FUNKCIJSKO PROGRAMIRANJE

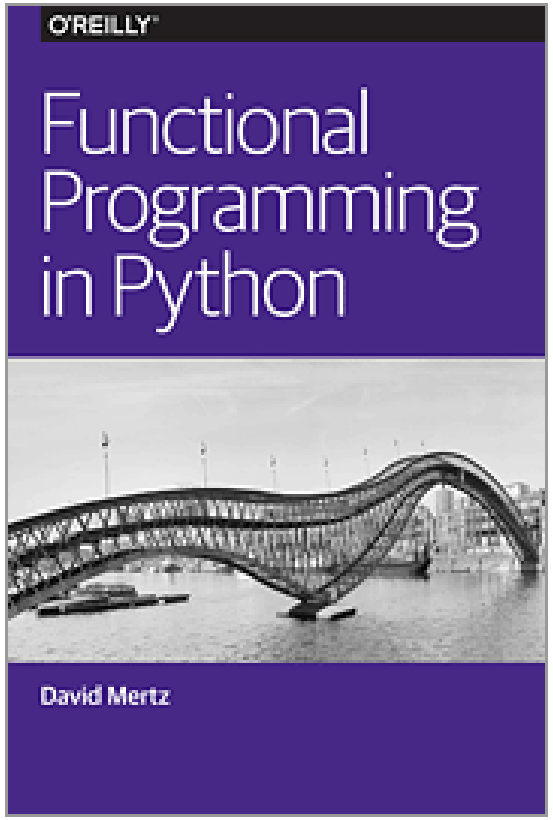
2023/24

funkcijsko programiranje v Pythonu

FP v Pythonu

- Python ni čisti funkcijski jezik, nudi različne paradigme programiranja
- ima zmožnosti funkcijskega programiranja
  - anonimne funkcije
  - funkcije višjega reda
  - map(), reduce(), filter()
  - izpeljani seznami
  - iteratorji
  - generatorji
  - memoizacija
  - currying
  - in še druge…

- e-knjiga (10-dnevni brezplačni dostop, povezava):
  David Mertz: Functional Programming in Python, O'Reilly, 2015



Anonimne funkcije

```
# poimenovana funkcija
def kvadrat(x):
    return x**2
```

ali

```
# anonimna funkcija
kvadrat = lambda x: x**2
```

Funkcije višjega reda

- definicija funkcij višjega reda

```
def izbira(ocena):
    if ocena >5:
        return lambda dan: "V " + dan + " praznujemo."
    else:
        return lambda tocke, datum:
                    "Dobil sem samo " + str(tocke)
                    + " tock. Dne " + datum
                    + " je naslednji rok."

>>> rezultat = izbira(5)
>>> rezultat(33, "1.1.2012")
'Dobil sem samo 33 tock. Dne 1.1.2012 je naslednji rok.'

>>> rezultat = izbira(10)
>>> rezultat("petek")
'V petek praznujemo.'
```

- uporaba vgrajenih funkcij

```
>>> map(lambda x: x**2, range(1,5))
[1, 4, 9, 16]
>>> filter(lambda x: x%2==0, range(10))
[0, 2, 4, 6, 8]
>>> reduce(lambda x,y: x+y, [47, 11, 42, 13])
113
```

Funkcijske ovojnice

- ovojnica = telo funkcije + okolje
- možni načini uporabe :
  - uporaba privzetih vrednosti
  - ovijanje funkcije v zunanjo funkcijo
  - uporaba posebnih razredov in orodij (closure, Bindings)

dinamični doseg

```
>>> N = 10
>>> def pristejN(i):
      return i+N


>>> pristejN(7)
17
>>> N = 20
>>> pristejN(7)
27
```

zaprtje s privzeto vrednostjo ("zapečena" v funkcijo)

```
>>> N = 10
>>> def pristejN(i, n=N):
      return i+n


>>> pristejN(7)
17
>>> N = 20
>>> pristejN(7)
17
```

zaprtje z ovijanjem funkcije v zunanjo funkcijo

```
>>> N = 10
>>> def pristejNx(N):
      def pristejN1(i):
        return i+N
      return pristejN1
>>> moja = pristejNx(10)
>>> moja(5)
15
>>> N = 20
>>> moja(5)
15
```

Sestavljeni seznami

- način avtomatskega sestavljanja seznamov v skladu z matematično formulacijo
- Lahko običajno nadomestijo map, filter in reduce,
- $A = \{f(x) \mid x \in D, pogoj(x)\}$, npr. $A = \{x^2 \mid x \in \mathbb{N},\ 5 \leq x \leq 15\}$
- sestavljen seznam (list comprehension)

```
[funkcija(x) for x in D if pogoj(x)]
```

je enakovredno

```
r = []
for e in D:
    if pogoj(e):
        r.append(funkcija(e))
```

- primer:

```
def prastevilos(n):
    return not [x for x in range(2, n) if n % x == 0]
```

Gnezdeni sestavljeni seznami

Primer: poiščimo pitagorejske trojčke (za njih velja $x^2 + y^2 = z^2$)

```
[(x,y,z) for x in range(1,30)
    for y in range(x,30)
        for z in range(y,30)
            if x**2 + y**2 == z**2]
```

rezultat:

```
[(3, 4, 5), (5, 12, 13), (6, 8, 10),
 (7, 24, 25), (8, 15, 17), (9, 12, 15),
 (10, 24, 26), (12, 16, 20), (15, 20, 25),
 (20, 21, 29)]
```

Sestavljene množice in slovarji

```
# obstajajo tudi sestavljene MNOŽICE
        squares_set = {x**2 for x in [1,2,3,4,5]}
        print(squares_set)
        print(type(squares_set))

    rezultat:
        {16, 1, 9, 25, 4}
        <class 'set'>
```

```
# obstajajo tudi sestavljeni SLOVARJI
        squares_dict = {x : x**2 for x in [1,2,3,4,5]}
        print(squares_dict)
        print(type(squares_dict))

    rezultat:
        {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
        <class 'dict'>
```

Dekoratorji kot funkcije višjega reda

- dekorator: funkcija, ki sprejme neko funkcijo (f) in jo vrne ovito v ovoj (wrapper):

```
def decor(f):
    def wrapper():
        f()
    return wrapper
```

- pripis dekoratorja funkciji:

```
@decor          # dekorator
def f():        # definicija dekorirane funkcije
    ...
```

- klic dekorirane funkcije: `f()`
  - zelo podobno (vendar ne čisto enako kot): `decor(f)()`

Dekoratorji in argumenti funkcij

- pri klicu funkcije lahko v Pythonu podamo pozicijske in imenovane argumente
- presežek nepodanih **pozicijskih** argumentov je shranjen v spremenljivko, zapisano z **eno** zvezdico (npr. *args), ki je terka
- presežek nepodanih **imenovanih** argumentov je shranjen v spremenljivko, zapisano z **dvema** zvezdicama (npr. **kwargs), ki je slovar

```
def test(a, b, c, *args, **kwargs):
    return (args, kwargs)

>>> test(1, 2, 3, 4, 5, 6, x=5, y=12)
((4, 5, 6), {'x': 5, 'y': 12})
```

Dekoratorji in argumenti funkcij

- pravilni prenos argumentov v dekorirano funkcijo

```
def decor(f):
    def wrapper(*args,**kwargs):
        print("Kličem funkcijo %s" % f.__name__)
        f(*args,**kwargs)
    return wrapper


@decor
def funk(arg1, arg2, ..., argn):
       ...
```

Currying funkcij višjega reda

- denimo, da želimo definirati obliko naslednje funkcije, ki uporablja currying

```
def f(a,b,c):
    return a+b+c
```

```
f(a)
f(a)(b)
f(a)(b)(c)
```

- ročna definicija:

```
def f(x, *args):
    def f1(y, *args):
        def f2(z):
            return x+y+z
        if args:
            return f2(*args)
        else:
            return f2
    if args:
        return f1(*args)
    else:
        return f1
```

sprejme z in izračuna vsoto x+y+z

sprejme y in preda izračun f2

sprejme x in preda izračun f1

Currying: preprosteje

- uporabimo lahko modul *pymonad*

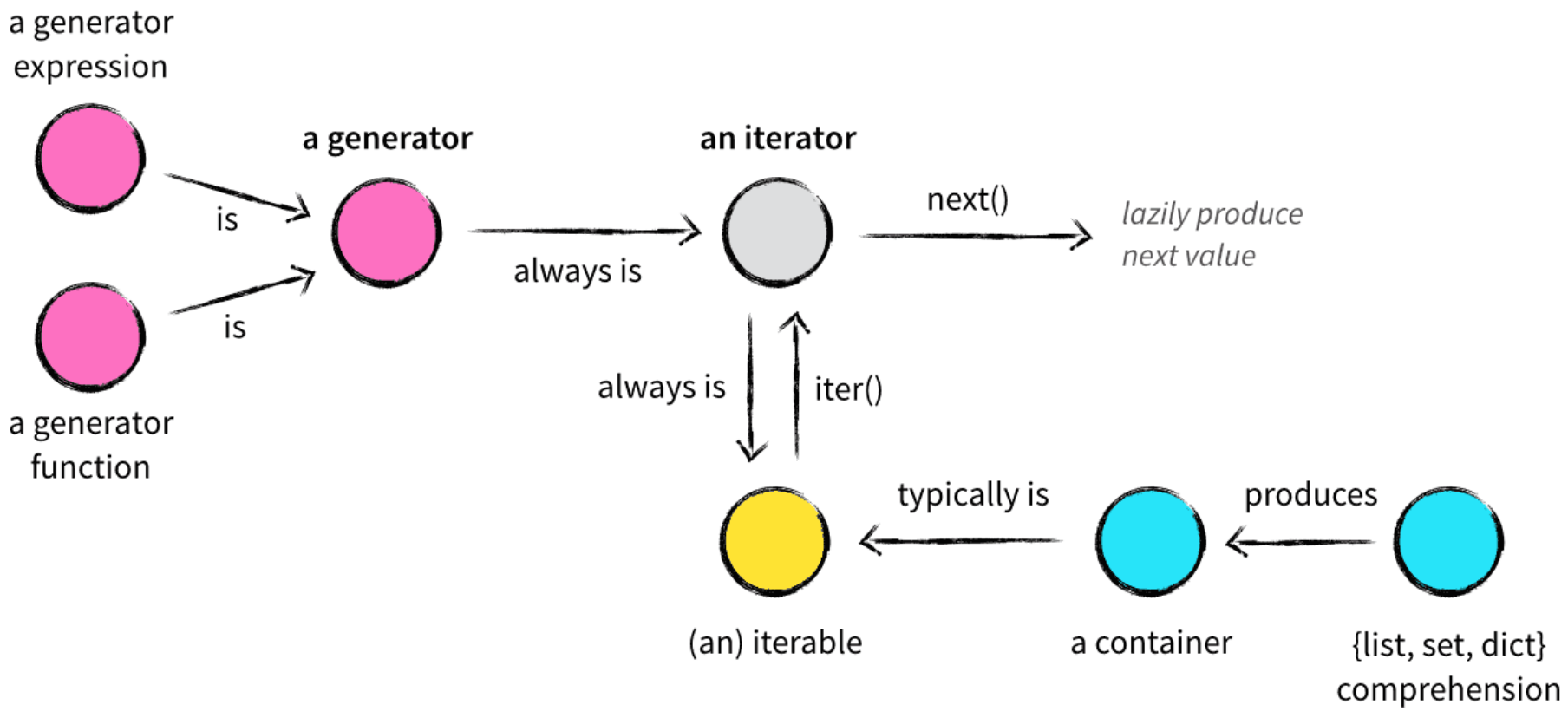
- namestitev (ukazna vrstica):

```
pip install pymonad
```

- nato uporabimo dekorator @curry

```
from pymonad import curry

@curry
def f(a,b,c):
    return a+b+c
```

Iterabilni objekti



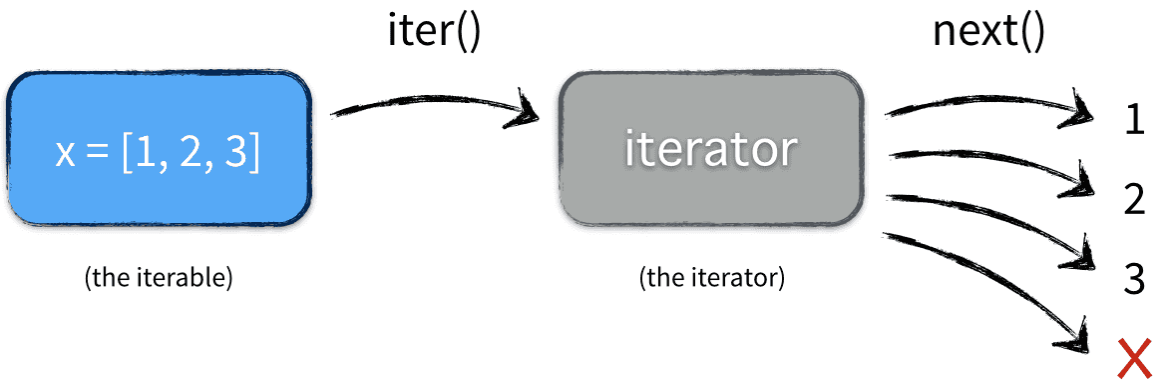
Iteratabilni objekt (razred)

- iterator je objekt, ki predstavlja tok podatkov in vrača posamezne elemente iterabilnega objekta
  - zakasnjeno izvajanje
- iterabilen razred implementira metodo **__iter__()** - vrne objekt-iterator nad tem razredom
- razred lahko implementira tudi metodo **__next__()** - vrne naslednji element do izčrpanja (**StopIteration**) ali v neskončnost
- premikanje nazaj ni možno
- funkcija **iter(objekt)** - pretvorba iterabilnega objekta v iterator

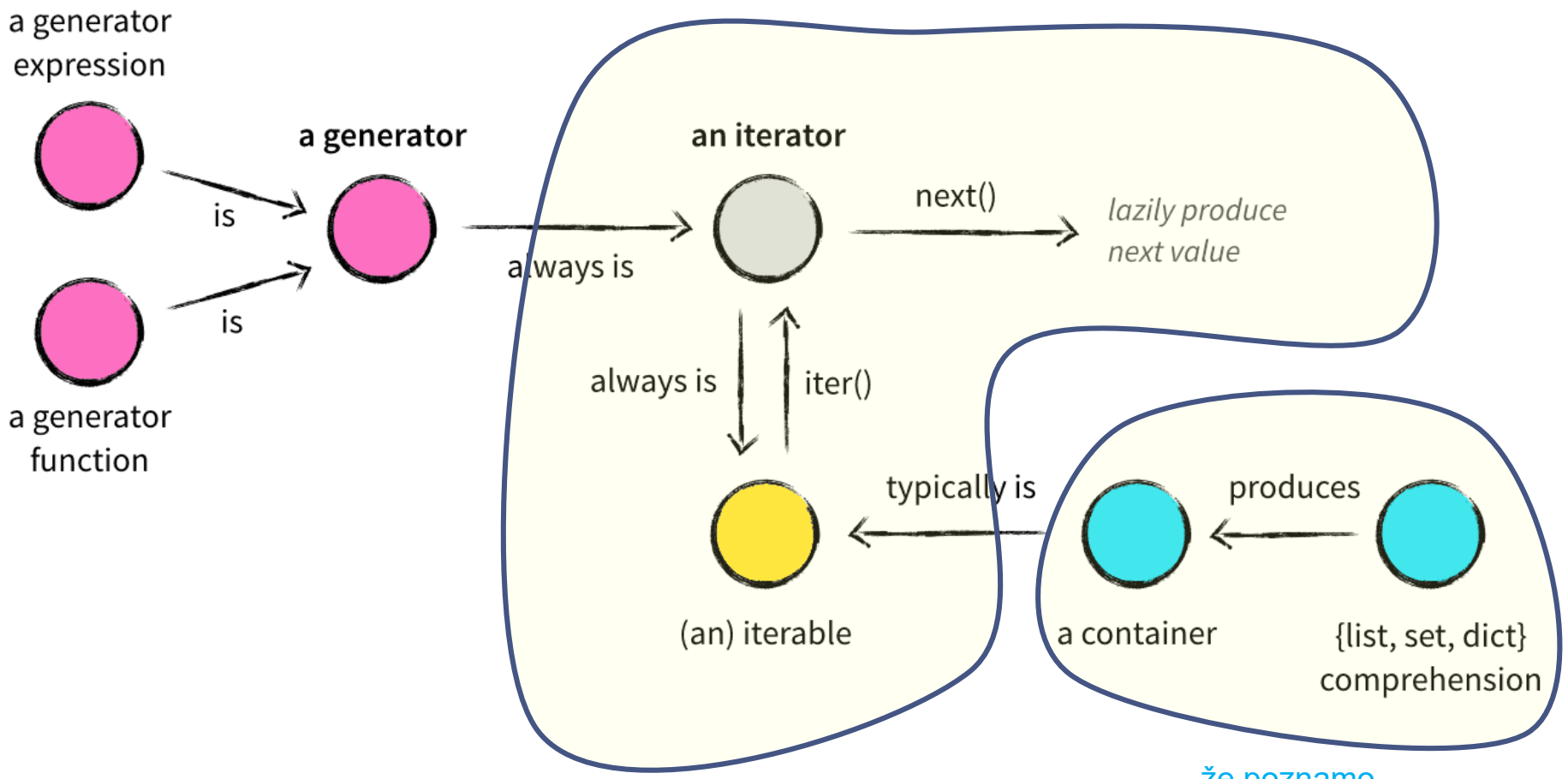
```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1


for c in Counter(3, 8):
    print(c)
```

Iterator

- funkcija **iter(objekt)** pretvori iterabilni objekt v iterator



Iterabilni objekti



že poznamo

Generator

- generatorji so tudi iteratorji
- **poenostavijo zapis** iteratorja

- so funkcije, ki iterirajo preko toka vrednosti in omogočajo zakasnjeno izvajanje izrazov:
  - izvajanje funkcije se lahko zaustavi in nadaljuje
  - lokalno okolje funkcije se ohranja tudi ob zaustavitvi
  - stavek ***yield*** zaustavi izvajanje in preda kontrolo klicočemu okolju
  - funkcija nadaljuje z izvajanjem ob klicu metode **.__next__()**
  - generiranje se zaključi s stavkom return ali izjemo StopIteration
  - generatorju lahko podamo argument z metodo **.send(arg)**

- zgodovinsko: do Pythona 3 sta obstajala range(min,max) in xrange(min,max) (takojšnje in leno evalviran range)

```
def genstevec(max):
    i = 0
    while i<max:
        yield i
        i += 1
```

```
def genstevec(max):
    i = 0
    while i<max:
        vnos = (yield i)
        if vnos == None:
            i += 1
        elif vnos>=max:
            raise StopIteration
        else:
            i = vnos
```

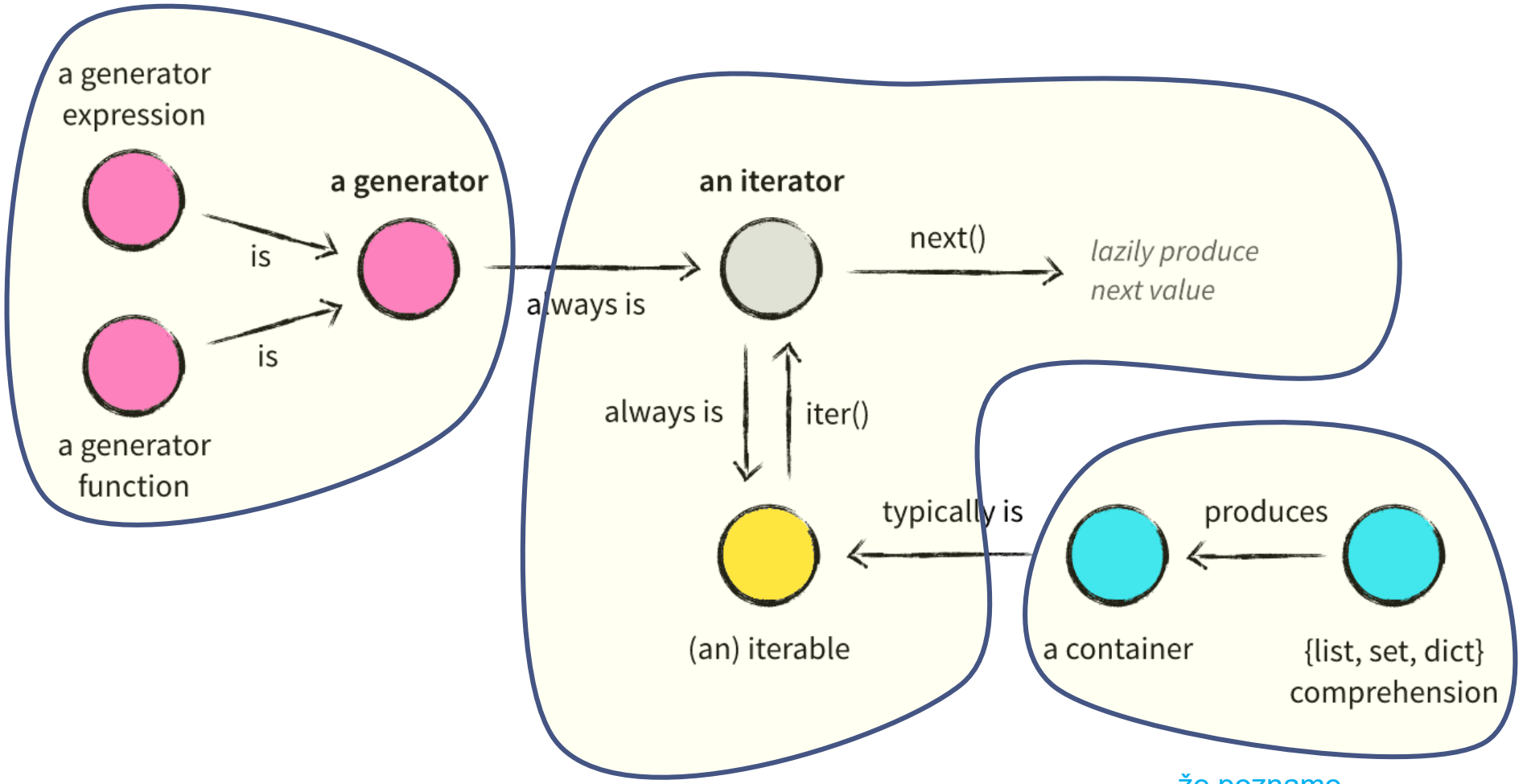
Generatorski izrazi

- krajši način za specifikacijo iteratorjev
- sintaksa podobna sestavljenim seznamom/množicam/slovarjem
- nekoliko okrnjena funkcionalnost

```
def Squares(max):
    i = 0
    while i<max:
        yield i*i
        i += 1

g1 = Squares(10)
```

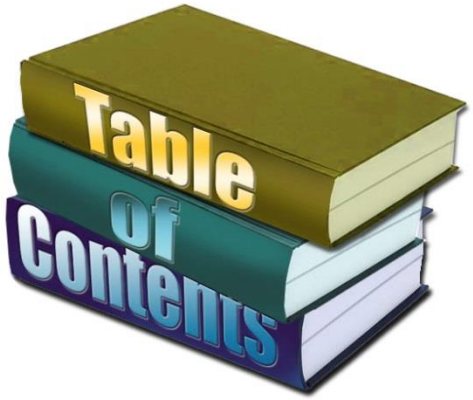
```
g2 = (x*x for x in range(10))
```

Iterabilni objekti



že poznamo

Pregled



- funkcijsko programiranje v Pythonu
- zaključek

Zaključek

cilji predmeta:

- postati boljši programer
  - naučiti se novih konceptov (polimorfizem, zakasnjena evalvacija, tokovi, memoizacija, funkcije višjega reda, ovojnice, delna aplikacija, currying, ...)
- razumeti delovanje programskega jezika
  - pridobiti sposobnost hitrega učenja novega programskega jezika
  - ločiti bolj in manj elegantne implementacije
- izstopiti iz okvira objektno-usmerjenega programiranja
  - razumeti funkcijsko in objektno-usmerjeno paradigmo
- naučiti se funkcijskega programiranja. Njegove prednosti:
  - bolj abstrakten opis problema
  - možnost paralelizacije
  - brez mutacije vrednosti (manj semantičnih napak)
  - lažje testiranje (testi enot)
  - lažji formalni dokaz pravilnosti
- dojeti "motivacijski članek" ☺

In še nekaj drugih zanimivih virov

- Dmytro Khmelenko: When To and When Not To Use Functional Programming
- Ari Joury: Why developers are falling in love with functional programming
- Richard Feldman: Why Isn't Functional Programming the Norm?

May the λ be with You!