

Generatorji

V oglate oklepaje zapiramo (izpeljane ali naštevne) sezname, v zavite zapiramo množice in slovarje. V okrogle pa ... terke? Lahko naredimo "izpeljano terko"? Ne, tega ne bi potrebovali velikokrat. Pač pa nas čaka - in bo pogosto v okroglih oklepajih - še nekaj bolj nenavadnega in imenitnejšega: generatorji.

```
g = (x ** 2 for x in range(4))
```

Tale `g`, kot se hitro prepričamo, ni terka, temveč nekaj drugega.

```
g
```

```
<generator object <genexpr> at 0x1067263b0>
```

Kaj je generator, čemu služi in kako ga uporabljamo? Na prvi dve vprašanji lahko odgovorimo naenkrat: generator je nekaj, kar generira objekte. Vsakič, ko bomo od njega zahtevali nov objekt, bo vrnil novo število - najprej 0, potem 1, potem 4, potem 9. pač kvadrate naravnih števil od 0 do (vključno) 3.

Kako pa "zahtevamo" nov objekt. Tega ne počnemo s funkcijo `next`.

```
next(g)
```

```
0
```

```
next(g)
```

```
1
```

```
next(g)
```

```
4
```

```
next(g)
```

```
9
```

```
next(g)
```

```
-----  
StopIteration                                Traceback (most recent call last)  
/var/folders/f9/lgq5ysmn24j3lkss6zkgmtr00000gn/T/ipykernel_96026/4253931490.py in <module>  
----> 1 next(g)
```

StopIteration:

Na koncu, ko smo zahtevali že peti objekt - `g` pa je zmožen generirati le štiri, se je pač pritožil, javil napako.

Sem se zgoraj zatipkal, ko sem napisal, da generatorjev ne uporabljamo s funkcijo `next`, potem pa točno to počel? Ne, nisem se zatipkal. V resnici ne bomo. Funkcijo `next` sem jo pokazal samo zato, da bi razumeli, kaj počnejo generatorji. V praksi pa klicanje `next`-a prepustimo `for`-u.

```
g = (x ** 2 for x in range(4))
```

```
for i in g:  
    print(i)
```

```
0  
1  
4  
9
```

Če to poskusimo ponoviti, ne gre: generator je že "iztrošen". Kar je imel zgenerirati, je zgeneriral.

```
for i in g:  
    print(i)
```

Lahko zavrtimo generator nazaj? Obstaja `prev`? Ali `rewind`, `restart`. Ne, to bi bila prehuda zahteva, kot bomo videli v nadaljevanju, ko bomo spoznavali, kako so narejeni in česa vsega so v resnici zmožni. Pa tudi uporabno in smiselno ne bi bilo.

Generatorji kot argumenti

Generator je torej videti kot izpeljani seznam, le da namesto oglatih oklepajev uporabimo okrogle. Kadar ga pošljemo kot edini argument funkciji, smemo oklepaje celo izpustiti. Napišimo funkcijo, ki kot argument dobi seznam števil in vrne prvo število, ki je večje od 50.

```
def nad50(s):  
    for e in s:  
        if e > 50:  
            return e
```

Preskusimo jo, trikrat:

```
nad50([5, 1, 40, 1, 82, 12, 6])
```

```
82
```

```
nad50([x ** 2 for x in range(10)])
```

```
64
```

```
# Dodatni presledki so neumni in so tu samo zaradi pregledanosti primera  
nad50(    (x ** 2 for x in range(10))    )
```

```
64
```

V prvem primeru je dobila najobičajnejši seznam. Tako smo si predstavljali, ko smo jo pisali. V drugem primeru smo ji dali seznam kvadratov števil do 10 in vrnila je prvi kvadrat, ki je večji od 50. V tretjem klicu pa ji sploh nismo dali seznama, temveč generator, ki je dajal kvadrate - natančno isto reč kot gornji `g`.

Funkcija je čez generator pognala zanko `for` natančno tako, kot jo sicer poganja čez sezname.

V čem je prednost generatorjev pred seznamami? V tem, da ne sestavijo seznama. Če želimo izračunati vsoto kvadratov prvih stotih števil in za to uporabimo generator, na ta način ne sestavimo seznama teh števil (kot bi ga, če bi rekli `sum([x**2 for x in range(100)])`), temveč števila, namreč kvadrate, generiramo sproti (tako, da pokličemo `sum((x**2 for x in range(100)))`). Hm, pa to v resnici deluje? No, seveda. Funkcija `sum` bi lahko bila napisana takole

```
def sum(s):
    vsota = 0
    for e in s:
        vsota += e
    return vsota
```

Zanka `for` lahko gre prek generatorja, torej ona reč deluje.

Že, že, poreče pozornejši študent: kaj pa `range(100)`? Mar ta ne sestavi seznama stotih števil? Smo res kaj dosti pridobili - namesto seznama kvadratov števil do 100 imamo pač števila do 100 - je to res tak napredek? Tu lahko študenta pohvalimo za budnost, vendar se moti.

```
range(100)
```

```
range(0, 100)
```

V resnici `range` ne sestavi seznama, temveč generator. Ampak temu se bomo posvetili malo kasneje, ko bomo o generatorjih vedeli še malo več.

Tu si raje oglejmo še en lušten detalj. Spomnimo se, da okrog elementov terke ni potrebno pisati oklepajev; to smo uporabili, recimo, pri vračanju rezultatov z `return x, y` ali pri menjavi vrednosti spremenljivk `a, b = b, a`. Podobno tudi okrog generatorjev ni potrebno pisati oklepajev, kadar so edini argument funkcije. Zadnji klic funkcije `nad_50`, ko smo podali generator, je imel precej oklepajev - da bi se znašli med njimi, smo dodali celo nepotrebne presledke.

```
nad50( (x ** 2 for x in range(10)) )
```

Ker je tale generator edini argument funkcije, smemo oklepaje izpustiti in pisati kar

```
nad50(x ** 2 for x in range(10))
```

```
64
```

Vsoto kvadratov števil od 0 do 9 lahko izračunamo tako.

```
sum(x ** 2 for x in range(10))
```

```
285
```

Tako bomo počeli poslej.

Funkcije, ki generirajo

Še nekoliko naprednejša snov: Bi znali napisati generator Fibonaccijevih števil, tako kot smo napisali generator kvadratov? Da, vendar bo preprostejša nekoliko drugačna pot. Napisali bomo funkcijo, ki ne vrača le enega rezultata temveč "generira" rezultate. Torej, nekaj takšnega:

```
def fibonacci(n):  
    # Tole ne deluje!  
    a = b = 1  
    for i in range(n):  
        return a  
        a, b = b, a+b
```

Tale funkcija ne deluje, napisali smo jo le, da ilustriramo idejo: radi bi naredili zanko. Ko funkcijo prvič pokličemo, bi **return** vrnil prvo Fibonaccijevo število. Ko jo pokličemo naslednjič, se funkcija ne bi izvajala od začetka, temveč od tam, kjer smo nazadnje vrnili rezultat, se pravi z vrstico, ki sledi **returnu**. No, v resnici naredimo natanko tako, le namesto **returna** moramo uporabiti **yield**:

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

Preskusimo.

```
f = fibonacci(10)  
f
```

```
<generator object fibonacci at 0x1068bf840>
```

Rezultat je torej - generator! Generatorje lahko napišemo v takšni obliki, kot smo jih spoznali sprva, lahko pa v obliki "funkcije". Mehanika je v resnici takšna, da klic te funkcije vrne generator - kot smo poklicali **fibonacci(10)**, smo kot rezultat dobili generator.

```
next(f)
```

```
1
```

```
next(f)
```

```
1
```

```
next(f)
```

```
2
```

```
next(f)
```

```
3
```

```
next(f)
```

```
5
```

Kako to deluje, bomo najlažje videli, če v funkcijo nasujemo nekaj `print`-ov.

```
def fibonacci(n):  
    print("Spet bo treba računati")  
    a = b = 1  
    for i in range(n):  
        print("Zdaj bom vrnil člen", a)  
        yield a  
        print("In potem bomo nadaljevali")  
        a, b = b, a + b
```

Pokličimo funkcijo, shranimo generator.

```
f = fibonacci(10)
```

Se je kaj izpisalo? Nič!!! "Funkcija" se sploh še ni začela izvajati! Funkcija steče vsakič, ko pokličemo `next` in "zamrzne" ob `yield`-u.

```
next(f)
```

```
Spet bo treba računati
```

```
Zdaj bom vrnil člen 1
```

```
1
```

Poglejte `print`-e in izpis, pa boste točno videli, kaj se je izvedlo.

In potem glejte, kako teče funkcija naprej vsakič, ko pokličemo `next`.

```
next(f)
```

```
In potem bomo nadaljevali
```

```
Zdaj bom vrnil člen 1
```

```
1
```

```
next(f)
```

```
In potem bomo nadaljevali
```

```
Zdaj bom vrnil člen 2
```

```
2
```

```
next(f)
```

```
In potem bomo nadaljevali
```

```
Zdaj bom vrnil člen 3
```

```
3
```

Funkcija vsakič nadaljuje od tam, kjer se je prejšnjič ustavila, se pravi od `yield`-a.

Seveda tudi tega generatorja ne bomo klicali z `next`, temveč s `for`.

```
for x in fibonacci(10):  
    print(x)
```

Napišemo lahko celo funkcijo, ki vrne (no, generira) **vsa** Fibonaccijeva števila.

```
def fibonacci():  
    a = b = 1  
    while True:  
        yield a  
        a, b = b, a+b
```

Neskončno zanko, `while True`, smo že videli, vendar je bil v njej vedno `break`, ki jo je nekoč prekinil. Kdo pa prekine to zanko? Če nismo previdni, nihče.

```
for i in fibonacci():  
    print(i)
```

se vidimo onstran večnosti. Pač pa lahko poiščemo, recimo, prvo Fibonaccijevo število, ki je večje od 50.

```
for i in fibonacci():  
    if i > 50:  
        print(i)  
        break
```

55

Ah, saj imamo že funkcijo za to reč, `nad50`. Naj kar ta pove, katero je prvo Fibonaccijevo število večje od 50!

```
nad50(fibonacci())
```

55

Primer: Generator deliteljev

Še en zanimiv primer je generator, ki vrne vse delitelje podanega števila.

```
def delitelji(n):  
    for i in range(1, n + 1):  
        if n % i == 0:  
            yield i
```

```
list(delitelji(42))
```

```
[1, 2, 3, 6, 7, 14, 21, 42]
```

Opazimo lahko, da z enim deliteljem dobimo dva: če je `i` delitelj `n`-ja, je tudi `n // i` delitelj `n`-ja. Če je tako, zadošča, da gremo do korena iz `n` in vrnemo po dva delitelja.

```

from math import sqrt

def delitelji(n):
    for i in range(1, int(sqrt(n) + 1)):
        if n % i == 0:
            yield i
            yield n // i

list(delitelji(24))

[1, 24, 2, 12, 3, 8, 4, 6]

```

Koren iz `n` moramo spremeniti v celo število, ker `range` ne mara necelih.

Pazite, tole sta dva `yield`a. Funkcija se izvaja tako, da vrne najprej eno število, in ko zahtevamo naslednje, se izvede naslednji `yield`.

Funkcija je zdaj bistveno hitrejša (pri velikih številih bi se to utegnilo kar poznati - namesto, da gre do milijona, bo šla le do 1000. Vendar žal ne dela pravilno. Če je `n`, recimo, 25, bo funkcija dvakrat vrnila 5. A tega se znamo hitro znebiti.

```

def delitelji(n):
    for i in range(1, int(sqrt(n) + 1)):
        if n % i == 0:
            yield i
            if i ** 2 != n:
                yield n // i

```

Zdaj nas morda moti le še to, da števila ne prihajajo v pravem vrstnem redu. Kot delitelje 42 bi namesto 1, 2, 3, 6, 7, 14, 21, 42 dobili 1, 42, 2, 21, 3, 14, 6, 7. To lahko popravimo tako, da `n // i` ne vračamo sproti, temveč jih le shranjujemo in jih vračamo kasneje.

```

def delitelji(n):
    ostali = []
    for i in range(1, int(sqrt(n) + 1)):
        if n % i == 0:
            yield i
            if i ** 2 != n:
                ostali.append(n // i)
    for e in ostali:
        yield e

```

Nismo še čisto zmagali. Zdaj imamo 1, 2, 3, 6, 41, 21, 14, 7 - prva polovica je iz prvega `yield`a, druga (od 41 naprej) iz drugega. Te, druge, vrača v enakem vrstnem redu, v katerem jih je vstavljajal v seznam, saj `append` pač vstavlja na konec.

Pa vstavljajmo raje na začetek! Uporabimo `insert`.

```
def delitelji(n):
    ostali = []
    for i in range(1, int(sqrt(n) + 1)):
        if n % i == 0:
            yield i
            if i ** 2 != n:
                ostali.insert(0, n // i)
    for e in ostali:
        yield e
```

Žal se je `insert`u modro izogibati. Za to, da vstavi element na prvo mesto, mora enega za drugim premakniti vse ostale. V teoriji (ki se jo boste učili drugo leto) je ta program enako počasen kot bi bil, če bi prvo zanko spustili prek `range(1, n + 1)`. S tem, ko smo zamenjali `append` z `insert`, smo, vsaj v teoriji, zapravili ves prihranek.

To je preprosto urediti. Vstavljali bomo na konec, z `append`. Pač pa bomo seznam nato prehodili v obratnem vrstnem redu. To se da narediti z indeksiranjem (`for i in range(-1, -n - n, -1): yield ostali[i]`), vendar si bomo raje pomagali s priročno funkcijo `reversed`, ki obrne seznam.

```
def delitelji(n):
    ostali = []
    for i in range(1, int(sqrt(n) + 1)):
        if n % i == 0:
            yield i
            if i ** 2 != n:
                ostali.append(n // i)
    for e in reversed(ostali):
        yield e
```

Če nam je pomnilnika žal bolj kot časa, pa lahko naredimo drugače: gremo do korena in vračamo delitelji, nato pa od korena nazaj dol in vračamo njihove pare. Z drugimi besedami: to kar v gornjem programu shranjujemo, v spodnjem ponovno zgeneriramo.

```
from math import sqrt, ceil
```

```
def delitelji(n):
    ostali = []
    for i in range(1, int(ceil(sqrt(n)))):
        if n % i == 0:
            yield i
    for i in range(int(sqrt(n)), 0, -1):
        if n % i == 0:
            yield n // i
```

Prepričajmo se, da deluje tudi v zoprnih robnih primerih.


```
list(delitelji(42))
[1, 2, 3, 6, 7, 14, 21, 42]
list(delitelji(25))
[1, 5, 25]
list(delitelji(4))
[1, 2, 4]
list(delitelji(3))
[1, 3]
list(delitelji(2))
[1, 2]
list(delitelji(1))
[1]
```

Iteratorji

Generatorji so samo posebna zvrst iteratorjev. Kaj pa so iteratorji?

Uvedimo jih tako, da najprej posplošimo nekaj, kar že poznamo.

Ob slovarjih smo izvedeli, da kot ključ ne moremo uporabiti ravno poljubnega objekta, temveč le objekte, ki so nespremenljivi *immutable*. To je samo približno res. Uporabiti smemo objekte, ki so *hashable*. Objekti morajo imeti določeno (bolj interno) metodo `__hash__`, ki jo slovarji (in množice) potrebujejo zato, da se bodo odločili, kam shraniti določen ključ (oz. element). Če podatkovni tip to metodo ima, je *hashable*; če ne, ne in ne more služiti kot ključ slovarja (in biti elemnet množice). Metode `__hash__` nikoli ne kličemo neposredno, temveč pokličemo funkcijo `hash(obj)`, ki pokliče `obj.__hash__`.

```
hash("Berta")
-3551166701062947909

"Berta".__hash__() # tega ne počnemo, temveč kličemo `hash`
-3551166701062947909
```

(Ta ovinek je pogost. Tudi funkcija `len(obj)` v resnici pokliče `obj.__len__` in vrne njen rezultat. Metode `__len__` nikoli ne kličemo neposredno. Kot tudi pišemo `s[i]` in ne `s.__getitem__(i)`. Ker nismo neumni.)

Zdaj pa se vrnimo h generatorjem. Tudi gornja metoda `next` je samo ovinek: `next` pokliče metodo `__next__`. Podatkovni tipi, ki imajo metodo `__next__` so *iterable*. Konkretno, takim objektom rečemo *iteratorji*.

Generatorji so vrsta iteratorjev. Niso pa generatorji edini iteratorji. V Pythonu je kup priložnostnih iteratorjev. Iterator dobimo s funkcijo `iter`, ki ji podamo objekt, prek katerega bi radi iterirali. Če imamo seznam `s` in pokličemo `iter(s)`, bomo dobili iterator prek tega seznama.

```
s = ["Ana", "Berta", "Cilka"]
t = iter(s)
t

<list_iterator at 0x1068569e0>

next(t)

'Ana'

next(t)

'Berta'

next(t)

'Cilka'
```

Funkcija `iter` je sicer enaka prevara kot `hash`, `len` ali `iter`: seznamami imajo metodo `__iter__`. Funkcija `iter` torej le pokliče `__iter__` in ta vrne iterator, torej objekt, ki ima `__next__` in ta vrača zaporedne elemente seznama.

Prav tako imajo metodo `__iter__` tudi niz (vrne iterator, ki vrača zaporedne znake niza), množica in terka (vračata zaporedne elemente) in slovar (vrača ključe v nekem kakršnemkoli vrstnem redu).

```
d = {"Ana": 12, "Berta": 20, "Cilka": 15}
t = iter(d)
t

<dict_keyiterator at 0x1068a6520>

next(t)

'Ana'

next(t)

'Berta'

next(t)

'Cilka'
```

Kaj torej počne zanka `for`?

Z zanko `for` lahko gremo čez tiste in natančno tiste reči, ki so iterabilne - torej, ki imajo metodo `iter`.

Zanka `for` je zelo preprosta reč. Če imamo zanko

```
for spremenljivka in nekaj:
```

bo poklicala `g = iter(nekaj)`, potem pa izvajala `spremenljivka = next(g)`, dokler `next(g)` ne vrne napake `StopIteration`. Napako bo seveda skrila, saj v resnici ne gre za napako, temveč le za signal, da je delo opravljeno. Pokažimo kar primer. Vzemimo

```
s = ["Ana", "Berta", "Cilka"]
```

Najprej ga prehodimo s `for`.

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

```
Ana
Berta
Cilka
```

Kar se v resnici dogaja zgoraj, je to:

```
g = iter(s)
while True:
    x = next(g)
    print(x)
```

```
Ana
Berta
Cilka
```

```
-----
StopIteration                                Traceback (most recent call last)
/var/folders/f9/lgq5ysmn24j3lkss6zkgmtr00000gn/T/ipykernel_96026/2251081137.py in <module>
      1 g = iter(s)
      2 while True:
----> 3     x = next(g)
      4     print(x)
```

StopIteration:

No, skoraj to. Naš `while` se konča z napako, `for` pa to napako prestreže. Tega še ne znamo, a pokažimo, saj bodo mnogi itak razumeli.

```
g = iter(s)
while True:
    try:
        x = next(g)
        print(x)
    except StopIteration:
        break
```

```
Ana
Berta
```

Cilka

To je to. To v resnici dela zanka `for`.

Na doslejšnjih predavanjih smo govorili stvari kot "z zanko `for` lahko gremo prek seznamov", "zanka `for` zna iti tudi prek množic" ... Figo. Zanka `for` ne zna ničesar. Vse delo opravijo iteratorji. Zanka `for` od objekta zahteva iterator in ga kliče.

Generatorji so tudi iteratorji; tudi iteratorji so iteratorji

Naslov zveni malo bedasto. Gre le zato, da imajo iteratorji metodo `iter` in če jo pokličemo, iteratorji vrnejo kar samega sebe. Zato lahko `for`-u damo seznam ali generator, in `for` lahko v vsakem primeru pokliče `iter`.

```
t = iter([])
t
<list_iterator at 0x106855450>
iter(t)
<list_iterator at 0x106855450>
```

Številka v obliki `0x...nekaj`, pove, kje v pomnilniku se nahaja ta reč. Na ta način lahko prepoznamo, ali gre za isto stvar ali ne. Tu je številka enaka, torej `iter(t)` vrne kar sam `t`. Kot sem rekel.

Kaj so `zip`, `range`, `enumerate` in podobni?

Nekoč smo imeli

```
imena = ["Ana", "Berta", "Cilka", "Dani", "Ema"]
teze = [72, 65, 75, 68, 63]
```

in ko sem povedal za `zip(imena, teze)`, sem rekel, da se vede, kot da bi vrnil seznam parov imen in tež. Vendar nisem napisal

```
zip(imena, teze)
temveč
list(zip(imena, teze))
[('Ana', 72), ('Berta', 65), ('Cilka', 75), ('Dani', 68), ('Ema', 63)]
```

in rekel, da spreglejmo tisti `list`.

`zip` je v resnici generator. Ko ga podamo `list`-u, le da z nekakšno zanko `for` pobere vse njegove elemente in jih zloži v novi seznam. Kasneje smo `zip` uporabljali le v zanki `for`, ki ji je vseeno, ali dobi seznam ali generator. Zato sem lahko govoril, da se `zip` vede, kot da bi vrnil seznam.

Najprej preverimo, da je res, kar govorim.

```

t = zip(imena, teze)
t

<zip at 0x1068fc380>

next(t)

('Ana', 72)

next(t)

('Berta', 65)

next(t)

('Cilka', 75)

```

Res je generator. Ker po današnjem predavanju kar pokamo od pameti, pa napišimo svoj `zip`. Če torej `zip`-a še ne bi bilo, bi si ga sami sprogramirali približno tako.

```

def my_zip(s, t):
    for i in range(len(s)):
        yield s[i], t[i]

t = my_zip(imena, teze)
print(next(t))
print(next(t))

('Ana', 72)
('Berta', 65)

for ime, teza in my_zip(imena, teze):
    print(ime, teza)

Ana 72
Berta 65
Cilka 75
Dani 68
Ema 63

```

Takle `zip` bi delal samo, če mu podamo stvari, ki jih je možno indeksirati (recimo sezname), pri čemer mora biti za prvo stvar (`s`) možno poklicati `len`, poleg tega pa prva stvar ne sme biti daljša od druge. In, končno, tale `zip` sprejme samo dva argumenta, pravi `zip` pa jih poljubno. A za zdaj bodi. Boljše naredimo kasneje.

Kaj pa `enumerate`. Isto, seveda. Tudi generator.

```

t = enumerate(imena)
t

<enumerate at 0x1068ef040>

next(t)

```

```
(0, 'Ana')
```

```
next(t)
```

```
(1, 'Berta')
```

Tudi `enumerate` bi si znali narediti sami.

```
def my_enumerate(s):  
    for i in range(len(s)):  
        yield i, s[i]  
  
for i, ime in my_enumerate(imena):  
    print(i, ime)
```

```
0 Ana  
1 Berta  
2 Cilka  
3 Dani  
4 Ema
```

`range`? Isto. Tule je preprostejša različica `range`-a, ki zahteva zgornjo in spodnjo mejo, korak pa je vedno 1.

```
def my_range(start, end):  
    i = start  
    while i < end:  
        yield i  
        i += 1
```

```
for i in my_range(5, 8):  
    print(i)
```

```
5  
6  
7
```

Od Pythona 3 naprej vse tovrstne funkcije vračajo iteratorje. Tudi metode slovarjev, `values`, `items` in, hm, `keys`, vrnejo iteratorje.

Lepota iteratorjev je, da nikoli ne ustvarijo celotnih seznamov. To je hitrejše in prijaznejše do pomnilnika. Obenem uporaba iteratorjev vodi k malo drugačnemu pogledu na potek programa - predstavljamo si ga lahko kot tok podatkov. V Pythonu je to malo manj izrazito, ker ni ravno idealen jezik za tak način programiranja. A vendar se ga da prikazati tudi z njim.

Najbolj frajerski so iteratorji, ki generirajo neskončna zaporedja. V `itertools` imamo iterator `count(n)`, ki šteje od `n` do neskončno. Če argument `n` izpustimo, bo štel od 0.

```
from itertools import count
```

```

for i in count(5):
    if i == 10:
        break
    print(i)

```

5
6
7
8
9

Če ne bi imeli `enumerate`, bi namesto `enumerate(s)` pisali `zip(count(), s)`.
 Iterator `zip` bi šel pač hkrati prek iteratorja, ki ga vrne `count` in prek `s`.

Tudi `count` ni vesoljska znanost, ta pa res ne.

```

def my_count(n=0):
    while True:
        yield n
        n += 1

```

```

for i in my_count(5):
    if i == 10:
        break
    print(i)

```

5
6
7
8
9

`count()` izgleda relativno neuporaben. Ni. Sam ga imam zelo rad in ga pogosto uporabim. V določenih situacijah (katerih opis pa tule izpustimo) ga dejansko uporabljam namesto `enumerate`. Prav pa pride tudi vedno, kadar nekaj štejemo, iščemo ... in ne vemo, do kod bo potrebno šteti.

Katero je prvo število, katerega kvadrat presega 1000?

```

for i in count():
    if i ** 2 > 1000:
        break
print(i)

```

32

Seveda lahko sestavimo tudi kvadrate vseh naravnih števil.

```
kvadrati = (x ** 2 for x in count())
```

In potem spustimo zanko čeznje.

```

for x in kvadrati:
    if x > 1000:
        break
print(i)

```

32

Katero pa je prvo popolno število? Se pravi, prvo število, ki je enako vsoti svojih deliteljev?

```

for n in count(1): # šteti začnemo pri 1, sicer bi dobili 0
    if n == sum(x for x in range(1, n) if n % x == 0):
        break
print(n)

```

6

Še bolj imenitno: s `count` si lahko pripravimo generator vseh popolnih števil!

```

popolna = (n for n in count(1) if n == sum(x for x in range(1, n) if n % x == 0))
next(popolna)

```

6

```
next(popolna)
```

28

```
next(popolna)
```

496

```
next(popolna)
```

8128

`for n in count(1)` poskrbi, da bo šel `n` od 1 do neskončno, `if` pa izloča vsa nepopolna števila. Ko pokličemo `next`, bo zanka gnala `n` toliko časa, da bo `if` zadovoljen in generator bo "izgeneriral" `n`. Ko naslednjič pokličemo `next`, teče zanka naprej. In tako vsakič, ko zahtevamo naslednje število. Prva tri je našel hitro, na 8128 pa je bilo potrebno že malo počakati.

Če razumete tale, zadnji generator, vam je jasno vse.

Resnični `zip`, `range`, `enumerate`

Zgornje funkcije so bile malo poenostavljene. Resnične so sprogramirane v C-ju. Če bi jih hoteli narediti v Pythonu, pa tudi ne bi bile bistveno daljše od gornjih približkov, le malo bolj previdno se jih moramo lotiti.

```

def my_zip(*args):
    args = [iter(arg) for arg in args]
    try:

```



```

    while True:
        yield tuple([next(arg) for arg in args])
    except StopIteration:
        pass

```

Funkcija dobi poljubno število argumentov, zato `*args`.

V prvi vrstici takoj pokličemo `iter` za vsak argument in to zložimo kar nazaj v `args`. Tako zagotovimo, da imamo same iteratorjeve, se pravi, da nam je vseeno, ali je nek argument (že) iterator ali pa (še) seznam ali kaj podobnega.

Nato bomo v neskončnost ponavljali `tuple([next(arg) for arg in args])`. Z `[next(arg) for arg in args]` sestavimo seznam vsega, kar vračajo posamični iteratorji, in to pretvorimo v terko, ker `zip` pač vrača terke. (Zakaj ne kar `tuple(next(arg) for arg in args)`? Daljša zgodba. PEP-479.)

`try-except` prestrežeta `StopIteration` in končata delo.

Preverimo, da res deluje: pokličimo ga s tremi argumenti, pri čemer je en (neskončen) generator, ostala dva pa sta tudi različno dolga.

```

for i, ime, teza in my_zip(count(), imena, teze[:-2]):
    print(i, ime, teza)

```

```

0 Ana 72
1 Berta 65
2 Cilka 75

```

Kaj pa `range`?

```

def my_range(start, end=None, step=1):
    if end == None:
        end = start
        start = 0
    while start < end if step > 0 else start > end:
        yield start
        start += step

```

```

list(my_range(5))
[0, 1, 2, 3, 4]
list(my_range(5, 10))
[5, 6, 7, 8, 9]
list(my_range(5, 15, 2))
[5, 7, 9, 11, 13]
list(my_range(15, 5, -2))
[15, 13, 11, 9, 7]

```

Pravi `range` zna še marsikaj, ampak "generatorski" del smo podelali.

`enumerate` je v primerjavi z njima res preprost.

```
def my_enumerate(s, start=0):
    s = iter(s)
    try:
        while True:
            yield start, next(s)
            start += 1
    except StopIteration:
        pass

list(my_enumerate(imena, start=5))

[(5, 'Ana'), (6, 'Berta'), (7, 'Cilka'), (8, 'Dani'), (9, 'Ema')]
```

Konec

Enkrat se bo treba ustaviti. V zvezi z vsemi temi rečmi bi bilo mogoče povedati še zelo veliko. Tole je ena najbolj kul tem v Pythonu. Pravzaprav treba priznati, da to niti ni tema iz Pythona. V ozadju tega, kar počnemo tule, je poseben slog programiranja, funkcijsko programiranje. Python ga omogoča in med "normalnimi" jeziki je za takšen slog pravzaprav eden boljših. Obstajajo pa jeziki, ki so posebej narejeni za takšno programiranje. Če je bilo komu tole, kar smo počeli doslej, všeč naj si nujno ogleda kak SML ali Haskell, morda pa ga bo zabaval tudi Racket (dialekt Lispa).

V Pythonu pa si bo ta, ki so mu bile te reči všeč, dobro ogledal module `functools`, `iterools` in `operator`.