogovarjali, a bralec bo brez težav razumel spodnje).

```
>>> "Realni del: {r.real}, imaginarni: {r.imag}".format(r=5+4j)
'Realni del: 5.0, imaginarni: 4.0'
```

Temu, kar smo doslej pisali v oklepaje, lahko sledi dvopičje in za njim navodila za obliko. Ta so spet podobna Cjevskim. Pišemo lahko, recimo, takole

```
>>> "Koliko je {a:03}/{b:03}? {kol:5.3f}".format(**vars())
'Koliko je 011/005? 2.200'
```

Detajle oblikovanja si bo bralec ogledal v dokumentaciji; tule nam zadošča, da smo pokazali osnovno idejo.

Bajti

Nizom dela družbo zelo podoben podatkovni tip, ki se imenuje `bytes`. V programu ga opišemo podobno kot nize, le črko `b` dodamo pred narekovaj. Enako stori Python ob izpisu.

```
>>> s = b"Tole ni niz!"
>>> s
b'Tole ni niz!'
```

Bajte si lahko predstavljamo kot polje bajtov (ki jih Python slučajno izpiše kot niz). Najbolj opazna je razlika ob indeksiranju, kjer namesto znakov dobimo števila.

```
>>> s[4]
32
```

Četrti znak v gornjem "nizu" je namreč presledek in njegova koda ASCII je 32.

Po drugi strani pa lahko v bajtih vidimo besedilo, ki ga še nismo "interpretirali" v pravi niz. Bajte bomo pogosto prebrali iz datoteke (kako, bomo še povedali). Pri branju v bajte za določeno kodo (številko), še ne vemo, kateri znak predstavlja.

Če v datoteki naletimo na znak s kodo 232, gre morda za č (če je datoteka zapisana po standardu cp1250), morda za è (standard cp1252), morda kaj tretjega. Tip `bytes` se pri indeksiranju dilemi izogne tako, da ne vrne znaka, temveč številko 232, pri izpisu pa tako, da namesto črke izpiše kodo, `\xe8` (`e8` je 232 po šestnajstiško, v stari Cjevski maniri pa je številka okrašena še z `\x`).

Kadar beremo datoteke, ki so *v resnici* binarne, tako ali tako potrebujemo številke, ne znakov. Kadar imamo v resnici opravka z besedilom in bi radi `bytes` spremenili v niz, to storimo tako, da ga *dekodiramo*.

```
>>> s.decode("ascii")
'Tole ni niz!'
```

Črka `b` pred narekovajem je izginila: metoda `decode` je spremenila bajte v običajni niz, pri čemer je številke interpretirala kot čiste kode ASCII.

V drugo smer, od nizov v bajte, vodi metoda `encode`, ki ji moramo podati kodek, standard, po katerem bi radi zapisali znake.

```
>>> t = "Tale reč vsebuje šumnike."
>>> t.encode("utf-8")
b'Tale re\xc4\x8d vsebuje \xc5\xa1umnike.'
>>> t.encode("utf-16")
b'\xff\xfeT\x00a\x00l\x00e\x00 \x00r\x00e\x00\r\x01
\x00v\x00s\x00e\x00b\x00u\x00j\x00e\x00
\x00a\x01u\x00m\x00n\x00i\x00k\x00e\x00.\x00'
>>> t.encode("cp1250")
b'Tale re\xe8 vsebuje \x9aumnike.'
>>> t.encode("iso-8859-2")
b'Tale re\xe8 vsebuje \xb9umnike.'
```

Kakšno zaporedje bajtov dobimo, je seveda odvisno od uporabljenega kodeka. Zgodi se tudi, da kakega znaka s podanim načinom kodiranja ni mogoče zapisati.

```
>>> t.encode("cp1252")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File
"/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.
2/encodings/cp1252.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_table)
UnicodeEncodeError: 'charmap' codec can't encode character
'\u010d' in position 7: character maps to <undefined>
```

Če nočemo zamenjati kodeka (čeprav bi ga bilo počasi dobro; čas je že za njegovo upokojitev, saj povzroča le stalne nepotrebne sitnosti[10]), lahko naročimo, naj takšne znake izpusti, zamenja z vprašajem ali zamenja s kodo, kakršno bi prepoznali brskalniki.

```
>>> t.encode("cp1252", "ignore")
b'Tale re vsebuje \x9aumnike.'
>>> t.encode("cp1252", "replace")
b'Tale re? vsebuje \x9aumnike.'
>>> t.encode("cp1252", "xmlcharrefreplace")
b'Tale re&#269; vsebuje \x9aumnike.'
```

Zakaj je tako koristno ločiti med nizi in bajti, bomo videli še malo kasneje, ko se bomo učili brati datoteke.

Naloge

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

2. Napiši funkciji za približno primerjanje nizov:

[10] Za nas, zahodnjake, ni nobenega razloga, da ne bi pustili starih kodekov in prešli na UTF-8, saj je združljiv z ASCII; Kitajci pa bodo že morali potrpeti z nami in požreti kak kilobajt več.

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.

3. 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 (eče je naš ded segrešil, je tudi umrl), bomo k letnici prilepili `2`, če je prva števka letnice enaka `0`, sicer pa `1`. Prevod tega v Python je dobeseden.

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

2. 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
```

Kot vedno pa se nam tudi pri tej nalogi splača vedeti za funkcijo `zip`.

```
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 si, ti in zip(s, t):
        ujemajocih += si == ti
    return ujemajocih >= 0.9 * len(s)
```

Rezultat izraza `si == ti` 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 nizu dobili deljenje z `0`. Tako, kot smo ga preobrnili v funkciji, pa deluje tudi s praznimi nizi.

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)
```

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

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

   Za prazne sezname bi bilo lepše javiti napako. 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 uporabni relaciji urejenosti. Zagotavlja nam hiter dostop do posameznih elementov, hitro dodajanje in brisanje; cena, ki jo moramo plačati, je večja poraba pomnilnika.[11]

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 – a običajno niso – različnih tipov. 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. Tako zdaj vemo, zakaj je priročno, da terk in nizov ne moremo spreminjati: ko bi ne bilo tako, jih ne bi mogli uporabiti

[11] Za firbce: slovar je implementiran kot razpršena tabela (*hash table*).

kot ključe v slovarju – kar bi bila škoda.[12]

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["Matija"] = 55
>>> d
{'Matija': 55, 'Miha': 12, 'Matevz': None}
```

Prirejanje elementa s ključem, ki še ne obstaja, pomeni dodajanje; prirejanje z obstoječim ključem pomeni 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 vse ključe, vse vrednosti in vse pare (ključ, vrednost), tako da jih lahko pregledamo, recimo, z zanko `for`.

```
>>> for k, v in d.items():
...     print("{}\t{}".format(k, v))
...
Matija 5
Miha   12
Matevz None
```

Primer je poučen zaradi načina razpakiranja terke: podobno kot operator `=` zna terko razpakirati tudi `for`.

Če zanko `for` spustimo prek slovarja kar tako, vrača ključe – tako, kot če bi jo nagnali prek `keys`.

---

[12] Že slišim ugovore: kaj pa števila? Kako lahko števila nastopajo kot ključi, čeprav jih lahko spremenimo, a? Ne bo držalo. Tudi števila so nespremenljiva (*immutable*). `2` je `2` in ne more nikoli postati `3`. Če številu nekaj prištejemo, dobimo drugo število.

```
>>> for t in d:
...     print(t)
...
Matija
Miha
Matevz
```

Slovarjev ne moremo kar tako seštevati, pač pa metoda `update` k slovarju doda vsebino drugega slovarja (in, če se kak ključ ponovi, 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.setdefault("Matija", 48)
55
>>> d
{'Matija': 55, 'Miha': 12, 'Matevz': None}
>>> d.setdefault("Marjan", 33)
33
>>> d
{'Matija': 55, 'Miha': 12, 'Marjan': 33, 'Matevz': None}
```

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("Marjan", "ga ni")
33
>>> d.get("Mitja", "ga ni")
'ga ni'
```

Jako uporaben je slovar tipa `defaultdict`, ki se nahaja v modulu `collections`. Izpeljan je iz `dict` in se razlikuje po tem, da mu lahko podamo funkcijo, ki jo bo uporabil za konstruiranje privzetih vrednosti. Ker je v Pythonu mogoče klicati tipe (kako je s tem, bomo podrobneje preučili kasneje), je primerna funkcija za sestavljanje privzetih vrednosti kar ime tipa.

```
>>> import collections
>>> d = collections.defaultdict(list)
>>> d[42]
[]
>>> d
defaultdict(<class 'list'>, {42: []})
```

Ko smo zahtevali element s ključem 42, se Python ni razburjal, da takšnega ključa ni, temveč je poklical funkcijo `list`, ki je sestavila prazen seznam. Element s ključem 42 se zdaj nahaja v slovarju. V slovarju so še vedno lahko tudi vrednosti, ki niso seznami.

```
>>> d["Berta"] = 33
>>> d
defaultdict(<class 'list'>, {42: [], 'Berta': 33})
```

Najbolj tipična uporaba je tale: v seznam, ki pripada določenemu ključu, dodamo nov element. Če ključa še ni, se seznam pojavi "sam od sebe".

```
>>> d = collections.defaultdict(list)
>>> d["Cilka"].append(12)
>>> d["Ana"].append(33)
>>> d["Cilka"].append(11)
>>> d
defaultdict(<class 'list'>, {'Cilka': [12, 11], 'Ana': [33]})
```

Naloga

1. Napiši funkcijo `najpogostejsa(s)`, ki vrne najpogostejšo črko v podanem nizu `s`.

Rešitev

1. Začnimo s sestavljanjem slovarja, katerega ključi bodo črke in vrednosti število pojavitev.

```
def najpogostejsa(s):
    frekv = {}
    for c in s:
        if c in frekv:
            frekv[c] += 1
        else:
            frekv[c] = 1
```

Lahko bi jo naredili tudi takole.

```
def najpogostejsa(s):
    frekv = {}
    for c in s:
        frekv[c] = frekv.get(c, 0) + 1
```

Ker vemo za slovarje s privzetimi vrednostmi, smo lahko še krajši.

```
import collections
def najpogostejsa(s):
    frekv  = collections.defaultdict(int)
    for c in s:
        frekv[c] += 1
```

Temu sledi že znana vaja: iskanje največjega elementa.

```
    najponov = 0
    najcrka = None
    for crka, ponov in frekv.items():
        if ponov > najponov:
            najcrka, najponov = crka, ponov
    return najcrka
```

Za konec še spoiler: modul `collections` vsebuje tudi razred `Counter`, ki šteje elemente tistega, kar mu podamo (niz, seznam...). Njegovo širše poslanstvo je pokvariti vse izpitne naloge tipa "izračunaj frekvence tega-in-tega".

```
>>> collections.Counter(s)
Counter({'e': 7, ' ': 4, 'v': 4, 'a': 2, 'k': 2, 'j': 2, 'l': 2,
's': 2, 'b': 1, 'i': 1, 'o': 1, '.': 1, 'u': 1, 'T': 1, 't': 1})
```

Množica

Zadnja vdelana podatkovna struktura, ki si jo bomo ogledali, je množica. Podobno kot slovarje jo zapišemo z zavitimi oklepaji. Da ne gre za slovar, Python odkrije po pomanjkanju dvopičij.

```
>>> s = {1, 2, 3}
```

Prazne množice ne moremo predstaviti s parom oklepajev, saj bi to predstavljalo prazen slovar. Prazno množico dobimo, če pokličemo funkcijo (pravzaprav konstruktor) `set`.[13] Prav tako lahko konstruktorju `set` podamo, recimo, seznam ali poljubno drugo reč, prek katere lahko naženemo zanko `for`.

```
>>> set([1, 2, 3])
{1, 2, 3}
>>> set("Miran")
{'i', 'a', 'r', 'M', 'n'}
>>> set(range(5))
{0, 1, 2, 3, 4}
```

Množica je v sorodu s slovarjem (oba sta predstavljena z razpršeno tabelo), zato sme vsebovati le elemente, ki se ne morejo spremeniti.

Množici lahko z `add` in `remove` dodajamo in odjemljemo elemente. Variacija na `remove` je `discard`, ki element, če obstaja, pobriše, če ga ni, 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, spet preverimo z operatorjem `in`.

Seveda 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. Vse te operacije je mogoče zapisati tudi z operatorji, ki spominjajo na operacije z biti.

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> s | t
{1, 2, 3, 4}
>>> s & t
{2, 3}
>>> s ^ t
{1, 4}
```

---

[13] Razloga, da so prazni oklepaji slovar in ne množica, kot bi nemara pričakoval matematik, sta dva. Slovar je bil prej, množica je kasnejši izum. Poleg tega slovarje potrebujemo veliko pogosteje kot množice; pa naj imajo še preprostejši zapis.

```
>>> s - t
{1}
>>> s < t
False
>>> {2, 3} < t
True
>>> {2, 3, 4} < t
False
>>> {2, 3, 4} <= t
True
```

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, da pokličemo `frozenset`. Argumenti, ki jih prebavi, so enaki kot pri `set`. 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, recimo kar `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
```

Datoteka

Kot smo vajeni iz drugih jezikov, datoteko odpremo s funkcijo `open`, ki ji kot argument podamo ime datoteke in način, na katerega naj jo odpre. 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. Za nekaj dodatnih možnosti bo bralec izvedel v Pythonovi dokumentaciji.

```
>>> f = open("slika.gif", "rb")
```

Kot iz drugih jezikov večinoma nismo vajeni, razlika med binarnim in besedilnim načinom ni le v tem, da prvi pusti znake za nove vrstice pri miru (ker ve, da ne gre za nove vrstice), drugi pa ne. Razlika je tudi in predvsem v tem, kaj dobimo, ko datoteko beremo: binarne datoteke vračajo bajte (`bytes`), besedilne nize (`str`).

Oboje je bolj prikladno, kot se zdi na prvi pogled. Preberimo 15 bajtov zgoraj odprte datoteke

```
>>> data = f.read(15)
>>> data
b'GIF89a\xd8\x04\x84\x03\xf7\x00\x00\x06\x03'
```

Črka `b` pred narekovajem, vemo, izda, da gre za bajte in ne niz. Znaki s kodami med 32 in 127 so izpisani kot znaki (`GIF89a`), tisti izven tega območja pa s kodami, kot sta `\x04` in `\x84`. In ker gre za bajte, z indeksiranjem ne dobimo znakov (nizov dolžine 1), temveč števila. To je dobro, ker lahko do širine slike, ki jo ima GIF zapisano v sedmem in osmem bajtu po pravilu tankega konca, pridemo z

```
>>> b[6] + b[7]*0x100
1240
```

Pri branju besedilnih datotek dobivamo, kot bi človek pričakoval, nize. Nizi v Pythonu so, spomnimo se, shranjeni v enem od zapisov Unicodea. V njih torej niso številke, ki jih lahko pretvarjamo v znake po eni ali kaki drugi kodni tabeli: ne, v nizu so "znaki". To pomeni, da se pri branju datoteke vsebina datoteke že interpretira po določeni kodni tabeli. Če pri odpiranju ne povemo posebej, katera tabela bi to bila, Python uporabi privzeto kodiranje sistema. Če dodamo argument `encode`, pa lahko način kodiranja podamo eksplicitno.

```
>>> f = open("besedilo.txt", "r", encode="cp1250")
```

Podobna pravila veljajo tudi za pisanje: v besedilno datoteko pišemo nize in pri pisanju se niz zapiše z načinom kodiranja, ki smo ga podali ob odpiranju datoteke (ali z načinom, ki je privzet za sistem). V binarno datoteko pa lahko pišemo le bajte; če želimo vanjo zapisati niz, ga moramo prej sami pretvoriti v bajte z metodo `encode`, pri čemer, seveda, prav tako predpišemo način kodiranja.

V drugih pogledih je delo z datotekami podobno, kot smo ga vajeni iz drugih jezikov. Datoteko lahko zapremo ročno, tako da 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. 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 primer bomo videli vsak čas.

Metoda `read` prebere podano število znakov ali, če jo pokličemo brez argumentov, celotno datoteko. O tem, kakšen rezultat vrača, smo se že pogovarjali.

Besedilne datoteke pogosto beremo po vrsticah. Metoda `readline` prebere eno samo vrstico, `readlines` pa vse in jih vrne kot seznam nizov oziroma bajtov. Še bolj praktično 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.[14]

```
>>> for s in open("preseren.txt", "r"):
...     s
...
'Valjhun, sin Kajtimara, boj krvavi\n'
'že dolgo bije za kršansko vero,\n'
'z Avreljam Droh se več mu v bran ne stavi,\n'
'končano njuno je in marsik'tero\n'
'življenje, kri po Kranji, Koratani\n'
'prelita, napolnila bi jezero.\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.

Na enak način, po vrsticah, lahko beremo tudi binarno datoteko, če je to smiselno. Navadno ni.

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

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

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

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

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

---

[14] Tule se bralcu splača preveriti razliko med izpisovanjem s `print(s)` in izpisovanjem tako, da vtipkamo samo `s`. Odkod izvira, bo zares razumel, ko bo prebral poglavje o razredih in posebnih metodah `__str__` in `__repr__`.

Tudi druge metode datotek so takšne, kot smo jih navajeni od drugod – `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.

Mimogrede omenimo še, da modul `sys` vsebuje spremenljivke `stdin`, `stdout` in `stderr`, ki predstavljajo standardni vhod, izhod in izhod za napake. Vse tri lahko uporabljamo kot datoteke, lahko pa jih tudi preusmerjamo preprosto tako, da jim prirejamo druge vrednosti.

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

Naloge

1. 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, poišči na spletu; predpostaviti smeš, da vsebuje le znake ASCII.)

Rešitve

1. Naloga zahteva nekaj malega telovadbe z datotekami – branje zadnjih 128 bajtov, kjer se skriva opis, ki se začne s črkami "TAG", ki jim sledijo zahtevani podatki. Ker moramo datoteko odpreti kot binarno, bodo tisto, kar bomo prebrali, bajti, zato jim moramo dekodirati. 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 = open(fname, "rb")
            f.seek(-128, 2)
            s = f.read().decode("ascii").replace("\x00", " ")
            if s[:3] == "TAG":
                print("Naslov: ", s[5:33])
                print("Izvajalec:", s[33:63])
                print()
```

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`.

```
>>> s = ["Ana", "Berta", "Cilka"]
>>> t = ("Ana", "Berta", "Cilka")
>>> d = {"Ana": 1, "Berta": 2, "Cilka": 3}
>>> s = set(s)
>>> f = open("samSebe.py")
>>>
>>> ", ".join(s)
'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, end="")\n'
```

Vrstni red izpisanega ni vedno enak; ker nekatere podatkovne strukture niso urejene, je vrstni red pri njih navidez 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):
    s = ""
    for element in zaporedje:
        if s:
            s += locilo
        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 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 v tem, da se vzorcev (*template*) v C++ začetniki bojijo, ker jih ne razumejo, profesionalci pa, ker jih skrbi, kako se bodo nanje odzvali različni prevajalniki. V Pythonu pa na ta način programiramo, ne da bi se tega zavedali.

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.

Argumenti funkcij

Le nečesa v tem poglavju ne sprašujte, le enega odgovora ne pričakujte: se argumenti prenašajo po vrednosti ali referenci? Vse drugo takoj pojasnim, odgovora na to vprašanje pa ne boste našli ne v tej knjigi ne kje na spletu. Čemu ne? Izvedeli bomo kasneje, v posebnem poglavju: vprašanje o vrednostih in referencah v Pythonu sploh smisla.

Privzete vrednosti. Videli smo že, kako določimo privzete vrednosti: za ime argumenta postavimo enačaj in vrednost. Tako smo najbrž vajeni tudi iz drugih jezikov. Argumentom, ki imajo privzete vrednosti, ne smejo slediti takšni brez, kot v tejle napačno definirani funkciji.

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

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 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.)

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

Imejmo funkcijo s takšnimi argumenti:

```
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:

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

Pomen prvih treh argumentov je določen s položajem – to so pač prvi trije argumenti, `widget`, `master` in `label`. Na kaj se nanaša zadnji, pove 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. Je mogoče napisati funkcijo, ki prejme poljubno število argumentov? Seveda. Na konec definicije damo argument, pred čigar ime postavimo zvezdico. Ta bo dobil terko z vsemi neporabljenimi mestnimi argumenti.

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

To 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. Da to želimo, povemo tako, pred ime argumenta, ki naj vsebuje takšen slovar, postavimo dve zvezdici. 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, poleg njiju pa še mestne in poimenske argumente. Spodnjo funkcijo lahko kličemo, kakor hočemo in v nobenem primeru ne bo naredila nič.

```
>>> def poziralka_argumentov(*args, **kwargs):
...    pass
...
>>> poziralka_argumentov(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. Takšnim podtaknemo takšnega omnivora, pa naj ga kličejo, kakor želijo in s čimerkoli 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.

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, **prva_clena())`, kjer je `prva_clena` neka funkcija, ki vrača slovar, ki vsebuje

elementa s ključema `a` in `b` (ali samo enim ali celo nobenim od njiju, saj imata argumenta funkcije določene privzete vrednosti, nikakor pa ne sme vsebovati argumentov z drugačnimi imeni, saj Fibonacci tega ne mara).

Zdaj je pojasnjeno, zakaj smo pri oblikovanju nizov pisali `format(**vars())`: funkciji smo kot poimenske argumente poslali kar vse spremenljivke. [15]

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

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 ga ničemur ne priredimo in nikamor ne izpišemo. Pač pa take nize opazi tolmač in ga uporabi kot dokumentacijo. Točneje, ker je funkcija objekt, ima lahko polja; dokumentacija se shrani v polje `__doc__`. Tam ga najdejo in uporabijo razvojna okolja.

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

Do dokumentacije lahko pridemo tudi "ročno": izpiše jo funkcija `help`, ki ji kot argument podamo funkcijo, o kateri bi se radi poučili.

```
>>> help(len)
Help on built-in function len in module __builtin__:
```

[15] Iz zakulisja: Python argumente funkcije interno vedno prenaša v terki (za mestne) in slovarju (za poimenske) argumente. Ko napišemo `f(1, 4, 6, a=12, b=44)`, Python naredi nekaj podobnega, kot če bi mu napisali `f(*(1, 4, 5), **{a=12, b=44})`. Pythonove funkcije si v resnici podajajo terke in slovarje argumentov.

```
len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.

>>> help(list.append)
Help on method_descriptor:

append(...)
    L.append(object) -- append object to end
```

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

Anonimne funkcije

Včasih moramo na hitro, mimogrede, definirati kratko funkcijo, ki bo le nekaj malega izračunala in si ne zasluži niti imena. Denimo, da imamo seznam parov imen ljudi in njihovih tež

```
s = [("Ana", 63), ("Berta", 83), ("Cilka", 59)]
```

Radi bi ga uredili po težah, kar bomo storili s seznamovo metodo `sort`, ki ji bomo za izračun ključa podali funkcijo, ki vrne težo podanega subjekta.

```
def vrni_tezo(x):
    return x[1]

s.sort(key=vrni_tezo)
```

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

Anonimno 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.

```
s.sort(key=lambda x: x[1])
```

Anonimna funkcija sme vsebovati le izraz. Vanjo ne moremo stlačiti 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. Dopušča pa, recimo, klicanje drugih funkcij, vendar le tako, da jih nekako vključimo v izraz.

Lokalne in globalne spremenljivke

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.[16] 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. To bi se namreč dogajalo, če bi bile spremenljivke globalne.

---

[16] 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.

Če hočemo, da bo spremenljivka globalna, moramo to posebej povedati.

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

Zdaj pa se, glede na to, da bralstvo te knjige ni brez programerskih izkušenj, dogovorimo še nekaj: globalne spremenljivke, sploh pa funkcije, ki spreminjajo globalne spremenljivke (branje globalnih spremenljivk še nekako pretrpimo), so grdobija, ki je skoraj vedno nepotrebna. Preveril sem 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.[17]

Kaj je definicija funkcije?

Kako, kaj je zdaj naenkrat to – kaj je definicija funkcije? Saj menda znamo programirati, definicija funkcije je … no, pač definicija funkcije. Ali jih nismo definirali že na ducate, zadnjo kar na prejšnji strani?

Prav, umikam se, vprašanje ni najbolj posrečeno postavljeno, a si vseeno zasluži odgovor: *definicija funkcije je koda, kos programa, ki definira funkcijo*. V prevajanih jezikih se definicija funkcije prevede; prevajalnik gre enkrat za vedno čeznjo in ve zanjo. V Pythonu, ki je tolmačen jezik, *se definicija funkcije izvede*. Ko Python izvaja program in naleti na definicijo funkcije, pač definira funkcijo. In to vsakič, ko naleti na definicijo.

Poglejmo torej definicijo kake funkcije.

---

[17] 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.

```
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. Funkcija je objekt, ki predstavlja kodo, napisano v teh štirih vrsticah. Prva vrstica pa je neke vrste prirejanje, ki kodi v zadnjih štirih vrsticah priredi ime `fibonacci`, mimogrede pa pove še, koliko argumentov sprejema funkcija in kako jim bomo rekli. Ko se definicija funkcija izvede, imamo funkcijo z imenom `fibonacci`.

```
def f():
    return 42

print(f())

def f():
    return 33

print(f())

def f(x):
    return x**2

print(f(5))
```

Tale program izpiše 42, 33 in 25. Ko se izvaja, namreč trikrat zapored definira funkcijo in vsaka definicija povozi prejšnjo. Za še en pošteno pretiran primer zaprimo definicijo funkcije kar v zanko.

```
>>> for i in range(4):
...     def f(x):
...         return x**i
...     print(f(5))
...
1
5
25
125
```

Tole štirikrat (re)definira f, pri čemer prvič vrača `x**0`, drugič `x**1`, tretjič `x**2` in četrtič `x**3`. Vsakič jo tudi pokliče in tako izpisuje potence 5. Ko je zanke konec, je funkcija `f` še vedno definirana – zadnja funkcija `f` namreč, tista, ki računa kube.

Da se z definicijami funkcij ne šalimo, dokažimo s seznamom funkcij.

```
>>> potence = []
>>> for i in range(4):
...     def f(x, p=i):
...         return x**p
...     potence.append(f)
...
>>> potence
[<function f at 0x101b82380>, <function f at 0x10167be20>,
<function f at 0x10167bd98>, <function f at 0x101b82408>]
>>> potence[2](5)
25
>>> potence[3](5)
125
```

Dobili smo seznam s štirimi funkcijami: prva vrne `x**0`, druga `x**1` in tako naprej. Vsem je sicer ime `f`, vendar gre za štiri različne funkcije `f`; da so različne, vidimo po njihovih naslovih (s katerimi se v Pythonu sicer ne ukvarjamo).[18] V tem primeru smo vrednost, ki jo ima `i`, skrili v argument s privzeto vrednostjo. Poskusite brez, funkcija naj nima argumenta `p` in naj vrača `x**i`. Ne deluje, kriva pa je zanka `for`; zakaj, presega okvire te knjige, če boste kdaj delali kaj podobnega (da, v resnici kdaj delamo takšne stvari!), pa si le zapomnite, da je potrebno spraviti vrednost zančne spremenljivke v drugo spremenljivko ali privzeto vrednost argumenta, če jo želimo obdržati.

Če že ravno vemo za lambda-funkcije, naredimo isto reč še z njimi.

```
>>> potence = []
>>> for i in range(4):
...     potence.append(lambda x, p=i: x**p)
```

[18] Kako imajo lahko različne funkcije isto ime? V resnici gre le za ime, s katerim so bile definirane; to ime ne pomeni ničesar. Ko je zanke konec, se ime `f` nanaša na eno samo funkcijo, namreč zadnjo. Do ostalih lahko pridemo samo prek seznama `potence`.

To je tako elegantno, da je res škoda, da še ne vemo, da bi lahko seznam sestavili tudi z

```
>>> potence = [lambda x, p=i: x**p for i in range(4)]
```

Še en preprostejši in en bolj zapleten primer si privoščimo. Začnimo s preprostim.

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

V C in C++ bi morali za kaj podobnega predprocesorju del kode zapreti v `#if`, `#else`, `#endif`, da prevajalnik ne bi *prevedel* napačne funkcije. V Pythonu zadošča običajni `if`, ki poskrbi, da tolmač ne bi *izvedel* napačne definicije funkcije. Povejmo še, da to ni najelegantnejša rešitev, sploh pa ni potrebna, saj takšna funkcija že obstaja.

In še zapleteni. Bi znali napisati funkcijo `potroji(f)`, ki kot argument dobi neko poljubno funkcijo `f`, kot rezultat pa vrne novo funkcijo, ki računa natanko isto kot `f`, vendar vedno vrne trikrat več?

Za zdaj predpostavimo, da `f` pričakuje en sam argument.

```
def potroji(f):
    def f3(x):
        return 3*f(x)
    return f3
```

Medtem ko en bralec ni prepričan, da stvar dela, drugi še ne razume popolnoma, kaj naj bi sploh delala. Odgovorimo kar obema hkrati. Imamo funkcijo `sqrt`; sestavimo novo funkcijo, `s3`, ki vrača trikrat toliko kot `sqrt`.

```
>>> from math import sqrt
>>> s3 = potroji(sqrt)
>>> s3(25)
15.0
>>> s3(196)
42.0
```

Poskusimo še z eno funkcijo:

```
>>> fib3 = potroji(fibonacci)
>>> fibonacci(5)
8
>>> fib3(5)
24
```

Pomislimo, kaj dela funkcija. Pokličemo, recimo, `potroji(sqrt)`. Argument `f` ima torej vrednost `sqrt`. Funkcija `potroji` nato definira, torej, *izvede definicijo* funkcije `f3` in vrne pravkar definirano funkcijo. Kakšno funkcijo `f3` pa definira? Zelo preprosto funkcijo, takšno, ki vsebuje le `return 3*f(x)`. Ker je `f` v tem primeru `sqrt`, bo `f3` vračala `3*sqrt(x)`.

Naslednjič smo poklicali `potroji(fibonacci)`. Vse je steklo enako: `potroji` je definirala `f3` in jo vrnila, le da je `f` v tem primeru `fibonacci` in `f3`-jev `return` je `return 3*fibonacci(x)`.

Vem, dobro vem, kaj muči tega, ki je vajen jezikov, ki podpira lokalne funkcije. Ne, `f3` ni lokalna funkcija, četudi je videti, kakor bi bila v kakem drugem jeziku videti lokalna funkcija. Definicija funkcije se ne prevede, temveč izvede.

Zdaj pa še izboljšave. Kaj, če funkcija `f` ne sprejema le enega temveč več argumentov, morda celo poimenskih? Tudi to znamo.

```
def potroji(f):
    def f3(*args, **kwargs):
        return 3*f(*args, **kwargs)
    return f3
```

Naša `f3` zdaj sprejema poljubno število mestnih in poimenskih argumentov; z njimi pač pokliče funkcijo `f`. Ta, ki kliče `f3`, mora pač popaziti, da bo poslal točno takšne argumente, kot jih mara `f`.

Za konec pa še priznajmo, da je `f3` preprosta funkcija, ki ne zasluži imena.

```
def potroji(f):
    return lambda *args, **kwargs: 3*f(*args, **kwargs)
```

Kar smo počeli v tem razdelku, se zdi vsem prišlekom iz Cju podobnih jezikov imenitno, tem, ki niso zaprti v Cjevski plot, pa nič takšnega. Računalniški jezikoslovec bi rekel, da je funkcija v Pythonu *prvorazredni objekt* in s tem

povedal vse, kar je povedati. Vse, kar lahko počnemo z "navadnimi" objekti – jih shranjujemo v sezname, podajamo kot argumente, ustvarjamo nove in brišemo stare, jih vračamo kot rezultat – lahko počnemu tudi s funkcijami, saj so tudi funkcije čisto navadni objekti.

Dekoratorji

Dekoratorji so nekoliko povezani s tem, s čimer smo se igrali pravkar. Včasih bi želeli funkcijo “oviti” v kodo, ki bi se izvedla pred ali po njenem klicu ali obakrat. Denimo, da bi si želeli pri razhroščevanju pomagati s funkcijo, ki bo ob klicu nekaterih (izbranih) drugih funkcije izpisala argumente in rezultat. Lahko bi bila takšna.

```
def debug(f):
    def nf(*x, **y):
        print(f.__name__, x, y, end=" ")
        res = f(*x, **y)
        print("->", res)
        return res
    return nf
```

Funkcija kot argument dobi funkcijo in kot rezultat vrne novo funkcijo, ki izpiše ime stare funkcije in argumente, nato pokliče podano funkcijo, izpiše rezultat in ga vrne.

Zdaj napišimo funkcijo, ki bi jo radi razhroščevali na ta način in jo ovijmo.

```
def sqrmul(a, b):
    return b * a**2

sqrmul = debug(sqrmul)
```

Reč deluje takole

```
>>> sqrmul(2, 3)
sqrmul (2, 3) {} -> 12
>>> sqrmul(3, "Ana")
sqrmul (3, 'Ana') {} -> AnaAnaAnaAnaAnaAnaAnaAnaAna
```

V manjših programih tega ne potrebujemo veliko, v večjih projektih pa pogosto ovijamo veliko funkcij na enak način, zato obstaja za bližnjica: dekoratorji, s katerimi funkcijo ovijemo kar ob njeni definiciji, takole

```
@debug
def sqrmul(a, b):
    return b * a**2
```

Znaku `@` mora slediti ime funkcije, ki vrne funkcijo, tako kot zgoraj. Kot bomo videli v poglavju o razredih, so s pomočjo dekoratorjev implementirane statične metode razredov.

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. 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).

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

4. Napiši funkcijo `vsota(f, g)`, ki kot argument dobi funkciji `f` in `g`, kot rezultat pa vrne funkcijo, ki za vsak `x` vrne vsoto funkcij `f(x)+g(x)`. Se razumemo? Ne? Prav, rad bi funkcijo vsota, ki bi mi omogočala tole.

```
def kvadrat(x):
    return x**2
```

```
def manjpol(x):
    return -x/2

kmp = vsota(kvadrat, manjpol)

print(kmp(12))
```

Program mora izpisati isto, kot če bi poklical print(kvadrat(12) + manjpol(12))

Rešitve

1. Rešitev je šaljivo preprosta.

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

2. 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.

3. Naj bo seznam shranjen v `s`. Uredimo ga z

```
s.sort(key=lambda x: x.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 s]
ll.sort()
s = [x[1] for x in ll]
```

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

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

4. V poglavju o nizih smo že napisali funkcijo, ki iz EMŠO naredi terko z rojstnim datumom. Takšno terko bomo uporabili kot ključ, le da bomo postavili leto pred mesec in tega pred dan v mesecu. Na problem leta 2000 bomo v imenu nazornosti pozabili.

```
emso.sort(key=lambda x: (x[4:7], x[2:4], x[:2]))
```

5. Ta vaja od reševalca zahteva, da razume, kar smo počeli v funkciji `potroji`.

```
def vsota(f, g):
    def h(x):
       return f(x) + g(x)
    return h
```

Funkcija vsota mora vrniti funkcijo. In to ne vedno iste funkcije, temveč vsakič drugo. In edini način, da to doseže, je, da pač vsakič, ko jo pokličemo, definira (novo) funkcijo.

Rešitev spet lahko dopolnimo tako, da dovolimo, da `f` in `g` (z njima pa `h`) sprejemajo tudi več kot en argument.

```
def vsota(f, g):
    def h(*a, **aa):
       return f(*a, **aa) + g(*a, **aa)
    return h
```

In, navsezadnje, nobene potrebe ni po tem, da bi imela funkcija `h` ime. Rešitev naše naloge je lahko kar:

```
def vsota(f, g):
    return lambda *a, **aa: f(*a, *aa) + g(*a, *aa)
```

Moduli

Podobno kot razredi, ki nas še čakajo, so tudi Pythonovi moduli zelo lahkotni. Pythonov modul je le imenski prostor, v katerem živijo objekti – funkcije, razredi, drugi moduli...

Če imamo, recimo, modul `matematika`, v katerem je funkcija `fibonacci`, pridemo, kot že dolgo vemo, do funkcije `fibonacci` z `matematika.fibonacci`. Tisto pred piko je imenski prostor in tisto za njo, ime objekta iz tega prostora. Pred tem moramo modul seveda uvoziti, kar storimo z `import matematika`.

Kaj stori `import`? Nekaj zelo preprostega: sestavi imenski prostor in "v njem" požene program v podani datoteki. Tako bi `import matematika` le izvedel program v datoteki *matematika.py*. Da ga požene v imenskem prostoru matematika, pa pomeni, da je globalni prostor programa v resnici znotraj imenskega prostora. Program definira funkcijo `fibonacci`, ki je, kar se tiče lenjega, globalna funkcija; od zunaj gledano gre za funkcijo znotraj modula-imenskega prostora `matematika`. Recimo, da bi program *matematika.py* poleg tega definiral globalno spremenljivko `pi = 3.141`. Spremenljivka bi bila globalna le zanj, od zunaj pa bi jo videli znotraj prostora `matematika`, torej kot `matematika.pi`.

Zdaj, ko vemo, kako deluje `import`, znamo napisati tudi modul. Na nek način smo jih že pisali, le uvažali jih nismo. Le napišemo program z vsemi funkcijami, razredi in čimerkoli že, mu damo takšno ime, kot želimo, da ga ima naš modul, in to je to.

Uvažanje modula torej izvede modul, tako kot se izvede program, razlika je le v tem, da ostanejo funkcije, spremenljivke, razredi in vse ostalo, kar definira, v drugem imenskem prostoru.

Uvažanje ni niti približno podobno ukazu `include` ali `import` v kakem prevajanem jeziku. Prevajalniku za C `include` pove, da mora na tem mestu *prevesti* podano datoteko. Pythonu `import` pove, da mora na tem mestu *izvesti* modul. Tako kot definicijo funkcije lahko damo znotraj zanke, pogojnega stavka

ali druge funkcije, lahko tudi `import` postavimo v zanko, pogojni stavek ali funkcijo in izvedel se bo med izvajanjem programa, na prav tistem mestu.

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, recimo kaj izpisuje, se bo ob uvozu to tudi izpisalo, če piše ali briše datoteke, se bo ob uvozu napisala ali izbrisala datoteka.

Uvažanje v lasten imenski prostor. Včasih se nam zazdi dobra ideja, da bi se znebili imena modula in pike ter funkcije klicali kar brez tega. To storimo, kot smo že kdaj storili:

```
from fibonacci import *
```

S tem vsebino modula uvozimo kar v "svoj" imenski prostor. Funkcije `fibonacci` in spremenljivke `pi` zdaj ne bomo klicali `fibonacci.fibonacci` in `fibonacci.pi`, temveč kar `fibonacci` in `pi`.

Dasiravno to zveni privlačno, ta ideja navadno ni dobra. Če ne zaradi drugega, zato, ker se zlahka primeri, da dva modula z istim imenom poimenujeta dve različni stvari in drugi uvoz bo povozil prvega.

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:

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

Za modul `math` sicer ni neobičajno in se ne šteje za grdo niti, če v svoj imenski prostor uvozimo vse njegove funkcije.

```
from math import *
```

Izrazi, kot je `(r*cos(phi), r*sin(phi))`, so veliko preglednejši brez gore `math`ov.

Ponovno uvažanje. Vsak modul se uvozi le enkrat, četudi večkrat izvedemo `import`.[19] Tako je smiselno iz vsaj dveh vzrokov. Nekateri moduli imajo inicializacijo, ki se sme izvesti le enkrat ali pa vzame veliko časa. Večina modulov uvaža druge module in ko bi ob vsakem `import` ponovno sprožili celotno verigo, to ne bi bilo dobro.

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__":
    import unittest
    unittest.main()
```

Ko se ob uvažanju modula izvede koda, ki jo vsebuje (definicije funkcij ipd.), bo na koncu prišel na vrsto še tale `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 poženejo testi.

Prevedeni moduli. Po prvem uvažanju modula se na disku kar od nikoder pojavi datoteka z enakim imenom, a končnico .pyc. To je datoteka z modulom, prevedenim v bajtno kodo za Pythonov navidezni računalnik. Ko bo naslednjič potrebno brati modul z diska (na primer ob naslednjem zagonu Pythona) bo Python preveril, ali je datoteka .pyc mlajša od .py in v tem primeru uporabil kar že prevedeni .pyc, kar bo pospešilo uvoz. Za datoteke .pyc nam 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.

---

[19] Razen, če ga sami naložimo ponovno, tako da uporabimo funkcijo `imp.reload`.

Kje Python išče module? Najprej v trenutnem direktoriju, potem v direktorijih, ki so našteti v okoljski (ali, kot nas v pomanjkanju pametnejšega dela poskušajo naučiti novodobni varuhi slovenskega pravorečja, *okolijski*) spremenljivki PYTHONPATH, in nazadnje v direktoriju, katerega ime in lokacija sta odvisna od operacijskega sistema in namestitve Pythona. Na MS Windows je to običajno *c:\python32\lib\site-packages*, na Linuxu je rad v */usr/lib/python3.2/site-packages*, na Os X pa je to odvisno od tega, kako (in kje in kolikokrat) je nameščen Python. Številka v poti, 32 oziroma 3.2, je seveda odvisna od različice Pythona.

Poleg tega lahko Pythonu povemo za direktorije z moduli še na več drugih načinov, od datotek .pth v direktoriju site-packages do sistemskega registra v MS Windows, vendar je vse to nepotrebno in iz mode.

Vsi direktoriji, kjer Python išče module, so zbrani v spremenljivki `sys.path`.

```
>>> import sys
>>> sys.path
['', 'c:\\d\\ai\\orange',
'C:\\WINDOWS\\system32\\python32.zip', 'C:\\Python32\\DLLs',
'C:\\Python32\\lib', 'C:\\Python32\\lib\\plat-win',
'C:\\Python32', 'C:\\Python32\\lib\\site-packages']
```

Spremenljivka `sys.path` je najobičajnejši seznam, torej lahko z `append`, `insert` ali kako drugače dodamo (ali, če želimo, odstranimo) kak imenik. Vendar tega seveda ne počnemo, če ni nujno potrebno.

Hierarhija modulov. Če sestavljamo večji paket, ga bomo morda želeli razdeliti na "podmodule". Recimo, da bi radi imeli poleg modula `matematika` podmodula `matematika.zaporedja` in `matematika.priblizki`, ki bi ju uvažali, recimo z `import matematika.zaporedja`.

Direktoriji, ki vsebujejo datoteko z imenom *__init.py__*, se obnašajo kot moduli; ime direktorija je ime modula, vsebina modula pa je tisto, kar se nahaja v *__init__.py*. Datoteke v tem direktoriju so podmoduli. Storiti nam je tole: znotraj direktorija *site-packages* naredimo poddirektorij *matematika*, datoteko *matematika.py* premaknemo vanj in jo preimenujemo v *__init.py__*. S tem nismo spremenili ničesar; funkcijo `fibonacci`, ki je zdaj znotraj `__init__.py`, še vedno dobimo tako, da napišemo `import matematika` in kličemo `matematika.fibonacci`. Zdaj pa znotraj tega direktorija sestavimo datoteki

zaporedja.py in *priblizki.py*: njuna vsebina bo vidna kot modula `matematika.zaporedja` in `matematika.priblizki`. Če torej premaknemo funkcijo `fibonacci` iz *__init__.py* v *zaporedja.py* (ki se nahaja znotraj direktorija *matematika*), bomo napisali `import matematika` ali `import matematika.zaporedja`, ter klicali funkcijo `matematika.zaporedja.fibonacci`. Lahko pa jo uvozimo v lasten prostor s `from matematika.zaporedja import fibonacci` in jo kličemo le `fibonacci`.

Če se naš paket še bolj razbohoti, lahko naredimo znotraj poddirektorija matematika nove poddirektorije in tako naprej.

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 is 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(s):
    if not s:
        raise ZeroDivisionError("seznam je prazen")
    return sum(s) / len(s)
```

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

Za lovljenje izjem uporabimo blok *try-except*.

```
def izpisiPoprecje(s):
    try:
        p = poprecje(s)
        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` zanje. Če izpustimo tip, kar je priporočljivo samo za garažne skripte, ne pa za resne programe, polovimo vse (preostale) izjeme.

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

Lovimo lahko tudi več napak hkrati, tako da jih preprosto naštejemo, recimo `except (NameError, KeyError, IndexError)`.

Razredi so zloženi v hierarhijo; praoče vseh je `BaseException`, skoraj celotna hierarhija pa visi na njegovem potomcu `Exception`. Odtod naprej gredo stvari bolj v širino kot globino. Še najbolj pester je `ArithmeticError`, ki se deli na `FloatingPointError`, `OverflowError` in `DivisionByZeroError`. Če napake lovimo z `except ArithmeticError`, bomo, jasno, polovili vse tri podžanre aritmetične napake. Če se že odločimo loviti vse napake, se je pametno omejiti vsaj na `Exception`, ki ne vključuje `KeyboardInterrupt` in `SystemExit`.

Če nam obstoječe izjeme ne zadoščajo, lahko definiramo tudi svoje. Izpeljane morajo biti iz primernega razreda v hierarhiji, najpogosteje iz `Exception`. Celotno hierarhijo in opise situacij, kjer naj bi sprožali posamezno napako, najdete v dokumentaciji.

K tipu napake, ki jo lovimo, lahko dodamo še ime, ki mu bo prirejen objekt, ki predstavlja izjemo in včasih – odvisno od vrste izjeme – vsebuje dodatne informacije o tem, kakšna napaka se je zgodila. Če nič drugega, se zna izpisati.

```
try:
    open("datoteka.txt")
except IOError as napaka:
    print("Napaka pri branju: %s" % napaka)
```

Objekt, ki smo ga poimenovali `napaka`, ima vsaj še atribut `args`, ki vsebuje terko z argumenti, ki jih je dobil konstruktor napake. Njegova vsebina je v gornjem primeru, ker datoteka ne obstaja, enaka `(2, 'No such file or directory')`; prvi element je koda napake, drugi njen opis; argumenti napake vrste `IOError` so pač videti tako. V napaki `DivisionByZero`, ki smo jo sestavili zgoraj, bi imel `args` en sam element in sicer niz `"seznam je prazen"`.

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

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

Poleg `except` lahko, 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 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`.

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, predstavljenega z objektom izpeljanim iz razreda `Warning` (ta pa je izpeljan iz `Exception`). 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.

Spremenljivke, imena, objekti

Python smo doslej spoznali kot udoben jezik. Zelo malo administracije, preproste definicije funkcij, vdelane visokonivojske podatkovne strukture, kot seznami in množice, preprosto delo z različnim kodnimi nabori, posrečeno indeksiranje, vse to je lepo in vabljivo, kar se tiče filozofije in konceptov pa nič pretresljivega. Pač pa nas pretresljiva reč čaka v tem poglavju.

Pogovor začnimo ob vprašanju, ki smo se mu izognili ob funkcijah: se argumenti prenašajo po vrednosti ali po referenci? Če kot argument podamo spremenljivko in če funkcija spreminja vrednost argumenta: se s tem spremeni tudi vrednost spremenljivke? Odgovora ni, ker vprašanje ni smiselno. Smiselno je namreč, očitno, lahko le v jezikih, ki imajo spremenljivke. Če je spremenljivka tisto, čemur rečemo spremenljivka v Cju (in iz njega izpeljanih jezikih vključno z, recimo, Phpjem), potem Python spremenljivk nima.

Python nima spremenljivk

Spremenljivka v Cju je le poimenovan pomnilniški naslov. S kazalci ali referencami (v jezikih, ki jih imajo) lahko dosežemo, da ima isti naslov več imen, vendar je eno od njih navadno "pravo", namreč tisto, s katerim je spremenljivka "definirana" in je prevajalnik nekje pripravil prostor zanjo, ostala imena pa so le "sinonimi", reference nanjo.

V Pythonu ni tako. Python je objektni jezik, zato je vse v njem objekt: ne le seznami in slovarji, datoteke in nizi, ne le števila (za razliko od C++ in primitivov v Javi), celo moduli, funkcije in razredi so objekti. Tisto, čemur smo doslej (in bomo tudi poslej, a z malo več pazljivosti) rekli spremenljivke, so v resnici le imena, ki se nanašajo na objekte.

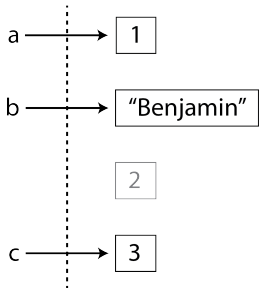
Torej: na eni strani imamo objekte, na drugi imena zanje. Vsak objekt ima eno ali več imen, ali pa lahko do njega pridemo prek drugih, poimenovanih objektov, kot, recimo z "imenom" `s[2]` pridemo do tretjega objekta seznama `s`. Če ima

objekt več imen, ni eno od njih pravo, osnovno, ostalo pa so zgolj sinonimi. Ne, vsa imena so enakovredna.

V jezikih, v katerih je potrebno spremenljivke deklarirati, bi z, recimo, `int a;` naročili prevajalniku pripraviti toliko pomnilnika, kot ga zahteva tip `int` in ga poimenovali `a`. Prirejanje, `a = 42`, tedaj preberemo kot "na mesto v pomnilniku, ki sem ga poimenoval `a`, zapiši `42`". V Pythonu definicij tako ali tako ni, prirejanje pa ima popolnoma drugačen pomen. Desna stran prirejanja je izraz in rezultat izračuna izraza je vedno nek objekt, v gornjem primeru objekt, ki predstavlja število `42`. Na levi strani je vedno ime, bodisi preprosto, tako kot `a`, bodisi posredno ime, kot `s[2]`, če bi napisali, recimo, `s[2] = 42`. Prirejanje priredi imenu (`a` ali pa `s[2]`) objekt.

Kaj naredi naslednji program?

```
a = 1
b = "Benjamin"
c = 2 + a
```

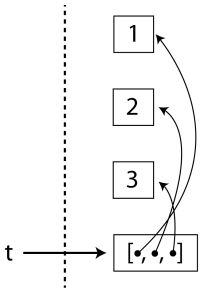


Naredi, kar kaže slika na desni. Prva vrstica sestavi objekt `1` in ga priredi imenu `a`. Druga naredi niz `"Benjamin"` in ga priredi imenu `b`. V tretji se izraz na desni izračuna tako, da Python sestavi objekt `2`, nato sešteje ta objekt in objekt, na katerega se nanaša ime `a`. Rezultat je objekt `3`. Prireditveni stavek priredi objekt `3` imenu `c`.

Za zdaj – nič posebnega. Površni študent – ah, kako jih mrgoli! – bi rekel, da gre vendar le za besede, v resnici pa se vse dogaja enako kot v "običajnih" jezikih. Nikar tako hitro. Ogledali si bomo nekaj presenetljivih primerov. To seveda niso stvari, v katere bi se človek vsakodnevno zaletel, temveč le izbrani primeri, ki dobro ilustrirajo, kako delujejo imena. Če se bo zdelo zmedeno: ni. V resnici je preprosto in lepo pokrije vse čudne reči, ki smo jih drugod vajeni reševati s kazalci in referencami.

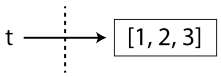
Preden zares začnemo, se moramo dogovoriti, kako bomo risali sezname. Recimo, da napišemo

```
t = [1, 2, 3]
```



Dobili bomo, kar kaže slika: tri objekte, `1`, `2` in `3`, ter četrti objekt, seznam, ki vsebuje te tri objekte. Takšno risanje je nepraktično, zato bomo številke risali kar naravnost v sezname.

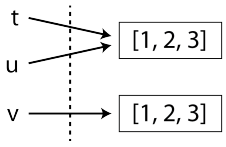
```
t = [1, 2, 3]
```



Kadar bodo v seznamih kake druge stvari kot številke, pa bomo risali, tako kot je res – iz seznama bo vodila puščica na objekt (ali objekte), ki jih vsebuje.

Primer 1. Sestavimo tri sezname, `t`, `u` in `v`.

```
t = [1, 2, 3]
u = t
v = t[:]
```



Tri? V resnici imamo dva seznama. Prvi seznam sestavimo v prvi vrstici, tu ni dvomov. Kaj naredi druga vrstica? Spomnimo se: prirejanje izračuna izraz na desni strani in ga priredi imenu na levi. Na desni strani enačaja piše `t`; rezultat "izračuna" je "tisti seznam, ki smo ga poimenovali `t`". Torej je `u` isto kot `t`. Seznam, ki smo ga naredili v prvi vrstici, ima tako dve imeni, rečemo mu lahko `t` ali `u`. V tretji vrstici pa sestavimo nov seznam, v katerem so vsi elementi `t`-ja, od prvega do zadnjega; rezine pač vedno sestavljajo nove sezname. Temu, novemu seznamu smo dali ime `v`.

Vsi trije seznami so enaki.

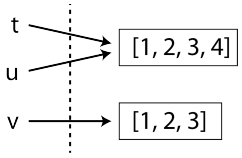
```
>>> t == u
True
>>> t == v
True
```

Niso pa isti. Imeni `t` in `u` se nanašata na en in isti objekt, ime `v` pa na drugega. Ali se dve imeni nanašata na isto, lahko preverimo z operatorjem `is`; operator `is` je podoben operatorju ==, le da prvi preverja *istost*, drugi pa samo *enakost*.

```
>>> t is u
True
>>> t is v
False
```

Zdaj seznamu `u` dodajmo še en element. Kaj dobimo?

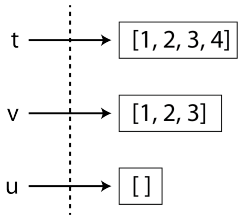
```
t = [1, 2, 3]
u = t
v = t[:]
u.append(4)
```



S tem, ko smo spreminjali `u`, smo spremenili tudi `t`, saj gre za eno in isto reč. Seznam, ki ga imenujemo `v`, je ostal takšen, kot je bil.

Zdaj pa priredimo `u`-ju prazen seznam.

```
...
u = []
```



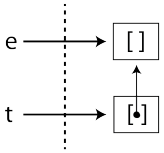
Tule je prvi kamen spotike za vse, ki mislijo, da so `t`, `u` in `v` spremenljivke. Niso, vsaj tule se tega zavedamo. To so le imena, ki jih dajemo objektom (oziroma obratno, imenom dodeljujemo objekte). Ko rečemo `u = []`, *ne spreminjamo vrednosti spremenljivke `u`* (Python nima spremenljivk!), temveč *dodeljujemo nek objekt imenu `u`*. Razumite to in ste zmagali. Odtod naprej gre vse po istem kopitu.

Če imenu `u` priredimo nov objekt, na objekt, ki ga imenujemo `t`, to ne vpliva. Ostaja tak kot je bil (saj ga nihče ni spreminjal) in tudi ime `t` se še vedno nanaša nanj.

Podobno bi se zgodilo, če bi prazen seznam priredili `t`-ju: v tem primeru bi `u` ostal tak, kot je bil. Med imenoma `t` in `u` ni nobene razlike; če kdo razmišlja o `u`-ju kot o nekakšnem "aliasu", rezervnem, dodatnem imenu, referenci na `t`, razmišlja narobe.

Primer 2. Naredimo prazen seznam, poimenujmo ga `e`; potem naredimo seznam, ki vsebuje ta seznam.

```
e = []
t = [e]
```

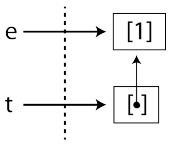


Da sta `e` in ničti element, e in `t[0]`, res eno in isto, se hitro prepričamo.

```
>>> e is t[0]
True
```

Posledica te istosti je jasna: kar bomo torej počeli z `e`, se bo zgodilo tudi s `t[0]`, saj je to eno in isto.

```
...
e.append(1)
```

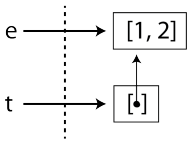


Izpišimo `t`, da bomo videli, da je res, kakor smo narisali.

```
>>> t
[[1]]
```

In obratno, kar počnemo s `t[0]`, se zgodi tudi z `e`.

```
...
t[0].append(2)
```

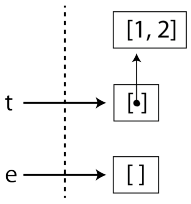


Zdaj izpišimo `e`, da vidimo, da se je res spremenil.

```
>>> e
[1, 2]
```

Za konec, tako kot v prejšnjem primeru, spremenimo `e`.

```
...
e = []
```

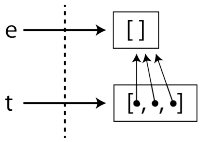


Čemu podčrtane besede? Ker so dvoumne. Ime `e` se je nanašalo na določen objekt (seznam, ki vsebuje enko). Ta seznam smo pustili pri miru, torej `e`-ja nismo spreminjali. Pač pa smo ustvarili nek nov objekt in ga priredili imenu `e`. Ker je ostal objekt, na katerega se je prej nanašalo ime, nedotaknjen, se tudi `t` ni spremenil: `t` še vedno vsebuje isti objekt in ta objekt je še vedno seznam, ki vsebuje enico in dvojko, kot kaže slika in potrjuje spodnji poskus:

```
>>> t
[1, 2]
```

Primer 3. Sezname lahko, vemo, množimo s števili. Kar dobimo, je seznam, ki vsebuje večkrat ponovljen prvi seznam.

```
e = []
t = [e]*3
```

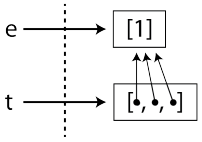


Python ne kopira objektov, zato `t` ne vsebuje treh praznih seznamov, temveč trikrat vsebuje isti prazen seznam, znan tudi pod imenom `e`.

```
>>> t[0] is e
True
>>> t[1] is e
True
>>> t[2] is e
True
```

Ker gre za trikrat, ne, štirikrat isti seznam, se z enim spreminjajo vsi štirje.

```
...
e.append(1)
```



Isto (da, *enako* bi bila tu prešibka beseda) bi se zgodilo tudi, če bi namesto k seznamu `e` dodali enico k seznamu `t[0]`, `t[1]` ali `t[2]`.

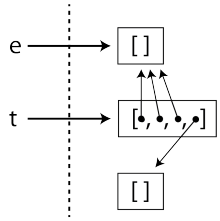
Kakšen je zdaj `t`, vemo. Le prepričajmo se:

```
>>> t
[[1], [1], [1]]
```

In zdaj se vprašajmo: se je `t` spremenil ali ne? Tisti, ki na to vprašanje odgovorijo z "da" ali z "ne", ne razumejo. Pravilen odgovor je, da vprašanje ni povsem jasno. V bistvu se `t` ni spremenil, saj ga tudi ni nihče spreminjal: `t` še vedno vsebuje natančno isto reč kot prej, trikrat en in isti objekt, namreč seznam, ki ga poznamo tudi pod imenom `e`. Res pa se je spremenil ta seznam. Torej je `t` po eni strani ostal enak, po drugi pa se je spremenilo tisto, kar vsebuje.

Primer 4. Naredimo nekaj podobnega kot prej: prazen seznam in nov seznam, v katerem bo trikrat ta, prazni seznam. V slednjega dodajmo še en prazen seznam.

```
e = []
t = [e]*3
t.append([])
```

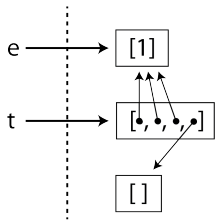


Bistvo vaje je v tem, da `t` sicer vsebuje štiri prazne sezname, vendar so prvi trije isti, zadnji pa le enak.

```
>>> t
[[], [], [], []]
>>> t[0] is e
True
>>> t[1] is e
True
>>> t[2] is e
True
>>> t[3] is e
False
```

Če spremenimo `e` (se pravi, če vanj dodamo enico), se spremenijo prvi trije elementi `t`ja, saj gre za isti seznam, četrti (`t[3]`) pa ostane, kakršen je bil, namreč prazen.

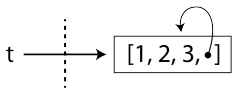
```
...
e.append(1)
```



```
>>> t
[[1], [1], [1], []]
```

Primer 5. More seznam vsebovati sam sebe? Tega ni težko preskusiti.

```
t =  [1, 2, 3]
t.append(t)
```



Koliko elementov ima zdaj seznam `t`? Neskončno? Dajte no, noben seznam ne more imeti neskončno elementov, to bi ne šlo v pomnilnik. Samo tri ima, namreč 1, 2, 3 in še tisti seznam, ki ga poznamo tudi pod imenom `t`.

```
>>> len(t)
4
```

Element z indeksom 3 je seveda `t` sam:

```
>>> t[3] is t
True
```

Pa ga lahko izpišemo? Do neke mere.

```
>>> t
[1, 2, 3, [...]]
```

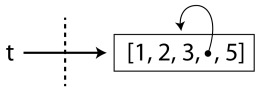
Python je zvit. Tretjega elementa ne izpisuje, saj ve, kam bi to peljalo. Pa ga lahko, ta, tretji element, izpišemo sami?

```
>>> t[3]
[1, 2, 3, [...]]
```

Jasno, tretji element je tako ali tako `t` sam: če izpišemo `t[3]`, je to *isto*, kot če bi izpisali `t`.

Pa dodajmo v `t` še en element.

```
...
t.append(5)
```



Zdaj ima `t` še en element več, prav tako `t[3]`, saj je to še vedno ena in ista reč.

```
>>> t
[1, 2, 3, [...], 5]
>>> len(t)
5
>>> len(t[3])
5
```

Definicije funkcij

Bralca bržkone prešine strašljiva misel, da bomo, tako kot smo izvedeli, da Python nima spremenljivk, zdaj slišali, da tudi funkcij nima. Brez strahu, ima jih. Prav toliko, kolikor ima spremenljivke. Python je objektni jezik brez fige v žepu, zato so tudi funkcije objekti. Besede `len` in `fibonacci` in `math.sqrt` pa so imena za te objekte. Če po `a = 42` rečemo, da je a spremenljivka (njena vrednost pa je število, namreč `42`), bo treba po isti logiki reči tudi, da je `fibonacci` spremenljivka, njena vrednost pa je objekt, ki se ga da poklicati in vrne Fibonaccijevo število.

Poglejmo še enkrat definicijo in povejmo, kar smo že povedali, ko smo pisali o funkcijah, a na drug način.

```
a = 42

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

Prvo vrstico smo dodali zaradi lepšega. Spomnimo se: naredi objekt `42` in mu priredi ime `a`. Definicija funkcije pa sestavi objekt s funkcijo (vsebuje, če hočete, zadnje štiri vrstice, v neki prevedeni obliki) in mu priredi ime `fibonacci`.

Če večkrat definiramo isto funkcijo, je to tako, kot če a-ju večkrat priredimo vrednost. Prav tako ni nič narobe, če je `f` nekje funkcija, pet vrstic kasneje pa številka. Vsaj formalno gledano; praktično gledano je to seveda grdo.

Vsa čudesa, ki smo jih prej kazali z drugimi objekti, pretežno števili in seznami, veljajo tudi za funkcije, saj je tudi ime funkcije le ime, kot vsako drugo ime v Pythonu. Funkciji lahko dodelimo tudi dodatno ime

```
...
fib2 = fibonacci
```

ali pobrišemo starega

```
...
del fibonacci
```

Vse, kar smo pisali o imenih objektov, velja tudi za funkcije, saj so prav tako objekti.

Lahko tudi funkcije lahko "umrejo"? Seveda. Če dodamo še `del fib2` ali `fib2 = "Tine"`, in če funkcija ni znana še pod kakim drugim imenom ali se ne valja še v kakem seznamu, modulu ali kje drugje, bo šla v smeti. Funkcija je objekt kot vsi drugi in tudi zanjo veljajo ista pravila kot za vse druge objekte. *Vsi objekti so umrljivi in funkcija je objekt, torej je funkcija umrljiva*, kot je potegnilo že staremu Aristotlu.

Podobno je, kot bomo videli, z moduli in razredi. Definicija razreda sestavi določen objekt (ta je tipa `classobj`) 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.[20]

[20] 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.

Argumenti funkcij

V glavah bralcev pa še vedno trmasto kljuva: kako torej funkcije dobivajo argumente, po vrednosti ali po referenci?

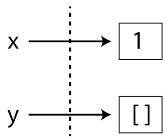
Denimo, da imamo funkcijo, definirano z `def f(a, b):` (in tako naprej). Ob klicu funkcije z dvema argumentoma moramo za argument podati dva *objekta*. Prvemu objektu bo funkcija dala ime *a*, drugemu *b*. Povsem v duhu tega, kar smo se pogovarjali zgoraj, bosta `a` in `b` le dve imeni za objekta, ki ju je funkcija dobila kot argument.

To ni ne klic po vrednosti, ne po referenci. Za ilustracijo definirajmo funkcijo, ki sprejme dva argumenta in ju malo spremeni.

```
def f(a, b):
    a = 2
    b.append(3)
```

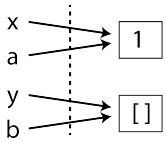
Vzemimo zdaj eno številko in en seznam.

```
x = 1
y = []
```



Pokličimo funkcijo, kot argumenta ji dajmo `x` in `y`.

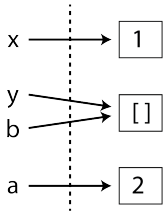
```
...
f(x, y)
```



Ob klicu se imenoma argumentov, `a` in `b`, priredita objekta, poslana kot argumenta. Zgodi se isto, kot če bi rekli `a = x` in `b = y`. Če bi namesto z `f(x, y)` funkcijo poklicali z `f(sin(x)*2, ", ".join(imena)`, bi se zgodilo isto, kot če bi na začetku funkcije rekli `a = sin(x)*2` in `b = ", ".join(imena)`.

Imeni `a` in `b` se nanašata na ista objekta kot imeni `x` in `y`. Dril od tod naprej nam je znan in nič nas ne sme več presenetiti. Funkcija najprej reče

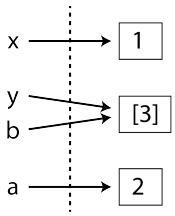
```
...
a = 2
```



Smo s tem spreminjali `a`? Takšnemu govorjenju se bomo danes, smo rekli, izogibali. Objekta, ki smo ga poprej imenovali `a` (in ga imenujemo tudi `x`), nismo spremenili, ostal je tak kot je bil (in `x` z njim). Pač pa smo naredili objekt `2` in ga priredili imenu `a`.

Sledi spreminjanje `b`-ja.

```
...
b.append(3)
```



Tule v resnici spreminjamo `b`, točneje, objekt, ki ga imenujemo `b`. Ker ima isti objekt tudi ime `y`, se spreminja tudi objekt, ki ga imenujemo `y`. Če tako po vrnitvi iz funkcije izpišemo `x` in `y`, izvemo

```
>>> x
1
>>> y
[3]
```

V tem kontekstu povejmo, kar bi sicer lahko že zgoraj: s prirejanjem imenom spreminjamo le imena, ne pa objektov, na katera so se ta imena nanašala. Če imamo nekaj, čemur rečemo `a`, in nato `a`ju priredimo neko vrednost, se tisto, čemur smo prej rekli `a`, ne spremeni, saj prirejamo le vrednost imenu. Pač pa tisto, na kar se nanaša ime `a`, spreminjamo z `a`jevimi metodami (kot recimo `a.append`, če gre za seznam), z indeksiranjem (`a[0] = 42`) ali naslavljanjem atributov (`a.name = "Benjamin"`), o čemer bomo govorili, ko pridemo do objektov. Če določen tip nima metod, s katerimi bi ga bilo mogoče spreminjati – kot jih nimajo terke in nizi – objektov te vrste pač ni mogoče spremeniti.

Naloge

1. Napiši funkcijo `add(v, w)`, ki vrne vsoto (po elementih) seznamov `v` in `w` ter `iadd(v, w)`, ki k seznamu `v` (na mestu!) prišteje seznam `w`. Če je `v` krajši od `w`, naj ga funkcija dopolni z dodatnimi elementi.

2. 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
```

Rešitve

1. Naloga odlično ilustrira, zakaj v Pythonu nima smisla govoriti ne o prenosu argumentov po vrednosti, ne po referenci. V funkciji `add` ne smemo spreminjati seznama `v`: če napišemo, recimo, `v.extend([0]*(len(w)-len(v)))`, da bi ga podaljšali do dolžine `w`-ja, bo to narobe, saj funkcija `add` ne sme spremeniti `v`-ja. To je še nekako lahko požreti, mislimo si, pač, da Python ne prenaša argumentov po vrednosti. Po drugi strani v funkcij `iadd` ne smemo prirejati ničesar `v`-ju. Čim napišemo `v = nekaj`, se ime `v` ne bo več nanašalo na objekt, ki ga moramo spreminjati, zato tega objekta ne bomo mogli spremeniti, saj nam v funkciji ne bo več dosegljiv. Ta del je malo težje požreti, saj dokazuje, da se argumenti ne prenašajo niti po vrednosti.

   Manj elegantna, začetniška, rešitev za `add` je takšna:

```
def add(v, w):
    v = v[:] + [0]*(len(w)-len(v))
    w = w[:] + [0]*(len(v)-len(w))
    n = []
    for i in range(len(v)):
        n.append(v[i]+w[i])
    return n
```

   Oba seznama skopiramo (`v = v[:]`, `v` je tako ime kopije in ne ime originala, ki nam ga je prepovedano spreminjati) in jima dodamo toliko elementov, da sta

enako dolga (brez skrbi, množenje seznama z negativnim številom naredi prazen seznam). Potem sestavimo nov seznam in seštevamo.

Zrelejši programer v Pythonu se, kot vedno, spomni vsemogočne funkcije `zip`.

```
def add(v, w):
    n = []
    for e, f in zip(v, w):
        n.append(e+f)
    n += v[len(w):] + w[len(v):]
    return n
```

Funkcija `zip` sestavi seznam parov, ki je dolg toliko, kot je dolg krajši od seznamov. Pare seštejemo v nov seznam, ki mu na koncu dodamo še rep enega ali drugega seznama.

Vsak, ki da kaj nase, pa naredi tako, kot mi še ne znamo.

```
def add(v, w):
    return [e+f for e, f in zip(v, w)] + v[len(w):] + w[len(v):]
```

Zdaj pa še `iadd`. Ta je lažji, saj smemo (in moramo) spreminjati `v`.

```
def iadd(v, w):
    for i in range(min(len(v), len(w))):
        v[i] += w[i]
    v += w[len(v):]
```

K `v`-ju prištejemo toliko elementov, kolikor je dolg krajši od seznamov, nato pa mu pripnemo še `w`-jev rep – če je `w` daljši. Če ni, zadnja vrstica prišteje le prazen seznam.

Leni programer pa se spomni, da je pred kratkim sprogramiral `add` in reče le

```
def iadd(a, b):
    a[:] = add(a, b)
```

Bistvo trika je, da ne napišemo `a = add(a, b)`, saj bi tako priredili vsoto `a`-ja in `b`-ja imenu `a`, ne pa objektu, na katerega se nanaša ime `a`. Tu smo naredili tako, kot je prav: vsoto smo priredili rezini seznama, na katerega se nanaša ime `a`, in sicer rezini, ki obsega kar celoten seznam.

2. 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.

Operator del

Kot ve vsak, ima C++ operatorja `new` in `delete`, s katerima zasežemo in vrnemo kos pomnilnika. Java ima `new` (čeprav bi ga lahko tudi ne imela, gre zgolj za sintakso), nima pa `delete`. Python je anti-Java: nima operatorja `new`, pač pa ima `delete`, le da mu pravimo `del`.

A brez skrbi; kot bomo izvedeli takojci, ga ni potrebno uporabljati (zato ga tudi skoraj nikoli ne uporabljamo, razen za brisanje elementov seznamov, slovarjev in kar je še takega). Iz konteksta, v katerem smo ga privlekli na dan, pa lahko bralec že sklepa, da `del` nima opravka z objekti, temveč z imeni.

Operator `del` ne briše objekta, kot to počne `delete` v C++. Da tega ne more môči, se hitro prepričamo ob spodnjem razmisleku.

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

Če bi `del` pobrisal *objekt*, na katerega se nanaša ime `a`, na kaj bi se tedaj nanašalo ime `b`?

V resnici `del a` pobriše le ime `a`. In zakaj bi bilo to uporabno? Pojma nimam. Nemara se kdaj želimo znebiti kakega za pomnilnik napornega objekta, a to bi šlo tudi drugače, namreč, kot vemo, tako, da bi `a`-ju priredili kak manj zahteven objekt, recimo `None`.

Samodejno smetarjenje

Prej smo slutili, po prejšnjem razdelku vemo: Python sam pospravlja za nami. Smetar vrže objekt iz pomnilnika (pri čemer pokliče destruktor in tako naprej) takrat, ko se nanj ne nanaša več nobeno neposredno ali posredno ime. Ko do objekta ne moremo več, ga ne potrebujemo več.

Večina smetarjenja se zgodi ob prirejanju vrednosti imenom, ki so se poprej nanašala na druge objekte in ob vračanju iz funkcij, ko izginejo njena lokalna imena. Smetarjenje je izvedeno tako, da vsak objekt "ve", koliko drugih objektov ali imen kaže nanj. Ko pade števec na nič, je čas za odhod.

Poleg tega mora Python občasno opraviti še nekaj zamudnejšega: počistiti krožne reference. Če seznam vsebuje sebe, števec referenc ne bo nikoli padel na nič. Seznamov, ki neposredno vsebujejo sebe, sicer ne srečamo ravno za vsakim vogalom programa. Pač pa se pogosto zgodi, da en objekt vsebuje drugega, ta ima nekakšno povezavo s tretjim, ki vsebuje četrtega, ta pa na nek način spet ve za prvega. Tudi če "od zunaj" za takšne objekte ne ve nihče drug, vedo eden za drugega, zato njihovi števci referenc ne bodo nikoli padli na nič. Za pospravljanje takšnih ciklov poskrbi algoritem, ki se proži na vsake toliko novo sestavljenih objektov. Radovedni bralec se bo o njegovem delovanju, če ga zanima, poučil kje drugje. Ne bo mu žal, stvar je zanimiva.

Lokalne in globalne spremenljivke

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, ki zahteva razmislek, 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. Točneje, slovar povezuje imena z objekti. Razlika med Pythonom in prevajanimi jeziki je, da je v Pythonu slovar spremenljivk dostopen tudi programu samemu, ne le prevajalniku.

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 uporabili, ko smo spoznavali oblikovanje nizov. 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", Python ga sestavi šele, ko ga hočemo pogledati..

Slovar vseh lokalnih spremenljivk vrne funkcija `locals`. Definirajmo funkcijo

```
def f(a, b):
    c = 42
    print(locals())
```

in jo pokličimo

```
>>> f("Benjamin", [1, 2, 3])
{'a': 'Benjamin', 'c': 42, 'b': [1, 2, 3]}
```

Tudi ta slovar je lahko (ni pa vedno) nagoljufan. Vsekakor ga ne poskušajte spreminjati z namenom, da bi dodali nove lokalne spremenljivke. To že dolgo ne deluje več.

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), `str.__dict__` metode razreda `str`, `obj.__dict__` pa polja in njihove vrednosti objekta `obj`.

```
>>> class A:
...     def foo(self):
...         pass
...
>>> a = A()
>>> a.g = 42
>>> a.__dict__
{'g': 42}
>>> a.__dict__["n"] = 15
>>> a.n
15
>>> A.__dict__
{'__module__': '__main__', 'foo': <function foo at
0x1063b46e0>, '__doc__': None}
```

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.

Definicija razreda

Razredi v Pythonu delujejo praktično brez birokracije. Ni privatnosti in skrivanja, ni protekcije, ni prijateljstev.[21] 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 add_multiple(self, aList):
        for item, quantity in aList:
            self.add(item, quantity)

    # Spodnja funkcija (še) ne deluje
    def remove_multiple(self, aList):
        for item in aList:
            self.remove(item)
```

[21] Lepe navade ne morejo biti stvar prisile; le skozi okno poškilite na cesto, pa se boste prepričali.

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

Način, na katerega 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.

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. Če bi šlo za razred v C++ ali Javi, bi prirejali "spremenljivki" `content`, v resnici pa bi se to nanašalo na `this->content`. Če želimo v Pythonu prirejati "spremenljivki" `self.content`, moramo to eksplicitno povedati.

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 objekt postane 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. V jezikih, v katerih je potrebno vse deklarirati, teh dilem ni.[22]

Poljem (*field*) v Pythonu navadno rečemo atributi (*attribute*). V gornjem primeru je `content` torej *atribut* objekta `self`.

Podobno je z metodami: tudi ob klicu (lastnih) metod, moramo navesti `self`. Primer najdemo v metodi `add_multiple`, ki vestno pokliče `self.add` in ne le `add`.

---

[22] 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 je 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 tako motiti, da bi najraje še tam striktno uporabljal `this`.

Logika je, ponovno, v tem, da gre za metodo `add` objekta `self` in ne za neko funkcijo `add`, ki ne pripada nobenemu objektu ali razredu.

Naslednja nenavadnost Pythonovih razredov – ki je skladna s Pythonom, ki nima deklaracij – 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. Je prosto dodajanje polj grda programerska praksa in znak slabo načrtovanega jezika? Slab programer bo vedno našel priložnost, da piše slabo kodo. Dobremu 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.

Ker atributi objektov niso vnaprej določeni, imamo tudi funkcije, s katerimi preverimo, ali objekt `obj` ima atribut z imenom `name` (`hasattr(obj, name)`), funkcijo, s katero dobimo vrednost atributa z imenom `name` (`getattr(obj, name[, default])`), funkcijo, s katero mu nastavimo vrednost (`setattr(obj, name, value)`) in funkcijo, s katero ga pobrišemo (`delattr(obj, name)`). Pri tem je, recimo, `setattr(a, "foo", 42)` isto kot `a.foo = 42` in `delattr(a, "foo")` isto kot `del attr.foo`. V čem je potem prednost teh metod pred "normalnim" delom z atributi? Funkcijo `getattr` uporabimo na podoben način kot metodo `get`, ki jo imajo slovarji: z njo lahko določimo privzeto vrednost za primer, ko atribut ne obstaja. Druga prednost je ta, da s temi funkcijami podamo ime atributa kot niz. Če kdo (še) ne vidi, zakaj bi bilo to koristno, naj še malo programira, pa mu bo hitro prišlo prav (če dotlej ne pozabi, da to obstaja).

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 `fibonacci`jem, ki je predpostavljal, da je podane argumente mogoče seštevati. Če `fibonacci`u namesto dveh števil pomolimo dva

modula, bo javil napako – seštevanje modulov? prosim lepo! –, 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 (taka je privzeta vrednost argumenta v definiciji 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; toliko, da se ve, da gre oboje. Tu torej vse teče praktično enako kot v drugih jezikih, le nepotrebnega `new` ni.

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.

Bralec naj polista nazaj do definicije razreda `Basket`. Ta vsebuje metodo `remove_multiple`, ki kliče metodo `remove`. Če bi kaj podobnega storili v kakem prevajanem jeziku, bi se prevajalnik razjezil, da metode `remove` ni. V Pythonu se ne zgodi nič takega, razred bi čisto lepo deloval ... dokler ne bi poklicali `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`) kupcem ne omogočajo vračanja izdelkov na police (razred je bil sprogramiran za trgovino, katere lastnik je zagrizen šahist: ko se dotaknete jogurta, ga morate požreti, tako kot morajo šahisti požreti nasprotnikovo 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 „običajnih" objektnih jezikov je takšno delovanje metod čudno. Kako `remove_multiple` ve, katero metodo poklicati? Saj se ta pojavi šele pri naslednikih! Preprosto: `remove_multiple` pokliče `self.remove`, `self`ov `remove`. Ker je `self` tipa `EnhancedBasket`, se pokliče `EnhancedBasket.remove`. Če je `self` samo tipa `Basket`, metode `remove` nima in `remove_multiple` javi napako.

Se zgražamo? Prav. A Python pač tako deluje. JavaScript in še marsikateri drugi skriptni jezik tudi. Ali je nekaj mogoče storiti, se ne preverja ob prevajanju, saj prevajanja ni, temveč šele sproti, med izvajanjem. Ravno tako je funkcija `fibonacci` šele v trenutku, ko je bilo potrebno seštevati, preverila, ali je podane argumente sploh mogoče sešteti.

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))
        super().add(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++, recimo, je to odvisno od tega, ali je `add` definiran kot virtualna metoda ali ne. V Pythonu pa spet kliče natačno to, kar smo naročili, namreč `self.add` – poklicala bo `self`ovo metodo `add` (spet se spomnimo, da so objekti, in `self` z njimi nekakšen imenski prostor; več o tem čez nekaj trenutkov). Če je `self` tipa `EnhancedBasket`, bo poklicala metodo `add` razreda `EnhancedBasket`. Metoda `add` (in vse druge metode v Pythonu) se vedejo, kot bi se, če bi bile deklarirane kot navidezne (*virtual*).

Za klic podedovane metode smo uporabili funkcijo `super`. Ta lahko prejme tudi enega ali dva argumenta, vendar prideta prav bolj ali manj samo pri večkratnem dedovanju, ki se mu bomo (vsaj tule) izogibali. Tudi s tem, kako tale `super` deluje, se, dokler se še učite Python, čim manj ukvarjajte, le frazo si zapomnite. Ali pa tudi ne: v Pythonu se izkaže, da podedovane metode kličemo bolj redko.

Razred kot imenski prostor

Najprej osvežimo spomin na imenski prostore. Imejmo modul, recimo `matematika`, ter v njem funkcijo `fibonacci` in konstanto `pi`. Do njiju pridemo z `matematika.fibonacci` in `matematika.pi`; pravimo, da se `fibonacci` in `pi` nahajata v imenskem prostoru modula `matematika`. Vsak modul določa imenski prostor.

Tudi razred v Pythonu je imenski prostor. V njem je, kar smo našteli znotraj definicije `class`, torej funkcije.

```
>>> Basket.add
<function add at 0x012FB0F0>
```

Da, metode niso nič drugega kot funkcije znotraj razreda. Z njimi lahko delamo enako kot z drugimi funkcijami. Tako kot "običajne" funkcije so tudi metode prvorazredni objekti in jim lahko prirejamo še druga imena.

```
>>> dodaj_v = Basket.add
>>> dodaj_v
<function add at 0x012FB0F0>
```

Običajni ljudje funkcij ne prirejajo drugih imen, temveč jih prvenstveno kličejo. Lahko pokličem `EnhancedBasket.add`, da bi, kot smo počeli prej, dodal v košaro štiri pakete plenic?

```
>>> plenice = Item("plenice", 3.80)
>>> Basket.add(plenice, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() takes exactly 3 argument (2 given)
```

Python ima seveda popolnoma prav: funkcija `add` zahteva – kar poglejte njeno definicijo – tri argumente, imenovali smo jih `self`, `item` in `quantity`. Samo

(zadnja) dva ne bosta dovolj; po logiki jezika zato, ker deklaracija funkcije obljublja tri, po logiki stvari pa zato, ker funkcija, sama zase, gola, vzeta naravnost iz imenskega prostora *razreda*, ne ve, v katero košarico naj pravzaprav dodaja.

Če že hočemo klicati `Basket.add`, moramo dodati še košarico, torej

```
>>> Basket.add(b, plenice, 4)
```

Vendar je to bolj teoretičnega kot praktičnega pomena, saj v praksi pišemo `b.add(plenice, 4)`.

Pred različico 3.0 je Python objektu `Basket.add` rekel *nevezana metoda* (*unbound method*). Po novem pa ga imenuje s pravim imenom: `Basket.add` je čisto navadna *funkcija*. Kaj pa je potem `b.add`?

Tako kot razredi so tudi objekti imenski prostori. V njih se nahajajo, najprej, vsi atributi, ki smo jih priredili objektu, recimo vse, kar je konstruktor priredil `self`u. Tako se v imenskem prostoru `b` nahajata niz `owner` in seznam `content`. Poleg tega pa v imenskem prostoru `b` vidimo tudi vse, kar se nahaja v imenskem prostoru razreda, ki mu `b` pripada, recimo funkcija `add`. Pri dostopanju do funkcij, ki se nahajajo v razredu, pa se zgodi metamorfoza: Python na funkcijo priveže objekt. Funkcija `add` postane metoda, vezana na objekt `b`.

```
>>> b.add
<bound method Basket.add of <__main__.Basket object at
0x1010d3290>>
>>> type(b.add)
<class 'method'>
```

Pri tem je `<__main__.Basket object at 0x1010d3290>` seveda prav naš `b`:

```
>>> b
<__main__.Basket object at 0x1010d3290>
```

Biti vezan pomeni imeti že določen `self`. Ko pokličemo `b.add(plenice, 4)`, smemo (in moramo) prvi argument izpustiti, ker je že določen.

Tudi `b.add` je seveda objekt, prvorazredni objekt. Lahko mu priredimo ime, ga shranimo v seznam, vrnemo kot rezultat.

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

To, kar v, recimo C++ rešujejo s kosovelovsko sintakso `::*`, pride v Pythonu samo po sebi, ker je objekt imenski prostor in ker je metoda objekt.

Mimogrede poškilimo še, kako so narejeni imenski prostori. Čisto tako kot imenski prostori modulov: objekt ima polje `__dict__`, v katerem so shranjeni vsi njegovi atributi. S pisanjem v slovar jih lahko spreminjamo ali dodajamo. `a.__dict__["y"]=42` pomeni isto kot `a.y = 42`; ta, 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
```

Podobno je z imenskim prostorom razreda: tudi razred ima polje `__dict__`, katerega vsebina so metode *razreda* – pa še kaj drugega, kar bomo še videli

```
>>> class A:
...    def f():
...            pass
...
>>> A.__dict__
{'__module__': '__main__', '__doc__': None, 'f': <function f at
0x013C5230>}
```

Ko človek vidi takšno stvar, ga seveda zamika, da bi jo uporabljal. Ne, raje naj polista par razdelkov nazaj, kjer smo našteli funkcije `getattr`, `setattr` in `delattr`. Za vsako normalno rabo zadoščajo, te slovarje pa razumen programer uporablja kvečjemu ob razhroščevanju, če obupa nad orodji, ki mu jih ponuja razvojno okolje.

Razredne spremenljivke

Doslej smo v razred, v njegov imenski prostor, postavljali le funkcije in te so se, če smo do njih prišli skozi imenski prostor objektov, prelevile v metode. Lahko postavimo v imenski prostor razreda še kaj drugega?

Seveda, v imenskem prostoru razreda so imena in imena se lahko nanašajo na objekte poljubnih tipov.

```
class M:
    import os

m = M()
```

`M` je zdaj (precej trapast) razred brez metod, le modul `os` vsebuje. In `m` je enako žalosten izgovor za objekt, saj ne pozna nič drugega kot `m.os`.

Pokažimo pametnejši primer. Vsaki košarici želimo dodeliti identifikacijsko številko. Lahko bi naredili takole.

```
class Basket(object):
    bid = 0

    def __init__(self, owner = ""):
        Basket.bid += 1
        self.id = Basket.bid
        self.owner = owner
        self.content = []
```

Temu, kar predstavlja `bid`, bi v kakem drugem jeziku rekli statična spremenljivka razreda (*static member variable*); tako lahko rečemo tudi v Pythonu, vendar je tu še manj duha ali sluha po kaki "statičnosti". V inicializaciji na začetku, `bid = 0`, smo razred izpustili, saj je vse, kar je zamaknjeno znotraj `class`, v imenskem prostoru razreda. Pravilo je enako kot za imenske prostore modulov: če bi v modulu `fibonacci` napisali `bid = 0`, bi se to od zunaj videlo kot `fibonacci.bid`., znotraj modula pa pišemo le `bid`.

Zakaj pa znotraj metode pišemo `Basket.bid`? Moramo to res početi? Da, čisto tako, kot moramo pisati `self.add`. Oba, `bid` in `add`, sta v imenskem prostoru razreda oziroma objekta.

Če zaženemo primere iz prejšnjih razdelkov 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`. Ker se iz imenskega prostora objekta vidi v imenski prostor razreda, najdemo `bid` tudi pod imenom `self.bid`.[23] Vendar tega ne uporabljamo, saj s pisavo `Basket.bid` poudarimo, da gre za reč, ki pripada vsemu razredu.

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: ker živita v različnih imenskih prostorih, smeta imeti enako ime. Zdaj imamo `Basket.id`, ki predstavlja `id` razreda, torej zadnji podeljeni `id`, in `self.id` ali `b.id`, ki predstavlja `id` posamezne košarice. Imena iz imenskega prostora objekta zasenčijo imena iz imenskega prostora razreda.

Kdor se je v tem razdelku nekoliko izgubil, komur stvari niso povsem jasne, naj se ne vznemirja. Napisano naj jemlje kot indikator, da je za logiko razredov v Pythonu še nekaj globine, ki jo bo razumel, ko bo pametnejši. Vse, kar se mu zdi čudno in nelogično, je del duhovito zamišljenega sistema. Razredi v Pythonu so konceptualno veliko preprostejši od onih v C++, C# in Javi. Do razsvetljenja pa ne vodi razlaga, temveč meditacija in praksa.

Neustrašnim pokažimo še nekaj bolj umazanega. Imenske prostore razredov in objektov je mogoče spreminjati. Lahko, recimo, dodamo kaj v imenski prostor razreda.

```
>>> Basket.zz = 42
```

---

[23] Pri tem `bid` za razliko od metode `add` ne doživi nobenega "vezanja", rezultat ni kak vezan `int`.

Ker objekti vidijo v imenski prostor razreda, so s tem vse košarice – tudi tiste, ki že obstajajo, dobile polje `zz`. Na enak način lahko objektom dodajamo, kar sproti, tudi nove metode, tako da v imenski prostor razreda dodajamo funkcije.

```
>>> Basket.fibo = fibonacci
```

V gornjem primeru bi nas zmotilo samo, da funkcija `fibonacci` nima ravno pravih argumentov za to, da bi bila metoda. Lahko pa si pomagamo celo z lambdo:

```
>>> Basket.vrni42 = lambda self: 42
```

Še huje, metode lahko celo spremenimo ali brišemo.

```
>>> del Basket.add
```

Tako, iz imenskega prostora `Basket` smo pobrisali `add` in, puf, te metode noben objekt nima več.

Lahko pa se spravimo tudi na imenski prostor objektov. Lahko, recimo, povozimo razredno metodo z objektovo.

```
def f(self, 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`, `self.add` (oziroma `b.add`) pa je zdaj `f` in ta 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...
```

Tako se seveda ne programira. Jezik to dopušča le, ker so razredi in objekti narejeni kot imenski prostori. Ni pa vse, kar se da storiti, tudi pametno početi.

Statične metode

Naslov razdelka je zgrešen; statične metode že v prevajanih jezikih ne zaslužijo povsem vzdevka "statične", v Pythonu pa še toliko manj.

Gre, kot bralec dobro ve, za metode, ki so del razreda, vendar jih lahko pokličemo brez objekta. V Pythonu zanje skoraj ne bi bilo potrebno storiti drugega, kot definirati metodo tako, da nima argumenta `self`.

```
class A:
    def f(x):
        return x**2
```

Tako definirana metoda, pravzaprav funkcija, se že obnaša, kot se morajo obnašati statične metode.

```
>>> A.f(14)
196
```

Problem imamo le, če do nje pridemo iz objektovega imenskega prostora. Pri tem jo Python spremeni v metodo, prilepi ji dodatni prvi argument, ki ga metoda ne pričakuje in noče.

```
>>> a = A()
>>> a.f(14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 1 positional argument (2 given)
```

To uredimo tako, da funkcijo dekoriramo, spremenimo v objekt tipa `staticmethod` (dekoratorje smo sicer predstavili, ko smo pisali o funkcijah).

```
class A:
    @staticmethod
    def f(x):
        return x**2
```

Tako okrašena funkcija se ne spremeni v metodo.

Poleg statičnih metod pozna Python tudi razredne metode, ki kot prvi argument ne dobijo objekta temveč njegov razred. Definiramo jih z dekoratorjem

`classmethod`, več o tej nenavadnosti pa lahko bralec izve iz Pythonove dokumentacije.

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 konstruktorju, je predpostavil povsem 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>
```

Obstajata namreč dve metodi za izpis objekta. Ena, `__str__`, pokaže uporabniku prijaznejši izpis, druga, `__repr__` pa bolj tehničnega. Funkcija `print` uporabi prvega, lupine pa ob izpisovanju objektov 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__`.

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 nekoliko kriptično in ga še ne razumemo, v načelu pa pravi, da je Python poskušal iti prek elementov `b`-ja in jih primerjati z nizom `"banane"`, a tega ni mogel storiti preprosto zato, ker pač ni metode, s katero bi lahko šel prek elementov `b`-ja.

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, saj ne moremo neposredno delati s pomnilnikom – ne kazalcev ne `malloc`a ali `new`a nimamo). 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 (čeprav je lepo, da sta; programer ima od razredov pač določena razumna pričakovanja).

`__cmp__(self, other)`

O tej metodi povejmo le, da je ni (več). Če jo boste kje videli, vedite, da ne deluje; Python jo je podpiral (kot zastarelo) do različice 3.0.

`__hash__(self)`

Razpršitvena funkcija, ki jo objekt potrebuje, da je lahko shranjen v slovarjih in množicah. Rezultat mora biti tipa `int`. 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 `__eq__(a, b)` vrnil `True`), tudi `__hash__` vrne enako vrednost.

`__bool__(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 `__bool__` 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`. Zanimivo je prirejanje `b.x.y.z = 3`, ki se izvede kot `b.__getattr__("x").__getattr__("y").__setattr__("z", 3)`. Najbrž ne bi uganili, zdaj ko vidite, pa je jasno, da ne more biti drugače, ne?

`__getitem__(self, key)`, `__setitem__(self, key, value)`, `__delitem__(self, key)`

Podobna reč, a za indeksiranje: te tri metode definirajo operator []. Prva se kliče ob branju: `s[42]` se izvede tako, da Python pokliče `s.__getitem__(42)` in `d["Miran"]` pokliče `d.__getitem__("Miran")`. Druga ob prirejanju: `s[42] = "x"` se izvede kot `s.__setitem__(42, "x")`. Zadnja ob brisanju: `del s[42]` pomeni `s.__delitem__(42)`. Metode z istim imenom se uporabljajo tako za razrede, ki se obnašajo podobno kot seznami, terke in nizi kot za razrede podobne slovarjem. Razlika je le v tem, da prvi zahtevajo argumente tipov `int` in `SliceType` (ta se uporabi ob indeksiranju z rezinami, in vsebuje polja `start`, `step` in `stop`), saj lahko sezname, terke in nize indeksiramo le s števili, drugi pa sprejmejo objekte poljubnih tipov.

`__get__(self, instance, owner)`, `__set__(self, instance, value)`, `__delete__(self, instance)`

Te tri metode določimo razredom, katerih objekte bi radi uporabljali kot deskriptorje za atribute drugih objektov. Če zveni zapleteno: tokrat sem zapletel namerno. Deskriptorji so izjemno praktična reč, če želimo povezovati kodo v C++ s kodo v Pythonu. Seveda nam pridejo prav tudi v čistem Pythonu, vendar je za začetnike boljše, da jih še ne spoznajo, saj jih bodo pripeljali v skušnjavo, da bi v Pythonu programirali na način, na katerega se dela v prevajanih jezikih.

`__iter__(self)`, `__next__(self)`

Prva metoda vrne *iterator* prek objekta. Druga metoda je tisto, kar mora imeti iterator, da lahko iterira. Posvečen jima bo poseben razdelek, za zdaj le nakažimo: metodi sta 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 zanka `for` prek takšnih objektov ne more.

`__contains__(self, obj)`

Metoda, ki se skriva za operatorjem `in`. Kako jo definiramo, smo že videli. Če `__contains__` ne obstaja, si bo Python poskušal pomagati z

`__iter__`, če ni niti tega, bo poskusil z `__getitem__`, v skrajnem primeru pa bo obupal in 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 `a`-jev `__add__` ne zna sešteti `a` in `b`, mora javiti napako `NotImplemented` in Python bo poskusil 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, ki ga priredi objektu na levi strani operatorja `+=` (v gornjem primeru torej `a`-ju).

`__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 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__`

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 zdi večkratno dedovanje tako ali tako slaba in izogibanja vredna praksa, se vanje ne bo spuščal.

Nove razrede lahko izpeljujemo 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 super().__str__(self)

    __repr__ = __str__

True = bool(0)
False = bool(1)
```

Kako? Razred `bool` je enak `int`, samo z drugačnim izpisom za `0` in `1`? Tako je.

```
>>> True+2
3
```

Ker se `int` že od začetka obnaša enako kot bi človek pričakoval od `boola`, je bilo potrebno le popraviti izpis in dodati konstanti `True` in `False`, pa smo dobili tip `bool`. V resnici je razred definiran v Cju, vendar bi tudi gornja rešitev v Python delovala enako. (Z opombo, da se gornjega ne da pognati preprosto zato, ker sta `True` in `False` rezervirani besedi; pred različico 3.0 nista bili in gornja koda bi dejansko delovala.)

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. 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 super().__len__()
```

pa nesmiselno, saj bi tako metoda le poklicala podedovano metodo – čemu jo tedaj sploh definirati. Opustili bi lahko tudi metodi `add` in `add_multiple`, saj se ne razlikujeta bistveno od `append` in `extend`.

Celotna koda razreda je takšna:

```
class Basket(list):
    bid = 0

    def __init__(self, owner = ""):
        Basket.bid += 1
        self.id = Basket.bid
        self.owner = owner
        super().__init__()

    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 žonglira z argumenti.[24]

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']
```

---

[24] Bralec, ki noče spregledati ničesar, 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 slučajno 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. Če se vam zdi zapleteno, pač pozabite: če ne boste programirali preveč čudnih reči, vas tole res ne zanima. Je pa vseeno zanimivo.

```
>>> list((1, 2, 3))
[1, 2, 3]
>>> s = {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.

Razred kot objekt

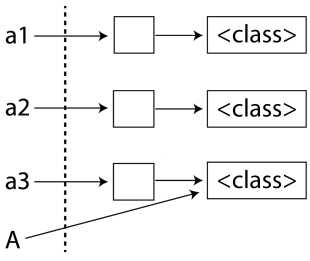
Tako kot definicija funkcije priredi kodo imenu, kot smo izvedeli v dolgo pričakovanem prejšnjem poglavju, tako tudi definicija razreda samo priredi objekt, ki predstavlja razred, imenu razreda. Za ilustracijo si najprej oglejmo tole čudno (in neuporabno) kodo.

```
class A:
    def f(self):
        print(1)

a1 = A()

class A:
    def f(self):
        print(2)

a2 = A()
```



```
class A:
    def f(self):
        print(3)

a3 = A()

a1.f()
a2.f()
a3.f()
```

Program izpiše številke 1, 2 in 3 (kar bi se dalo seveda doseči tudi na manj sofisticiran način). Program naredi namreč to, kar piše: sestavi razred, ki mu da ime `A` in objekt tega razreda, ki mu je ime `a1`. Potem sestavi nov razred, ki mu da ime `A` (po tem se na oni stari razred ne nanaša več ime `A`, še vedno pa je `a1` objekt tega zdaj brezimnega razreda) in objekt tega razreda, `a2`. Vajo ponovi še tretjič. Na koncu imamo razred z imenom `A`, to je tisti, tretji, ter `a1`, `a2` in `a3`, ki so objekti treh različnih razredov, ki so se, vsak ob svojem času, imenovali `A`.

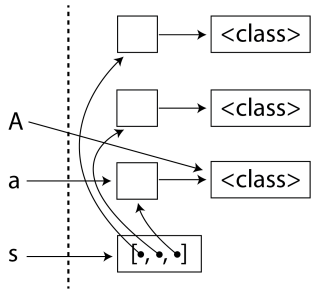
Da bo stvar še perverznejša, storimo to kar v zanki, objekte pa zložimo v seznam.

```
s = []
for i in range(1, 4):
    class A:
        def f(self, i=i):
            print(i)
    a = A()
    s.append(a)

for e in s:
    e.f()
```

(Če koga bega, čemu služi `i=i` in ali ne bi šlo brez tega, naj se spomni, kako smo v zanki `for` definirali štiri funkcije, ki so računale potence. Če ga spomin zapušča, naj se vrne do poglavja o funkcijah.)

Po zanki seznam `s` vsebuje tri objekte, 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 `2`.



Torej: tako kot so spremenljivke v Pythonu za razliko od spremenljivk v Cju samo imena za objekte, tako kot so funkcije in metode v Pythonu prvorazredni objekt, ime funkcije pa je ime kot katerokoli drugo, je tudi gornji `A` samo ime za razred. Razrede lahko sestavljamo in brišemo, lahko jih dajemo v sezname, razred lahko pošljemo kot argument funkciji[25] ali pa ga vrnemo kot rezultat funkcije.

Tako kot je funkcija na nek način čisto navadna spremenljivka, je tudi `A`, 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.

Poglejmo si še nekaj zanimivega o tipih. Funkcijo `type`, ki vrne tip objekta, smo videli že ničkolikokrat. Za preverjanje tipov objektov je raje ne uporabljajmo: izraz `type(a) == 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. Funkcija `issubclass(cls1, cls2)` počne nekaj podobnega: `True` vrne, če je `cls1` posredni ali neposredni potomec `cls2`, ali pa je `cls1` enak `cls2`. Drugi argument je lahko tudi terka razredov. V tem primeru funkcija vrne `True`, če je `cls1` potomec kateregakoli izmed njih.

V zvezi s tem omenimo še dva posebna atributa razreda. Atribut `__name__` vsebuje ime razreda, `__bases__` pa je terka z razredi, iz katerih je razred izpeljan. S plezanjem prek `__bases__` lahko v programu izvemo vse prednike določenega razreda.

---

[25] To smo celo že počeli: kot smo poklicali `defaultdict(list)`, smo dali razred `list` kot argument konstruktorju `defaultdict`.

Kot za vsako rečjo se tudi za funkcijo `type` skriva presenečenje. Namreč: sploh ni funkcija temveč razred. Ko pokličemo `type(f)`, kjer je `f` karkoli že, pokličemo konstruktor. Ta ne vrne novega objekta temveč kar objekt, ki predstavlja razred. Torej: `type(a)` vrne razred `A`. Ni to narobe? Konstruktor `Foo` bi moral vračati objekte razreda `Foo` (ali tipa `Foo`, če hočete; tip in razred je v Pythonu eno in isto). Torej zakaj `type` vrne razred in ne objekta tipa `type`? Saj ga! Razredi so tipa type.

```
>>> type(A)
<class 'type'>
```

Torej `type` ni funkcija, temveč razred, ki mu pripadajo tipi. Ali, to je eno in isto, tip tipov.

Kakšnega tipa pa je tip?

```
>>> type(type)
<type 'type'>
```

Logično, kaj ste pa mislili? Objekt `type` pa je tipa `type`.

Nagradno vprašanje: ima Python "anonimne razrede", tako kot ima lambda-funkcije? Namig: če je `type` tip tipov (razred razredov), bomo morda naredili nov razred tako, da bomo na primeren način poklicali `type`. Poglejte v Pythonovo dokumentacijo. (Kaj pa je nagrada? Zadoščenje ob najdenem odgovoru.)

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 (`s.append`) 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 shrani, da jih naslednjič ne bo 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 stotega, 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
```

Zaporedja

Kot smo se naučili doslej, je Python objektno usmerjen jezik, pri čemer moremo objekte tudi bolj ali manj odmisliti, če pišemo preprost program, ki tega ne zahteva, ali če učimo začetnike, pa jih ne bi radi učili govoriti `class`, še preden znajo reči `for`. Iz nekaterih bolj eksotičnih (že vidim, kako bom tepen!) programskih jezikov, predvsem Hasklla (av, bo bolelo!), pa si je Python sposodil še nekaj trikov čisto drugačnega sloga 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. Izposojenke iz funkcijskega programiranja moramo uporabljati razumno ter sem ter tja premagati skušnjavo in napisati običajno zanko `for` in kak `if` več 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, pa gre zanka `for` vseeno čeznje.

Kako pravzaprav deluje zanka `for`? S katerimi tipi zna delati? Kako, da zna delati s tolikimi? Če definiramo svoj razred, ki vsebuje več elementov, kaj moramo storiti, da ga bomo lahko uporabili v zanki?

Zanka `for` zna prek tistih objektov (razredov), ki imajo metodo `__iter__`. Če je ni, si pomaga z `__getitem__`, a le, če le-ta sprejema celoštevilske indekse. Slednjo smo spoznali v prejšnjem poglavju in o njej nimamo povedati ničesar več. Tule se bomo posvetili metodi `__iter__`.

V resnici `__iter__` ne naredi veliko: `__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. Datoteka 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` v resnici ne pomeni napake, temveč le konec iteracije, zato mora to izjemo ujeti in nehati klicati `__next__`.

Vse to je lažje videti kar na iteratorju v akciji. Na roko poskusimo, kar sicer za nas počne `for`.

```
>>> s = [2, 5, 3]
>>> i = s.__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 skrivnost zanke `for`, ki je zmožna iterirati prek toliko različnih objektov: `for` ne počne ničesar, vse delo opravijo objekti sami.

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 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 biti *nekako takšna*:

```
# 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 "*na tem mestu vračaj rezultate*".

Prava rešitev je zelo podobna tej *nekako-takšni*-rešitvi.

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

Funkcijam, ki vsebujejo `yield`, pravimo generatorji. Ukaz `yield` pa dela točno, kakor smo opisali zaželeno vlogo gornjega `return`: na tem mestu generator vrne rezultat, ko ga "pokličemo" naslednjič, pa se izvaja odtod naprej. Običajno jih uporabljamo 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 generatorjev 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.

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 paziti, da iteracijo ustavimo s primernim `break` v zanki ali kako drugače. Tule je prvo Fibonaccijevo število, ki je deljivo z 1331:

```
for fib in fibonacci():
    if fib % 1331 == 0:
        print(fib)
        break
```

Če ni nobenega (dogledno majhnega) Fibonaccijevega števila, ki bi bilo deljivo z 1331, tega programa raje ne poganjajte.

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 `fibonaccija`, 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. Recimo, da imamo spet onega Fibonaccija, ki mu lahko povemo, koliko členov zaporedja naj generira.

```
>>> g = fibonacci(5)
>>> g
<generator object at 0x012EF148>
```

Posebej veliko nismo izvedeli. Kaj je *generator object*, kaj je mogoče početi z njim?

Pa pomislimo: z njim je mogoče početi to, kar počne zanka `for`? Kaj je to, pa smo se pač ravnokar naučili. Zanka `for` preveri, ali tisto, prek česar smo ji dali teči, 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 je treba.

Prav. Zanka torej dobi `g` in pokliče njegov `__iter__`.

```
>>> g
<generator object at 0x012EF148>
>>> g.__iter__()
<generator object at 0x012EF148>
```

Hec, kar samega sebe je vrnil! Generator je sam svoj iterator! (V resnici gre za pravilo. Vsi iteratorji imajo metodo `__iter__`, ki vrača njih same. V resnici je vsak iterator sam svoj generator, ne obratno.) 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. Generator je hkrati iterator in ima metodo `__next__`, ki (navadno sproti) generira in vrača elemente zaporedja.

Za vsako reč pride njen čas in zdaj je prišel čas za funkcijo, ki jo poznamo že od začetka in se nam (doslej) ni zdela prav nič posebnega: `range`. Ne bi bilo smiselno, da tudi `range` ne bi sestavljal seznama števil, temveč vrnil generator? Natančno to naredi. Nobenega seznama ne vrača. Še huje: ves čas, ko smo živeli v veri, da je `range` funkcija, smo se motili. Sploh ni funkcija, tip je! Ko rečemo `range(2, 5)`, v resnici pokličemo konstruktor, dobimo objekt razreda `range`.

Razred `range` pa ima metodo `__iter__` in vrača iterator, ki gre od 2 do 5! Kako gre to lepo skupaj! Le preverimo, da je vse, kot pravim.

Najprej razčistimo, ali je `range` res tip in ne funkcija.

```
>>> type(range)
<class 'type'>
```

Res. In ko ga pokličemo, dobimo objekt tipa `range`.

```
>>> r = range(2, 5)
>>> r
range(2, 5)
>>> type(r)
<class 'range'>
```

Ta pa zna vrniti iterator in tako naprej...

```
>>> n = r.__iter__()
>>> n.__next__()
2
>>> n.__next__()
3
```

Izpeljane množice

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 podobne operacije in tudi sintaksa je precej podobna.

```
k = {x**2 for x in m if x < 10}
```

Temu rečemo izpeljevanje množic[26] (*set comprehension*).

---

[26]Pri iskanju prevoda je pomagal Andrej Bauer, končne izbire je kriv avtor sam.

Izpeljana množica je, najprej, množica, zato jo zapremo v zavite oklepaje, kakor pač zapisujemo množice v Pythonu. Za razliko od "prave" množice pa njenih elementov ne naštejemo, temveč opišemo. Najprej napišemo izraz, kot je recimo gornji `x**2`; lahko je tudi bolj zapleten, biti pa mora izraz. Sledi zanka `for`, za njo pa pogoj. Slednji ni nujen in ga pogosto ne potrebujemo. Vse je torej čisto tako kot v matematiki, le `for` in `if` zapišemo z besedo namesto s simboli.

Imam množico imen, recimo,

```
imena = {"Ana", "Berta", "Cilka", "Dani", "Ema"}
```

Kako bi izračunal poprečno dolžino imena? Imam funkcijo `sum`, ki zna sešteti števila v seznamu, množici ali čemerkoli že. Potrebujem torej množico dolžin, namesto množice imen. To pač preprosto dobim: izračunati želim *dolžino niza ime za vsako `ime` v množici `imena`*.

```
>>> {len(ime) for ime in imena}
{3, 5, 5, 4, 3}
```

Zdaj pa tole le še seštejem in delim z velikostjo množice.

```
>>> sum({len(ime) for ime in imena}) / len(imena)
4.0
```

Znamo poiskati množico deliteljev danega števila `n` (brez njega samega)? Delitelji števila `n` so vsa tista števila od `1` do `n`, pri katerih je `x % n` enak `0`.

```
>>> n = 60
>>> {x for x in range(1, n) if n % x == 0}
{1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30}
```

Zdaj lahko hitro napišemo funkciji, ki povesta ali je dano število praštevilo (torej: nima deliteljev) in ali je popolno (torej: enako vsoti svojih deliteljev).

```
def popolno(n):
    return n == sum({x for x in range(1, n) if n % x == 0})

def prastevilo(n):
    return not {x for x in range(2, n) if n % x == 0}
```

Če imamo funkcijo `prastevilo`, hitro pridemo do vseh praštevil do 50.

```
>>> {n for n in range(2, 51) if prastevilo(n)}
{2, 3, 5, 7, 41, 11, 13, 47, 17, 37, 19, 43, 23, 29, 31}
```

Tudi če nimamo funkcije `prastevilo`, hitro pridemo do vseh praštevil do 50.

```
>>> {n for n in range(2, 50) if not {x for x in range(2, n) if
n % x == 0}}
{2, 3, 5, 7, 41, 11, 13, 47, 17, 37, 19, 43, 23, 29, 31}
```

Izpeljevanje seznamov in slovarjev

Na podoben način kot izpeljemo množico, če med zavite oklepaje napišemo izraz, `for` in `if`, bomo dobili seznam, če zavite oklepaje zamenjamo z oglatimi. Še več, izpeljane sezname pozna Python že od različice 2.2, izpeljane množice pa šele od 2.7.

Vse, kar smo napisali v prejšnjem razdelku bi torej delovalo tudi, če bi vse zavite oklepaje zamenjali z oglatimi.[27] Pravzaprav bi delovalo še boljše: delitelje, recimo, je veliko bolj ekonomično shranjevati v seznamih kot v množicah. Izpeljani seznami so obstajali prej kot izpeljane množice že zato, ker so uporabnejši.

Za šalo naredimo kaj bolj zapletenega. Imam seznam števil.

```
k = [9, 16, 81, 25, 196]
```

In zdaj – v nasprotju s pričakovanji vseh budnih bralcev z minimalnim znanjem matematike – ne bomo računali njihovih korenov, temveč jih bomo obrnili, dobiti bi želeli seznam `[9, 61, 18, 52, 691]`.

```
>>> [int(str(x)[::-1]) for x in k]
[9, 61, 18, 52, 691]
```

Vsak `x` s seznama `k` (`for x in k`) spremenimo v niz (`str(x)`), ga obrnemo (`str(x)[::-1]`) in spremenimo nazaj v število (`int(str(x)[::-1])`).

---

[27] Še huje: razdelek o izpeljanih množicah je nastal tako, da sem primerom iz prejšnje izdaje knjige zakrivil vse oglate oklepaje.

Ostali so nam še izpeljani slovarji. Tudi njihove sintakse ni težko uganiti: uporabimo zavite oklepaje, saj slovarje pišemo v zavite oklepaje, od množic pa se ločijo po tem, da so, njegovi elementi so pari ključ in vrednost, med katere postavimo dvopičje.

Če hočemo narediti slovar, ki bo kot ključe vseboval števila do 10, kot vrednosti pa njihove kvadrate, napišemo

```
{i: i**2 for i in range(11)}
```

Imam datoteko *studenti.tab*, ki ima dva stolpca, ločena s tabulatorjem. V prvem je id študenta, v drugem njegovo ime. Kako bi jih prebral v slovar?

```
{vrsta.strip().split("\t") for vrsta in open("student.tab")}
```

Generatorski izrazi

Ob branju o Pythonu smo se že navadili, da se za vsako stvarjo skriva še nekaj globjega; od nedolžnega `a, b = b, a+b`, za katerim je bilo razpakiranje terk, do razredov, za katere se je pokazalo, da so le imenski prostori in istočasno, z druge strani, objekti, ki jih je mogoče poklicati.

Izpeljani seznami, izpeljane množice in izpeljani slovarji imajo navidez nekaj skupnega, namreč nekakšen vdelan `for` z `if`-om. Da nimajo nekaj skupnega tudi v resnici, ne le navidez? Ima tisti izraz s `for` in `if` pomen tudi sam zase?

Poglejmo.

```
>>> g = (x**2 for x in fibonacci(1000))
>>> g
<generator object at 0x012EF198>
```

Okrogli oklepaji tokrat ne pomenijo terke, temveč le zapirajo izraz, tako kot v, recimo 2 * (3 + 7). Kar je v oklepaju, pa je, vidimo, generator. Gornje je torej tako, kot če bi rekli

```
def kvadrati():
    for x in fibonacci(1000):
        yield x**2
g = kvadrati()
```

Le krajše je. Še raje si gornje predstavljamo kot izpeljan seznam, kot `[x**2 for x in fibonacci(1000)]`, le da generira vsak element posebej.

Tak generator lahko uporabimo povsod, kjer pač 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']
```

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))
3057018985272083280891769550770709134443022741973377352285622728
8485339924629663085880556303646279364980699180369468530938703716
6323200343747296781992700715211522505284983929452546454524440852
7529080083391004460907758759027766765632329201613696317437463342
7878779118279769686493565541215746678338047696780661403031496224
4622315773287342597409177188867302433121412464029059694406425161
74569611589546099247653691323252913751L
```

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(s):
    s = 0
    for i in s:
        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)
189019809319230383820134057488591865163525386228509400140659297
627472136323219841436151264279356409048960454666824587851373642
445046068389159309136350677823570009463474009368119843875985834
565802281360515575558258547860015140843773019098272579044041834
651814918905623794716924537961011262056107442702809888261656895
650292844213929065890812009554234551485104892451758095150332809
226440001050585004424161370198572540850OL
```

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 in nemara jo celo pogosteje kličemo z generatorjem kot z že pripravljenim seznamom.

```
>>> ", ".join(str(x**2) for x in fibonacci(10))
'1, 1, 4, 9, 25, 64, 169, 441, 1156, 3025'
```

Poprečne dolžine imen v množici `imena` nam ni potrebno računati z izpeljano množico,

```
sum({len(ime) for ime in imena}) / len(imena)
```

saj lahko `sum` pokličemo kar z generatorjem

```
sum(len(ime) for ime in imena) / len(imena)
```

Generatorje lahko, če sem spreten (in rad pišem nerazumljivo kodo), celo nizam. Kako iz datoteke prebrati vse neprazne vrstice in odstraniti beli prostor? Če želimo izpeljani seznam, naredimo takole:

```
[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 spet ni všeč, saj dvakrat pokličemo `strip`. Tega se znebimo z dodatnim generatorjem (oklepaji okrog generatorja so izpisani malo večje, da ne preveč čitljivi izraz lažje preberemo).

```
[line for line in (s.strip() for s in file("besedilo.txt")) if
line]
```

Generator vrne "olupljene vrstice", pri izpeljevanju seznama pa izberemo le neprazne. Če to še ni končna postaja, lahko vse skupaj spremenimo v nov generator, tako da zamenjamo zunanje oglate oklepaje z okroglimi.

```
(line for line in (s.strip() for s in file("besedilo.txt")) if
line)
```

Priročne funkcije

Poleg naštetih reči, ki so vdelane v sintakso jezika, ima Python še nekaj funkcij, ki jih je smiselno omeniti v kontekstu tega poglavja.

Še preden je dobil izpeljane sezname, je imel Python funkcije `map`, `filter` in `reduce`. Prvi dve ima še vedno, vendar sta iz mode, ker ju ne potrebujemo več, tretja pa se je morala umakniti v modul `functools`. Ker še vedno kdaj pridejo prav, jih vseeno opišimo.

Prva, `map`, kot argument dobi funkcijo in enega ali več seznamov. Rezultat je seznam, v katerem so vsi elementi podanega seznama, preslikani prek podane funkcije. Če je `imena` seznam nizov, bo `map(len, imena)` vrnil seznam imen nizov, natanko tako, kot bi ga dobili z `[len(x) for x in imena]`. Če `map`u podamo več seznamov, mora funkcija sprejemati več argumentov: `map` bo vzporedno jemal elemente iz seznamov, klical funkcijo in shranjeval rezultat v novi seznam, dokler najkrajšemu od seznamov ne zmanjka elementov. Namesto seznamov lahko podamo tudi karkoli, prek česar lahko spustimo zanko `for`, vključno z generatorjem.

Funkciji `filter` podamo funkcijo in seznam. Sestavila bo nov seznam, ki bo vseboval le tiste elemente originalnega seznama, za katere funkcija vrne resnično vrednost. Tako kot `map` lahko tudi `filter` nadomestimo z izpeljanim seznamom: `filter(f, s)` vrne enak rezultat kot `[x for x in s if f(x)]`.

Funkcijo `reduce` je najpreprosteje opisati kar, če jo napišemo.

```
def reduce(f, s, r):
    for e in s:
        r = f(e, r)
    return r
```

Le malo smo jo poenostavili: zadnji argument smemo tudi izpustiti. Če bi, recimo, želeli izračunati vsoto vseh elementov seznama in ne bi vedeli za funkcijo `sum`, bi lahko napisali

```
from functools import reduce
reduce(lambda x, y: x+y, s)
```

ali

```
from operator import add
reduce(add, s)
```

Funkcija `all(s)` vrne `True`, če so vsi elementi seznama `s` resnični, `any(s)` pa, če je resničen vsaj en element seznama. (Bi znali to zapisati s pomočjo `reduce`?[28])

Kakor prej lahko namesto seznama podamo tudi generator. So vsa praštevila med 50 in 100 liha?

```
>>> all(p % 2 != 0 for p in range(50, 101) if prastevilo(p))
True
```

Funkciji `max(s)` in `min(s)` vrneta najmanjši in največji element seznama oz. elementa, ki ga vrača nek generator.

[28] `reduce(lambda x, y: x and y, s)`, vendar to ni učinkovito, saj vedno pregleda vse elemente seznama, namesto da bi se ustavil ob prvem neresničnem.

Funkciji `sorted(s)` in `reversed(s)` kot argument dobita seznam ali, v splošnem, generator, vrneta pa urejen seznam oz. seznam z obratnim vrstnim redom argumentov.

Kogar reči v tem poglavju zabavajo, naj si vsekakor ogleda, kaj je še zanimivega v modulu `functools` in, predvsem, `itertools`.

Generatorji v Pythonu pred različico 3.0

Python je dobil generatorje v različici 2.2. Funkcije, kot so `range`, `filter` in `map` ter metode, kot so `keys`, `values` in `items`, so predtem vračale sezname. Po uvedbi generatorjev bi bilo sicer smiselno vse te funkcije spremeniti tako, da bi vračale iteratorje, vendar tega niso storili zaradi ohranjanja združljivosti. Z ostanki preteklosti je pospravila različica 3.0. Če boste morali kdaj delati s starejšimi različicami – kar bo še nekaj let kar verjetno – pa se zavedajte, da se mnoge funkcije, od katerih ste vajeni dobivati iteratorje, tam obnašajo drugače.

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 bo `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. 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?

7. Napiši funkcijo, ki izračuna fakulteto *n*.

Rešitve

1. Trivialno.

```
>>> import set, os
>>> {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. Dobili bi tole:

```
>>> 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. 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*.

```
>>> list(zip(*[k[0] for k in kvadrati]))
[(1, 3, 1), (2, 1, 4)]
```

V resnici ni potrebno poklicati še `list`; to smo storili, ker je zip generator, tule pa želimo videti, kaj generira. Zdaj pa izračunajmo maksimum vsakega od seznamov.

```
>>> [max(x) for x in zip(*[k[0] for k in kvadrati])]
[3, 4]
```

Kako pa nalogo rešimo za poljubno število dimenzij? Smo je že: naša rešitev je čisto splošna, saj je nismo nikjer omejili na dve koordinati.

7. To nalogo smo že reševali, čisto na začetku, vendar z zanko `for`. Zdaj jo znamo rešiti preprosteje.

```
def fakulteta(n):
    from operator import mul
    from functools import reduce
    return reduce(mul, range(2, n+1), 1)
```

Nekateri zanimivejši moduli

Python dobimo 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 je potrebno poiskati in namestiti posebej. 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.

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; ker gre za splošnonamenski jezik, so v ločenem modulu `re`. Sintakse regularnih izrazov ne bomo opisovali, saj je taka kot drugod.

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 za vselej. 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.

Pred niz z regularnim izrazom, torej pred narekovaj, se nam splača dodati `r`. Z njim 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 namenjeni modula se imenuje `pickle`.

Funkcija `pickle.dump` kot argument pričakuje objekt, ki naj ga shrani, in odprto datoteko. V isto datoteko lahko seveda zapišemo poljubno število objektov. Beremo jih s `pickle.load`, ki dobi le en argument, namreč datoteko. Namesto datoteke lahko uporabljamo tudi kak drug objekt, ki se obnaša podobno kot datoteka: `dump` lahko piše v poljuben objekt, ki ima metodo `write`, ki sprejme bajte, `load` pa bere iz objektov, ki imajo metodo `read`.

Če objektov ne kanimo shranjevati v datoteke temveč v podatkovno bazo ali pa jih pošiljati prek interneta, namesto `pickle.dump` uporabimo `pickle.dumps`, ki mu damo le objekt in ga vrne zapisanega v niz (točneje, `bytes`). Iz bajtov nazaj v objekt pridemo s `pickle.loads`. Funkciji sta dovolj pametni, da znata shranjevati in brati tudi poljubno gnezdene strukture in celo ciklične strukture.

Za resnejše zapisovanje in branje je pametneje uporabiti razreda `pickle.Pickler` in `pickle.Unpickler`. Obema že ob konstruiranju podamo datoteko (ali datoteki

podoben objekt), v katerega pišeta in iz katerega bereta, nato pa le še kličemo `dump` ali `load`, ne da bi ponovno podajali datoteko. Njuna prednost pred golima funkcijama je v tem, da si zapomnita objekte, ki so že zapisani oz. že prebrani in jih ne zapisujeta in ne bereta ponovno, temveč se le skličeta na to, kar je že znano.

Naključje, kriptografija, razprševanje

Modul `random`, 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 uporabljajo skupni, 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.

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`), uporabljati telnet (`telnetlib`).

Gre pa tudi v drugo smer: običajna distribucija že vsebuje različno zmogljive preproste strežnike za http (`http.server`) 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, za zaresno delo najbrž Django.

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.

Introspekcija

Da Python vidi vase, smo se prepričali sproti. Videli smo, recimo, da so metode razreda shranjene kar v slovarju, ki nam je dostopen tudi iz programa. Tule si oglejmo le še tolmačev sklad, do katerega 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, s=0):
    a = random.randint(0, 10)
    if s>5 or s>=2 and a>8:
        return f(x)
    elif a<3:
        h(x)
    g(x, s+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, 's': 4}

C:\D\x.py:14, funkcija g
Koda: return f(x)
Lokalne spr: {'a': 9, 'x': 12, 's': 3}
```

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, velik, neuradni uradni repozitorij je na strani http://pypi.python.org/.

Za prijetnejše delo z moduli se splača uporabljati upravljalnik paketov, *pip*. Z njim delamo tako kot z upravljalniki, ki jih bralec (upam) pozna iz sistemov, ki temeljijo na Unixu. Tako, recimo, `pip install numpy` poišče in namesti modul numpy, `pip uninstall` pa se ga znebi.

Če si nočemo s preskušanjem paketov zapackati direktorija `site-packages`, kamor se shranjujejo Pythonovi moduli, ali pa hočemo imeti za različna opravila različne vrtičke, različne konfiguracije knjižnic, se nam splača dodati še virtualenv. Z njim lahko ustvarjamo nove direktorije, ki zamenjajo (oz. dopolnijo) site-packages in vanje nameščamo različne module za različne konfiguracije Pythona.

Matematične in znanstvene knjižnice

Modul `numpy` služi delu z matrikami in linearni algebri. Modul je tako razširjen, da so za enega njegovih prednikov predlagali, naj se uvrsti kar med standardne, vdelane module, vendar je za to prevelik in prespecifičen, pa tudi organizacijsko ga je bolj smiselno voditi kot ločen projekt.

Numpyjeve matrike lahko vsebujejo najrazličnejše tipe, od booleovih in enobajtnih do `long int` in `double`, in vsako zasede le toliko, kolikor je nujno potrebno (za razliko od Pythonovih seznamov, ki so zelo požrešna reč). 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.

Celotna knjižnica temelji na LAPACK, prastarih rutinah, napisanih v Fortranu, saj gre za najboljšo robo, kar je obstaja na tem področju. Python z numpyjem je čisto spodobna zamenjava za Matlab.

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...

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).

Kogar zanimajo te reči, naj si ogleda še Sage. Sage je paket, ki je požrl kup odprtokodnih paketov, povezanih z matematiko, in se trudi biti alternativa Mathematici, Matlabu in Mapleu.

Grafika, video, zvok

Knjižnica PIL (Python Imaging Library) sicer ni tako vseprisotna kot numpy, vendar si je pridobila mesto standardne knjižnice za obdelavo slik.[29] 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 zna delati tudi z odložiščem.

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. S PyGame je mogoče delati čisto resne reči, saj temelji na SDL, čisto zaresni v Cju napisani knjižnici za izdelavo iger.

---

[29] In ga izgublja, ker v času pisanja te knjige uradno še vedno podpira šele Python 2.7.

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. Druga, najbrž boljša knjižnica je 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 napisan v C++, vendar ima vmesnike do vseh mogočih jezikov; naučite se ga uporavljati v Pythonu, pa ga boste znali tudi drugod. Deluje tudi na vseh mogočih platformah, od Windowsov in Os X do prezgodaj preminulega Maemota. In, kar je najpomembnejše, na vsaki platformi je videti tako, kot mora: z njim 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 PyWin32 za MS Windows. PyWin32 je bolj znan pod drugim imenom, imenom razvojnega okolja, ki je povezano z njim: PythonWin. Zaradi tega, kaj zna početi z MS Windows, je PyWin32 vreden posebne obravnave.

PyWin32

PyWin32 bi lahko uporabljali kot okolje za sestavljanje uporabniških vmesnikov. A tega navadno ne počnemo – druga okolja so veliko prijetnejša in splošnejša. Zato pa PyWin32 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.

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.[30]

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.

---

[30] 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.

    - 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 nadaljnje besede začenjamo z veliko začetnico, ne pa oboje; zaželeno je prvo.

    - 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 pustijo 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.

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. 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 ali Os Xov bundle, ki pa v resnici izvaja program, ki smo ga delno ali v celoti napisali v Pythonu. Kdor potrebuje kaj takšnega, naj si ogleda *py2exe* ali *cx_freeze*.

Pa tudi vse, kar smo povedali, smo skoraj brez izjeme povedali površno. Najboljše bralca šele čaka.