

## Generatorji

### Generatorski izrazi

Generatorje načelno poznamo. Sintaktično so podobni izpeljanim seznamom in množicam - oziroma, še bolj, *neobstoječim* izpeljanim terkam. Tole je generator, ki generira (= sproti sestavlja!) kvadrate števil do 10

```
kvadrati = (x ** 2 for x in range(5))
```

in tole je generator, ki generira (vsa) praštevila

```
from itertools import count
```

```
prastevila = (x for x in count(2) if all(x % k != 0 for k in range(2, x)))
```

Pognati ga je seveda potrebno previdno, saj sam od sebe nikoli ne bo jenjal generirati.

```
for p in prastevila:
    if p > 20:
        break
    print(p)
```

```
2
3
5
7
11
13
17
19
```

Mimogrede se spomnimo še, kaj generator pravzaprav počne: generira nov element vsakič, ko pokličemo `next`.

```
prastevila = (x for x in count(2) if all(x % k != 0 for k in range(2, x)))
```

```
next(prastevila)
```

```
2
```

```
next(prastevila)
```

```
3
```

```
next(prastevila)
```

```
5
```

Kaj pa počne generator med dvema klicema `next`? Neumno vprašanje. Kaj počne... nič. Pač, vsakič, ko pokličemo `next` vzame naslednjo vrednost, ki jo vrne zanke ter jo kvadrira in jo vrne. Oziroma preveri, ali je praštevilo, in če ni, vzame naslednjo in naslednjo, dokler ne naleti na praštevilo. Tega vrne in to

je to. Ob naslednjem `next`-u pa naprej. Pri tem seveda na nek način shranjuje "stanje" - zanka `for`, ki se skriva v generatorju, ve, do kod je prišla. (To sem napisal nekoliko poenostavljeno; v resnici je (še) preprosteje.)

### Seznam rekordov

Omejitev takšnega generatorja je, da ga smejo sestavljati le izraz, zanka in pogoj. Edino "stanje", ki ga vzdržuje generator, je stanje zanke. Težko si beleži, kaj se je dogajalo v preteklosti, vedno vidi le trenutni korak. Kako omejujoče je to, uvidimo, če poskusimo sestaviti generator `rekordi`, ki za podani seznam (terko, datoteko, ...) vrača zaporedne rekorde: za vsak element preverimo, ali je večji od elementov, ki smo jih generirali doslej - za kar seveda zadošča preveriti, da je večji od zadnjega generiranega elementa. Če imamo seznam

```
s = [3, 5, 8, 4, 2, 8, 3, 10, 11, 5, 13, 4]
```

mora generator `rekordi(s)` izgenerirati 3, 5, 8, 10, 11, 13.

```
def rekordi(s):
    r = []
    for x in s:
        if not r or x > r[-1]:
            r.append(x)
    return r
```

```
rekordi(s)
```

```
[3, 5, 8, 10, 11, 13]
```

Da nam bo kasneje lažje razmišljati, rešimo še malo drugače: izognimo se škiljenju v `r`, temveč vanj le dodajamo, zadnji element pa shranjujmo ločeno.

```
def rekordi(s):
    prejsnji = None
    r = []
    for x in s:
        if prejsnji is None or x > prejsnji:
            r.append(x)
            prejsnji = x
    return r
```

```
rekordi(s)
```

```
[3, 5, 8, 10, 11, 13]
```

### Funkcija, ki vrne generator

Z gornjo funkcijo ni nič narobe, dokler so sezname kratki in nas ne moti, da so shranjeni v pomnilniku. In dokler je argument, `s` v resnici seznam in ne generator, ki elemente vrača, kadar pridejo (na primer po omrežni povezavi, ali pa jih sproti vnaša uporabnik) in bi jih radi tudi generirali sproti.

Zanka v gornji funkciji ni "čista" (v smislu, v kateri so nekatere funkcije *čiste funkcije*, pure function): poleg tega, da preverja nek pogoj in dodaja v seznam, tudi spreminja vrednost **prejsnji**. V generatorskih izrazih vsak **x** pričakata enak pogoj in enak izraz - če je **x** enak **13**, ga generator, ki sestavlja praštevila bodisi sprejme bodisi zavrne, ne glede na to, kaj se je dogajalo pred njim. Pythonovi generatorski izrazi sicer ne zahtevajo čistoče -- posebej walrus jo zlahka povozi --, jo pa ljubijo. Z nekaj zvijanja rok lahko stlačimo **rekordi** v generatorski izraz, vendar bo grd.

Rešitev bomo našli v *generatorskih funkcijah*. A preden jih napišemo zares, se vrnimo k začetnima generatorjema. Zgoraj smo pisali

```
kvadrati = (x ** 2 for x in range(5))
```

Če bi človek pogosto potreboval generatorje kvadratov (dasiravno si ne predstavljam, kakšen človek bi to mogel biti), bi si morda pripravil kar funkcijo, ki mu vrne takšen generator.

```
def f_kvadrati():  
    return (x ** 2 for x in range(5))
```

Še več, dotični bi nemara potreboval različno dolge seznime kvadratov in bi funkciji dodal še argument.

```
def f_kvadrati(n):  
    return (x ** 2 for x in range(n))
```

Da to uporabi, mora poklicati funkcijo. Če je poprej pisal

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

```
0  
1  
4  
9  
16
```

Bo zdaj pač poklical funkcijo

```
for x in f_kvadrati(5):  
    print(x)
```

```
0  
1  
4  
9  
16
```

Ali, če bo ravno hotel

```

kvadrati = f_kvadrati(5)
for x in kvadrati:
    print(x)

0
1
4
9
16

```

Podobno bi lahko sestavil funkcijo, ki vrne generator praštevil in jih, spet, če bi hotel, omejil.

```

def f_prastevila(n):
    return (x for x in range(2, n) if all(x % k != 0 for k in range(2, x)))

for p in f_prastevila(10):
    print(p)

2
3
5
7

```

## Generatorske funkcija

Funkciji, ki vrača generator, rečemo *generatorska funkcija*. Ne brez razloga: vračati generator ni kar tako, kot vračati, recimo, terko, pa da bi potem takšnim funkcijam rekli "terčne funkcije", onim, ki vračajo slovarje, pa *slovarske*. Ne, ne, *generatorske funkcije* so čisto posebna vrsta funkcij. Imajo jih mnogi spodobni jeziki (pa tudi nekateri drugi).

Gornji generatorski funkciji sta vrnili generator, ki sta ga sestavili z generatorskim izrazom. V Pythonu, Javascriptu, C# (pa tudi v Php) bo funkcija vrnila generator, če namesto `return`-a vrača rezultat(e) z `yield`. Spodnji funkciji sta ekvivalentni gornjim:

```

def f_kvadrati(n):
    for x in range(n):
        yield x ** 2

def f_prastevila(n):
    for x in range(2, n):
        if all(x % k != 0 for k in range(2, x)):
            yield x

```

Detajli sledijo spodaj, ne skrbi. Za zdaj se le prepričajmo, da funkciji res vrneti generator:

```
prastevila = f_prastevila(10)
```

```
prastevila
<generator object f_prastevila at 0x106317010>
for x in prastevila:
    print(x)
2
3
5
7
```

In, da bomo še bolj prepričani, poskusimo počasi, ročno:

```
prastevila = f_prastevila(10)
next(prastevila)
2
next(prastevila)
3
next(prastevila)
5
```

Za podrobnejši uvid, napišimo še en, še preprostejši generator.

```
def endvatri():
    print("Pa začnimo!")
    yield 1
    print("Pa nadaljujmo.")
    yield 2
    print("Še malo, pa bo konec!")
    yield 3
    print("Pa smo končali.")
```

Pokličimo funkcijo, pridobimo generator.

```
g = endvatri()
```

Tole je pomembna opazka: nič se ni zgodilo. Nič od kode, ki je zapisana v funkciji, se ni izvedlo. Klic funkcije je zgolj vrnil generator.

Pokličimo next.

```
next(g)
Pa začnimo!
1
```

Zdaj se je koda generatorja začne izvajati: izvedel se je `print`, `yield` pa je vrnil vrednost - podobno, kot bi jo `return`. Generator tu vrne vrednost in s tem se njegovo izvajanje seveda ustavi.

Nadaljevalo se bo z novim klicem `next`.

```
next(g)
```

Pa nadaljujmo.

```
2
```

Potem se reč spet ustavi in nadaljuje ob naslednjem `next`.

```
next(g)
```

Še malo, pa bo konec!

```
3
```

Ob naslednjem `next` se izvede `print`, nove vrednosti pa ni in generator to sporoči tako, da sproži izjemo `StopIteration`.

```
next(g)
```

Pa smo končali.

```
-----
StopIteration                                Traceback (most recent call last)
Cell In[34], line 1
----> 1 next(g)
```

StopIteration:

Spomnimo se, kako sta videti naša generatorja kvadratov in praštevil:

```
def f_kvadrati(n):
    for x in range(n):
        yield x ** 2

def f_prastevila(n):
    for x in range(2, n):
        if all(x % k != 0 for k in range(2, x)):
            yield x
```

Isto, samo da je `yield` v zanki namesto na prostem.

`yield` je seveda podoben `return`-u. Naivna - in skoraj pravilna - razlaga `yield`-a je, da "zamrzne" izvajanje funkcije, in da se bo izvajanje ob "naslednjem klicu funkcije" nadaljevalo od `yield`-a naprej. Zgolj skoraj pravilna je zato, ker ne gre za klice funkcije: funkcijo smo poklicali le v začetku in vrnila je generator. Gre torej za "klice" generatorja, prek `next`.

## Generator rekordov

V čem je prednost generatorskih funkcij pred generatorskimi izrazi? Mar nismo kar vzdihovali od sreče in navdušenja, ko smo odkrili, da lahko namesto

```
t = []
for x in s:
    if x % 2 == 0:
        t.append(x)
```

pišemo

```
t = [x for x in s if x % 2 == 0]
```

Nismo zdaj naredili ravno koraka v nasprotno smer, ko namesto enega samega generatorskega izraza, `(x ** 2 for x in range(n))` pišemo celo funkcijo, v več vrsticah, z zanko?

Ne nujno. Saj vemo: ne da se vsega v eni vrstici. Generatorske funkcije nam bodo rešile problem s stanjem pri rekordih.

Funkcija, ki smo si jo napisali za prepoznavanje rekordov, je bila takšna:

```
def rekordi(s):
    prejsnji = None
    r = []
    for x in s:
        if prejsnji is None or x > prejsnji:
            r.append(x)
            prejsnji = x
    return r
```

Tole pa je generatorska funkcija:

```
def rekordi(s):
    prejsnji = None
    for x in s:
        if prejsnji is None or x > prejsnji:
            yield x
            prejsnji = x
```

```
r = rekordi(s)
```

```
next(r)
```

```
3
```

```
next(r)
```

```
5
```

```
next(r)
```

```
8
```

```
next(r)
```

```
10
```

Vrednost **prejsnji**, ki je prej nismo mogli stlačiti v eno vrstico, je brez težav našla mesto v generatorski funkciji.

Ne spreglejte: tale **rekordi** še ni generator: **rekordi** je generatorska funkcija, ki jo moramo poklicati in njen rezultat, **r** je generator.

### Terminološka zagata

Obstajajo *generatorji* in *iteratorji*. Tule uporabljamo le prvi izraz, drugemu pa se izogibam, ker bi potem moral razložiti razliko. Za to pa bi morali zariniti še globlje in pokazati, kako definirati razred, ki se vede kot generator ali kot iterator (ali kot oboje).

### Še dva primera

Navrzimo še dva primera.

Generator **unikati** bo vsako vrednost izgeneriral le enkrat, ponovitve pa preskočil.

```
def unikati(s):
    past = set()
    for x in s:
        if x not in past:
            yield x
            past.add(x)
```

```
s
```

```
[3, 5, 8, 4, 2, 8, 3, 10, 11, 5, 13, 4]
```

```
list(unikati(s))
```

```
[3, 5, 8, 4, 2, 10, 11, 13]
```

Manjša, s temo nepovezana, a pomembna pripomba: tole deluje le, če **s** vrača stvari, ki jih je mogoče zlagati v množico. Če bi bil med njimi kakšen seznam ... Ne bo šlo.

Še en generator iz obveznega programa: Fibonačijeva števila (ker vedno pozabim, s koliko c-ji se piše **Fibonac(c)i**, sem se ga odločil pisati kar po domače).

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

To je to.

```
for x in fibonači():  
    if x > 20:  
        break  
    print(x)
```

1  
1  
2  
3  
5  
8  
13