<-- vissza

Tartalomjegyzék

Függvényközpontú (funkcionális) programozás python-ban

Programozási paradigmák

Egy probléma megoldásához többféleképp közelíthetünk. Az, hogy a problémát hogyan bontjuk részekre, meghatározza a használható/andó programnyelvi elemeket. Ennek megfelel?en a programnyelvi elemeknek négy f? típusát szokás megkülönböztetni:

- **Parancsközpontú** (imperative), vagy vele majdnem azonos értelemben használt **eljárásközpontú** (procedurális, procedural)
  - algoritmikus gondolkodás, receptszer? programozás, „tedd ezt, majd ezt,..."http://wiki.math.bme.hu
  - A Neumann-elv? számítógép ötletére épül
  - Számítási lépések id?ben egymás utáni elvégzése (parancsok végrehajtása)
  - Tipikus parancsok: értékadás, eljárás meghívása,
  - Az eljárások megváltoztatják a program kontrollstruktúráinak állapotát
  - Pl. C, Fortran, Algol, Pascal, Basic
- **Objektumközpontú** (objektumorientált, object-oriented)
  - Az emberek közti interakciókat és a való világ jelenségeit modellezi
  - A fogalmakat osztályok, a jelenségeket objektumok valósítják meg
  - Pl. Java, C#,
- **Függvényközpontú** (funkcionális, functional)
  - A függvények matematikai elméletére épül,
  - A függvények értéket adnak vissza, de nincs mellékhatásuk, nincsenek változók
  - A program függvényhívásokból áll
  - A parancsközpontúnál egyszer?bb struktúra, könnyebben bizonyítható a program helyessége, könnyebben tesztelhet?, debug-olható
  - Pl. ML, Haskell, Lisp, Scheme,
- **Logikai**, vagy a nála általánosabb értelemben használt **deklaratív** (logic, declarative)
  - A program végrehajtása az axiómáknak és a következtetési szabályoknak (deklarációknak) megfelel? lehet?ség(ek) megkeresésb?l áll
  - Matematikai logikai alapú
  - Sz?kebb alkalmazási lehet?ségek
  - Pl. Prolog, SQL

A függvényközpontú programozás klasszikus elemei a Pythonban

A `lambda` kifejezések névtelen függvények, alakjuk:

```
lambda paraméterek: kifejezés
```

ahol a `paraméterek` egy esetleg üres lista, a `kifejezés` pedig egy elágazást, ciklust, `return` vagy `yield` utasítást nem tartalmazó (de esetleg feltételes) kifejezés lehet. El?nyük, hogy csak kifejezések, ezért más kifejezésekben is szerepelhetnek, és ha nem szükséges, nem kell nevet adni nekik. Csak nagyon egyszer? (félsoros) függvényekre használjuk!

```
lambda x: x ** 2
lambda x: "pozitív" if x > 0 else "nem pozitív"
lambda x, y: x + y
```

A ciklusok elkerülésére három egyszer? „klasszikus *függvény szolgál:*

- megfeleltetés -- egy függvény egy bejárható (iterable) objektum minden elemére hat: `map()`

```
map(lambda x: x ** 2, [1, 2, 3, 4])  # [1, 4, 9, 16]
map(lambda x, y: x + y, [1, 2, 3, 4], [0, 10, 100, 1000])
                                # [1, 12, 103, 1004]
```

- sz?rés -- egy bejárható objektumnak csak azokat az elemeit tartjuk meg, amelyre egy függvény `True` értéket ad: `filter()`

```
filter(lambda x: x > 0, [-1, 2, -3, 4])  # [2, 4]
```

- redukálás: `reduce()` (3.0-tól a standard könyvtárból törölve: `functools.reduce()`)

```
reduce(lambda x, y: x + y, [1, 2, 3, 4])  # 10
```

Az üres listára hibaüzenetet kapunk. Ez ellen véd a harmadik argumentum, ahol a kezd?érték megadható:

```
reduce(lambda x, y: x + y, [1, 2, 3, 4], 0)  # 10
reduce(lambda x, y: x * y, [1, 2, 3, 4], 1)  # 24
```

Listaértelmezések (list comprehensions) és generátorkifejezések

Megfeleltetés és sz?rés másként

A fenti három függvény *helyettesítésére* szolgál a lista elemeinek a matematikában szokásos halmazmegadási módjára emlékeztet? leírása, amelyben ciklus-szer? és feltétel-szer? nyelvi elem is szerepelhet. Alakjuk azonnal érthet?:

```
[kifejezés for elem in bejárható_objektum]
[kifejezés for elem in bejárható_objektum if feltétel]
[kifejezés for elem1 in bejárható_objektum1 if feltétel1
           for elem2 in bejárható_objektum2 if feltétel2
           for elemN in bejárható_objektumN if feltételN]
```

Az utóbbi listaértelmezés a következ? kóddal ekvivalens:

```
for elem1 in bejárható_objektum1:
    if not (feltétel1):
        continue
```

```
    for elem2 in bejárható_objektum2:
        if not (feltétel2):
            continue
        for elemN in bejárható_objektumN:
            if not (feltételN):
                continue
            kifejezés
```

Megadjuk a megfeleltetés és sz?rés fenti els? két példája e szintaktika szerinti változatát:

```
[x ** 2 for x in [1, 2, 3, 4]]  # [1, 4, 9, 16]
[x for x in [-1, 2, -3, 4] if x > 0]
```

A harmadik esetre Guido van Rossum azt mondja, hogy tipikusan csak asszociatív függvényekre használjuk, mint + vagy *.

```
sum([1, 2, 3, 4])   # 10
```

A Google is azt javasolja, hogy map, filter, reduce helyett használjunk listaértelmezéseket vagy for ciklust, jobban olvasható.

Az egymásba ágyazott ciklusok is helyettesíthet?ek listaértelmezéssel.

```
for x in (1,2,3):
    for y in [1,2,3,4]:
        if x < y:
            print (x, y, x*y),
# (1, 2, 2) (1, 3, 3) (1, 4, 4) (2, 3, 6) (2, 4, 8) (3, 4, 12)
```

E helyett írható a következ? kód:

```
[(x, y, x * y) for x in (1,2,3) for y in [1,2,3,4] if x < y]
# [(1, 2, 2), (1, 3, 3), (1, 4, 4), (2, 3, 6), (2, 4, 8), (3, 4, 12)]
```

Egy példa: a nyolc királyn?

Feladat: Hányféleképp lehet nyolc királyn?t föltenni egy sakktáblára, hogy semelyik kett? ne üsse egymást?

Azt tudjuk, hogy az ilyen bástyaelhelyezések száma 8! = 40320. A királyn?k azonban átlósan is üthetik egymást.

Az i-edik és j-edik sorban lév? királyn? pontosan akkor üti egymást, ha `abs(i - j) != abs(x[i] - x[j])`. Így a megoldás:

```
from itertools import permutations
for x in permutations(range(8)):
    if all([i == j or abs(i - j) != abs(x[i] - x[j])
        for i in range(8) for j in range(8)]):
        print x
```

Generátorkifejezések

A listaértelmezésekhez hasonló szintaktikával (szögletes helyett kerek zárójelekkel) generátorkifejezés is konstruálható. Ez tehát a next() függvény meghívásakor számítja ki a következ? elemet. Szintaktikája:

```
(kifejezés for elem in bejárható_objektum)
(kifejezés for elem in bejárható_objektum if feltétel)
(kifejezés for elem in bejárható_objektum1 if feltétel1
           for elem in bejárható_objektum2 if feltétel2
           for elem in bejárható_objektumN if feltételN)
```

A alábbi generátorkifejezés mindig a következ? négyzetszámot adja:

```
>>> y = (x**2 for x in range(10))
>>> print y
<generator object <genexpr> at 0xb740bb6c>
>>> y.next()
0
>>> y.next()
1
>>> y.next()
4
>>> y.next()
9
.......
>>> y.next()
64
>>> y.next()
81
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

A következ? két program ekvivalens, az egyikben generátorkifejezéssel, a másodikban yield segítségével adunk meg olyan függvényt, mely az els? n négyzetszámot generáló függvényt ad vissza.

```
>>> def kovetkezo_negyzet(n):
...     for x in range(n):
...         yield x ** 2
...
>>> y = kovetkezo_negyzet(3)
>>> print y
<generator object kovetkezo_negyzet at 0xb740ba7c>
>>> y.next()
0
>>> y.next()
1
>>> y.next()
4
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

A második változat generátorkifejezéssel:

```
>>> def kovetkezo_negyzet(n):
...     return (x ** 2 for x in range(n))
...
>>> y = kovetkezo_negyzet(3)
>>> print y
<generator object <genexpr> at 0xb740baa4>
>>> y.next()
0
>>> y.next()
1
```

```
>>> y.next()
4
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterátorok (bejárók) és bejárható adattípusok

Bejárható (iterálható, iterable) az az adattípus, melynek elemei egyesével megkaphatók. Minden objektum bejárható, melynek van __iter__() nev? metódusa (tagfüggvénye), és bejárható minden sorozat (melynek van __getitem__() metódusa). A bejárható objektumból **bejáró (iterátor)** képezhet?, ezek rendelkeznek __next__() metódussal, mely a következ? elemet adja, és végül StopIteration kivételt dob. Egy példa for ciklussal:

```
szorzat = 1
for i in [1, 2, 3, 4]:
    szorzat *= i
print szorzat    # 24
```

Ekvivalens kód iterátorral:

```
szorzat = 1
i = iter([1, 2, 3, 4])
while True:
    try:
        szorzat *= next(i)
    except StopIteration:
        break
print szorzat    # 24
```

Az enumerate() függvény egy bejárható objektumot kap, második opcionális argumentuma pedig egy kezd?érték, mely alapértelmezésben 0. Visszaadja a bejárható objektum sorszámmal ellátott elemeib?l álló párokat.

```
e = enumerate([1,2,4])
for i in e:
    print i
#(0, 1)
#(1, 2)
#(2, 4)
```

A következ? kódban egy fájlból kiírjuk az üres sorok sorszámait:

```
f = open('data.txt', 'r')
for i, sor in enumerate(f, 1):
    if sor.strip() == '':
        print 'A(z) %i. sor üres' % i
```

Az függvényközpontú programozást segít? modulok

itertools, functools, operator