

Funkcije

Vsem, ki že znate programirati in torej že znate napisati funkcijo v kateremkoli jeziku, je potrebno pokazati samo primer in vse bo jasno.

```
def delitelji(n):  
    s = []  
    for i in range(1, n):  
        if n % i == 0:  
            s.append(i)  
    return s
```

To je vse, kar nas zanima od "rednega predavanja".

Zdaj pa detajli v zvezi s funkcijami, s katerimi se na rednih predavanjih skoraj (ali sploh) ne bomo ukvarjali.

Generiki in račje tipiziranje

Vsaka funkcija v Pythonu je generik (ali, po C++-ovsko, template), ne da bi morali to kaj posebej definirati. Točneje, skoraj ne obstaja način, da bi to preprečili. Tako je zato, ker v Pythonu ne definiramo tipov spremenljivk in prav tako ne tipov argumentov ali rezultatov.

Za tiste, ki izraza generik ali template ne poznate: gre za to, da lahko ena in ista funkcija sprejema argumente različnih tipov.

```
def sestej(a, b):  
    return a + b
```

```
sestej(1, 5)
```

```
6
```

```
sestej("Ana", "marija")
```

```
'Anamarija'
```

```
sestej("Ana", 5)
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-5-2c6500e8edaf> in <module>  
----> 1 sestej("Ana", 5)  
  
<ipython-input-2-7e63af8aa86d> in sestej(a, b)  
      1 def sestej(a, b):  
----> 2     return a + b
```

```
TypeError: can only concatenate str (not "int") to str
```

Zadnja funkcija ne deluje, je pa tole lep trenutek za to, da izvemo, kako ne deluje. Ko funkcijo pokličemo s `sestěj(1, 5)`, imamo, v bistvu

```
a = 1
b = 5
```

Izraz `a + b` se izvede tako, da Python pokliče metodo `a.__add__` in poda `b` kot argument.

```
a.__add__(b)
```

6

Metoda `a.__add__` vrne vsoto "sebe" in argumenta.

Podobno je z nizi.

```
a = "Ana"
b = "marija"
a.__add__(b)
```

```
'Anamarija'
```

V zadnjem primeru, `sestěj("Ana", 5)` pa imamo

```
a = "Ana"
b = 5
a.__add__(b)
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-9-24945ae2d4a7> in <module>
      1 a = "Ana"
      2 b = 5
----> 3 a.__add__(b)
```

`TypeError: can only concatenate str (not "int") to str`

Python se torej preprosto ne vznemirja zaradi tipov. Vsaka funkcija se vede kot generik: podamo ji lahko poljubne argumente, funkciji je vseeno, Pythonu je vseeno ... če stvari ne gredo skupaj, pa bo prišlo do napake, ko nekdo nečesa ne bo mogel narediti. Se pravi, ko si bo moral niz prišteti število, ko bomo poskušali indeksirati `float` (v tem primeru bo Python poklical `float`-ovo metodo `__item__` in se bo izkazalo, da le-ta te metode nima) ...

Temu se reče *duck typing*: če nek objekt hodi kot raca in gaga kot raca, potem je raca. Vsaj za *all intents and purposes*.

Funkcije lahko pišemo z mislijo na to.

```
def vsota(s):
    v = 0
    for x in s:
```

```

        v += x
    return v

```

Tej funkciji lahko podamo kot argument stvari, prek katerih je možno nagnati zanko `for` (niz, seznam, terka, `range`, slovar, množica, datoteka...) in katere elementi so stvari, ki jih je možno seštevati. No, skoraj tako.

```
vsota({1, 2, 3})
```

```
6
```

```
vsota(range(5))
```

```
10
```

```
vsota(["Ana", "Berta", "Cilka"])
```

```

-----
TypeError                                Traceback (most recent call last)
<ipython-input-13-3951a9717356> in <module>
----> 1 vsota(["Ana", "Berta", "Cilka"])

<ipython-input-10-0f50951fc0c7> in vsota(s)
      2     v = 0
      3     for x in s:
----> 4         v += x
      5     return v

```

`TypeError: unsupported operand type(s) for +=: 'int' and 'str'`

To ne deluje, ker v začetku nastavimo `v = 0`, potem pa poskušamo `v += "Ana"`. Naša funkcija `vsota` torej ne zahteva le, da le, da se dajo elementi seštevati, temveč tudi, da se dajo prišteti k 0. Vsaj prvi od teh elementov.

Prva rešitev bi bila

```

def vsota(s):
    v = s[0]
    for x in s[1:]:
        v += x
    return v

```

Zdaj deluje

```
vsota(["Ana", "Berta", "Cilka"])
```

```
'AnaBertaCilka'
```

in celo

```
vsota("Benjamin")
```

```
'Benjamin'
```

ker gre pač z zanko čez niz in seštevaja njene črke. Vendar naša funkcija zdaj zahteva, da lahko dobimo prvi element `s`-a in da lahko naredimo rezino, ki vsebuje vse elemente, razen prvega. Zato je nehalo delovati seštevanje množic.

```
vsota({1, 2, 3})
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-17-5b638e840346> in <module>
----> 1 vsota({1, 2, 3})
```

```
<ipython-input-14-1e091943e7c7> in vsota(s)
      1 def vsota(s):
----> 2     v = s[0]
      3     for x in s[1:]:
      4         v += x
      5     return v
```

```
TypeError: 'set' object is not subscriptable
```

Najbolj splošna oblika je

```
def vsota(s):
    v = None
    for e in s:
        if v == None:
            v = e
        else:
            v += e
    return v
```

Lahko bi naredili tudi nekoliko drugače, uporabili bi lahko iteratorje. A to se bomo naučili kdaj drugič.

Več ali manj kot en rezultat

V Pythonu vse funkcije vračajo rezultat. Nobenih `void` funkcij. In to natančno en rezultat. Ne dveh. Nekaj primerov.

```
def ne_vrne_nicesar():
    x = 42
```

```
def vrne_odgovor():
    return 42
```

```
def vrne_dva():
    return "Koliko je 6x8?", 42
```

Najprej pokličimo funkcijo, ki ne vrne ničesar.

```
nic = ne_vrne_nicesar()
nic
```

No, vidite, nič se ne izpiše. :)

To je zato, ker Jupyter oz. IPython oz. Python v konzoli izpiše `None` le, če eksplicitno pokličemo `print`.

```
print(nic)
```

```
None
```

Če torej funkcija ne vrne ničesar, vrne `None`.

Druge funkcije nima smisla klicati. Pač pa bomo tretjo.

```
vprašanje, odgovor = vrne_dva()
```

```
vprašanje
```

```
'Koliko je 6x8?'
```

```
odgovor
```

```
42
```

Tudi ta funkcija je vrnila le eno stvar. Terko ("Koliko je 6x8?", 42). Le da smo jo napisali brez oklepajev. Kadar funkcija navidez vrača več stvari, napišemo `return` tako, da je videz pravi - kot da vračamo več stvari torej. In tudi kličemo jo, kot da bi vračala več stvari. V resnici pa gre za igro terk.

`None` je prikladna vrednost, ki lahko pove, da funkcija ni mogla vrniti ničesar pametnega.

```
def prvo_sodo(s):
    for e in s:
        if e % 2 == 0:
            return e
```

```
prvo_sodo([1, 2, 3, 4, 5])
```

```
2
```

```
prvo_sodo([1, 3, 5])
```

Tu je potrebno biti previden. Najprej: da gornja funkcija vrne `None`, če ni sodih števil, je praktično "stranski učinek". Tako raje ne pišemo. Lint bi nas na to opozoril. Kadar funkcija včasih vrne rezultat, včasih pa vrne `None`, pa je vračanje `None` praviloma eksplicitno. Sicer bi lahko kdo, ki bere funkcijo, mislil, da v nekaterih primerih ne vrne ničesar, ker je programer pozabil na `return`. Lepo napisana funkcija `prvo_sodo` je torej

```
def prvo_sodo(s):
    for e in s:
        if e % 2 == 0:
```

```

        return e
    return None

```

Drugo opozorilo: ne zlorablajmo `None`-a. Takšne funkcije nas lahko vodijo v prikrite napake. Če bi takole napisali funkcijo `prvo_sodo` in `prvo_liho`, bi lahko kdo ponevedoma takole računal vsoto prvega lihega in prvega sodega števila.

```

def prvo_liho(s):
    for e in s:
        if e % 2 == 1:
            return e
    return None

```

```

s = [1, 3, 5, 7]
prvo_sodo(s) + prvo_liho(s)

```

```

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-29-da993963350e> in <module>
      6
      7 s = [1, 3, 5, 7]
----> 8 prvo_sodo(s) + prvo_liho(s)

```

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

Nullable tipi

Dve možnosti imamo.

- Lahko se zmenimo, da je tole v redu. V dokumentaciji funkcije mora jasno pisati, da bo vrnila `None`, če v seznamu ni sodega oz. lihega števila. Vsak, ki pokliče to funkcijo, je odgovoren, da popazi na rezultat. Če pozabi - je to njegov problem.

Problem pri tem principu je, da se napaka ne zgodi nujno takoj temveč precej pozneje. Recimo, da kličemo `prvo_sodo` in rezultat shranimo - bogve kam. Nekje precej kasneje, v čisto drugem koncu programa, preberemo to vrednost, tam bi morala biti številka ... naletimo pa na `None`. In program se sesuje nekje tam. Zdaj pa najdi to napako, če jo moreš. Zato velja, da se morajo napake zgoditi čim prej in ne čim kasneje.

- Lahko se torej zmenimo, naj funkcija vrže exception. V dokumentaciji funkcije mora jasno pisati, da bo vrgla exception, če v seznamu ni sodega oz. lihega števila. Vsak, ki pokliče to funkcijo, je odgovoren, da popazi na argumente. Če pozabi - je to njegov problem.

Očitno smo v obeh primerih na približno istem. To je tudi razlog, da sem približno skopiral odstavek.

To nima zveze s Pythonovim dinamičnim tipiziranjem in še manj z generiki, to je problem mnogih jezikov.

Novejši popularni jeziki (se učijo od starejših bolj teoretičnih jezikov in) to rešujejo z "option"-i ali z "nullable tipi". V Kotlinu bi takšno funkcijo definirali z

```
fun firstEven(s: List[Int]) -> Int?  
    ... in tako naprej
```

Tip rezultata `Int?` pomeni, da bo funkcija vrnila `Int` ... ali pa ne. (V nekaterih jezikih se temu reče "maybe int", "mogoče int"). Tako kot Pythonova funkcija vrne `int` ali pa bo vrnila `None`. Če potem poskusimo sešteti

```
firstEven(s) + firstOdd(s)
```

nam tega ne bo pustil, ker rezultata teh dveh funkcij nista `Int`, temveč sta samo "mogoče `Int`". Tudi to ne gre.

```
const a: Int = firstEven(s)
```

Rezultata funkcije `firstEven` ne moremo prirediti spremenljivki tipa `Int` - in se potem delati, da obstaja. Biti mora `Int?`:

```
const a: Int? = firstEven(s)
```

Kako se znebimo tega "mogoče"? V Kotlinu tako, da priskrbimo privzeto vrednost ali pa preverimo, da vrednost ni `null`. Operator `a ?: b` vrne `a`, če le-ta ni `null`. Če je, pa vrne `b`. Recimo, da bi v primeru, da prvega lihega števila ni, želeli uporabiti kar 1. To naredimo tako: `firstEven(s) ?: 1`.