Iterabilni objekti, iteratorji in generatorji

Programiranje 1

Univerza v Ljubljani, FMF, 2013

Alen Orbanić

Iterabilni objekti in iteratorji

- **Iterabilni objekti** so objekti, pri katerih klic funkcije `iter()` vrne iterator.
- **Iteratorji** so objekti, ki jih je mogoče iterirati s klici funkcije `next()`, ki vrača zaporedne člene zaporedja določenega v objektu. Ko se zaključi zaporedje, klic funkcije vrže izjemo `St`

```
>>> s = [1, 2, 3]
>>> it = iter(s)
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    next(it)
StopIteration
```

Tipična uporaba ali “Kako deluje zanka for?”

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

# zanka for
for i in s:
    print(i)

# ekvivalentno
it = iter(s)
while True:
    try:
        i = next(it)
    except StopIteration:
        break
    print(i)
```

- S klicem funkcije `iter()` dobimo iterator seznama.
- Izvajamo zaporedje klicev funkcije `next()` ter v vsakem obhodu zanke uporabimo vrnjen rezultat.
- Ko klic `next()` vrže izjemo `StopIteration`, se zanka zaključi.

Funkcija range()

```
>>> r = range(10)
>>> next(r)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    next(r)
TypeError: range object is not an iterator
>>> it = iter(r)
>>> it
<range_iterator object at 0x02DDDDD0>
>>> next(it)
0
>>> next(it)
1
>>> ...
```

- Rezultat klica funkcije `range()` ni iterator, je pa iterabilen objekt.

Tehnično ozadje

- Klic funkcije `iter(o)` na poljubnem objektu `o` vedno najprej preveri obstoj metode `it.__iter__()` in jo, če obstaja, kliče ter vrne njen rezultat. V kolikor metoda ne obstaja, vrže izjemo.
- Klic funkcije `next(it)` na poljubnem objektu `it` vedno najprej preveri obstoj metode `it.__next__()` in jo, če obstaja, kliče ter vrne njen rezultat. V kolikor metoda ne obstaja, vrže izjemo.

```
>>> next(1)
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    next(1)
TypeError: int object is not an iterator
>>> iter(1)
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    iter(1)
TypeError: 'int' object is not iterable
>>>
```

```
>>> s = [1, 2, 3]
>>> it = s.__iter__()
>>> it.__next__()
1
>>> it.__next__()
2
>>> it.__next__()
3
>>> it.__next__()
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    it.__next__()
StopIteration
```

Razred interval

```
class Interval:
    def __init__(self, zac, kon):
        self.i = zac
        self.kon = kon

    def __iter__(self):
        return self

    def __next__(self):
        if self.i >= self.kon:
            raise StopIteration
        t = self.i
        self.i += 1
        return t
```

```
>>> for i in Interval(1,5): print(i)

1
2
3
4
```

- V konstruktorju inicializiramo števec `self.i` in si zapomnimo zgornjo mejo `self.kon`
- Metoda `__next__()` vrne trenutno stanje števca ter ga poveča. Če preseže zgornjo mejo, vrže izjemo `StopIteration`.
- Metoda `__iter__()` vrne kot iterator kar sam objekt (tako ga lahko uporabimo v for zanki).
- Slabost takšnih iteratorjev: razred, ne moremo iterirati v večih vzporednih iteracijah, saj obstaja le eno stanje iteracije (v razredu)

Drugi primeri uporabe iteratorjev

- Nizi, množice, slovarji
- Datoteke
- Sprotni viri, npr. iz spleta
- Neskončni seznami

```
sl = {"a": 1, "b": 2}
for k in sl: print(k, sl[k])
```

```
for vrstica in open("datoteka.txt"):
    predelaj(vrstica)
```

```
class Naravna:
    def __init__(self):
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        t = self.i
        self.i += 1
        return t

it = Naravna()
while True:
    n = next(it)
    predelaj(n)
```

Generatorji

- Če v definiciji funkcije uporabimo ukaz `yield` namesto kakega ukaza `return`, ustvarimo generator.
- Klic funkcije-generatorja vrne objekt generatorja, ki je iterabilen objekt
- Ta ob vsakem klicu funkcije `next` nad objektom generatorja (metode `__next__()`) vrne naslednji člen zaporedja, ki ga določa.
- Za konec zaporedja vržemo izjemo `StopIteration` oz. Kličemo `return`

Primer – generator kvadratov

- Klic funkcije ustvari generatorski objekt

```
def kvadrati(n):
    i = 0
    while True:
        yield i*i
        i += 1
        if i >= n: raise StopIteration
```

```
>>> for i in kvadrati(3): print(i)

0
1
4
```

```
>>> kva = kvadrati(3)
>>> kva
<generator object kvadrati at 0x02F0EB70>
>>> next(kva)
0
>>> next(kva)
1
>>> next(kva)
4
>>> next(kva)
Traceback (most recent call last):
  File "<pyshell#134>", line 1, in <module>
    next(kva)
  File "<pyshell#126>", line 6, in kvadrati
    if i >= n: raise StopIteration
StopIteration
```

Generatorski izraz

- Izpeljani seznami (seznamski izraz)

```
>>> s = [1, 2, 3]
>>> k1 = [i**2 for i in s]
>>> k1
[1, 4, 9]
```

- Generatorski izraz

```
>>> k2 = (i**2 for i in s)
>>> k2
<generator object <genexpr> at 0x02F16BC0>
>>> for i in k2: print(i)

1
4
9
```

Samoglasniki v nizu

- Generator, ki iterira po vseh samoglasnikih v nizu, ne glede na velikost črk

```
def samoglasniki(niz):
    for z in niz:
        if z.lower() in "aeiou":
            yield z

niz = "abeCE DA 12tRI"

>>> for z in samoglasniki(niz): print(z)

a
e
E
A
I
```

Besedni števec

- V datoteki imamo v vsaki vrstici eno besedo.
- Sestavimo generator, ki iterira po teh besedah

stej.txt

```
ena
dva
tri
štiri
pet
šest
sedem
osem
devet
deset
```

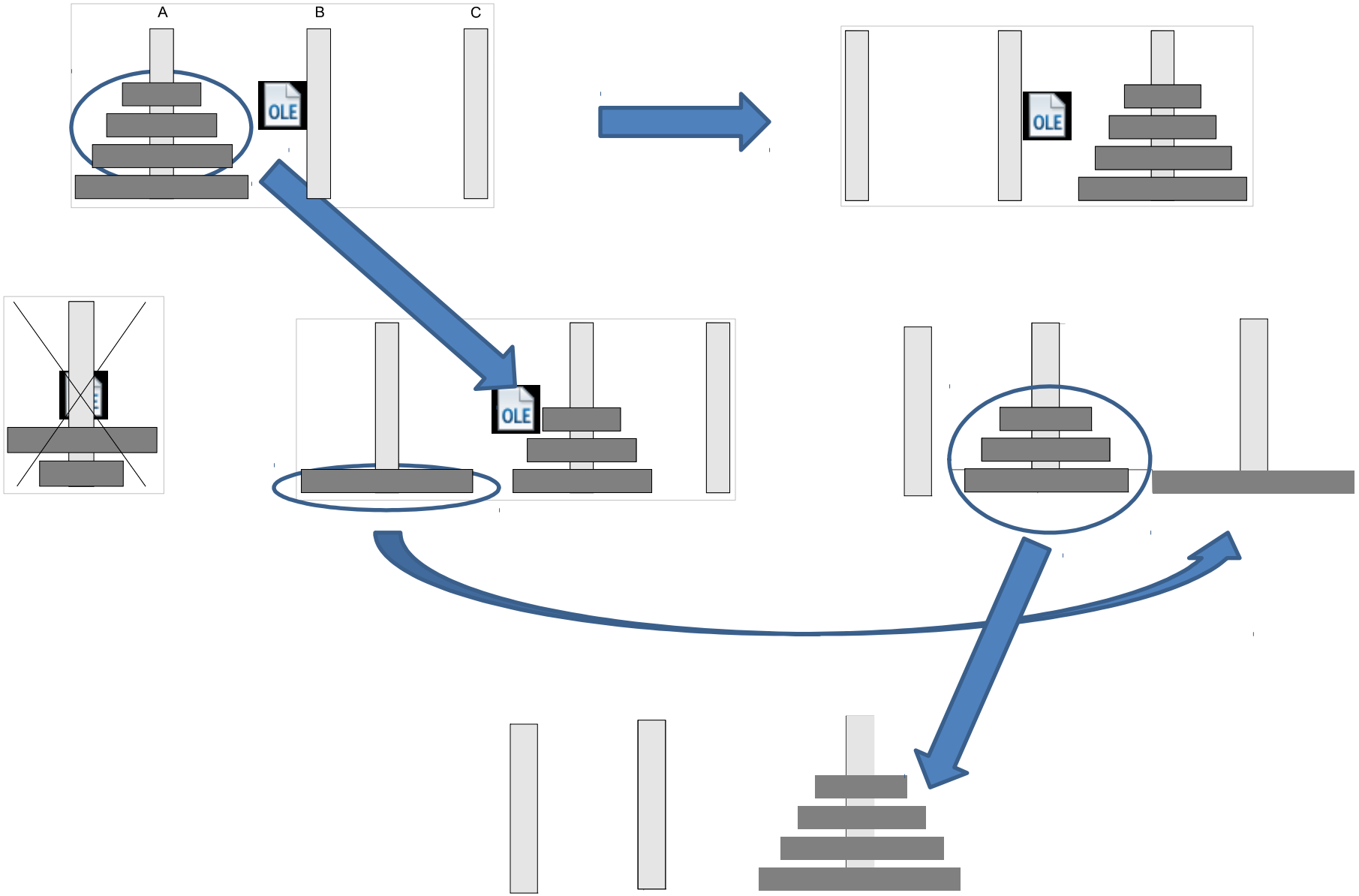
```
def stej():
    with open("stej.txt") as f:
        for v in f:
            yield v.strip()
```

```
>>> for b in stej(): print(b)

ena
dva
tri
štiri
pet
šest
sedem
osem
devet
deset
```

Ostali primeri

- Fibonaccijeva števila
- Števila v datoteki
- Generator podmnožic
- Hanoiski stolpiči


A
B
C
OLE
OLE
OLE