04a funkcije

January 28, 2024

1 Funkcije

Funkcije smo si doslej predstavljali kot škatlice: nekaj gre noter (temu smo in bomo rekli *argument*), nekaj pride ven (temu se reče rezultat funkcije), vmes pa se lahko še kaj opaznega dogaja, recimo izpisuje. Primer funkcije, ki je počela vse to, je `input`: kot argument smo ji povedali, kaj naj vpraša uporabnika; kot rezultat je vrnila, kar je vtipkal uporabnik; vmes se je zgodilo to, da je funkcija nekaj vprašala uporabnika in počakala, da je le-ta odgovoril. Druga funkcija, ki smo jo srečali, je bila `sqrt`, ki dobi kot argument neko število in vrne njegov koren. Vmes se ne dogaja nič opaznega, funkcija le "neopazno" naredi, kar mora narediti.

S tem, kako delujejo funkcije in kako kaj naredijo, se doslej nismo ukvarjali. Te stvari so za nas napisali drugi (hvala, hvala) in mi jih lahko uporabljamo, ne da bi nas vznemirjalo vprašanje, kako so napisane. S tem, kako so napisane funkcije, ki so jih naredili drugi, se tudi v prihodnje ne bomo ukvarjali. Pač pa se bomo danes naučili pisati svoje.

1.1 Popolna števila

Število je *popolno*, če je enako vsoti svojih deliteljev. 28 je deljivo z 1, 2, 4, 7 in 14 ter je popolno, saj je 1+2+4+7+14 ravno 28. Napišimo program, ki sestavi seznam vseh popolnih števil do 1000.

1.1.1 Delitelji števila

Znamo napisati program, ki sestavi seznam vseh deliteljev nekega števila n?

[1]:
```
n = int(input("Vnesi število: "))
s = []
for i in range(1, n):
    if n % i == 0:
        s.append(i)
print(s)
```

```
Vnesi število: 42
[1, 2, 3, 6, 7, 14, 21]
```

Najprej naredimo prazen seznam, nato gremo prek vseh števil od 1 do n in če število deli `n`, ga dodamo v `s`.

Kaj ne bi bilo lepo, če bi imel Python kar funkcijo `delitelji`, ki bi jo lahko uporabili? Potem bi lahko napisali kar

[2]:
```
stevilka = int(input("Vnesi število: "))
s = delitelji(stevilka)
print(s)
```

```
Vnesi število: 42
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-fc5d3539f150> in <module>
      1 stevilka = int(input("Vnesi število: "))
----> 2 s = delitelji(stevilka)
      3 print(s)

NameError: name 'delitelji' is not defined
```

Kot je povsem pravilno predlagal profesor Franc Oblak: *česar ni, se pa nardi* (ter v sredo polinoma, ki se ga na noben način ni dalo razcepiti, dodal "$+x - x$" in polinom je šel na faktorje kot toplo maslo). Napišimo si takšno funkcijo, da jo bomo lahko klicali.

[3]:
```
def delitelji(n):
    s = []
    for i in range(1, n):
        if n % i == 0:
            s.append(i)
    return s
```

Definicijo funkcije začnemo z `def`; to je rezervirana beseda, ki pomeni, da to, kar sledi, ni "program, ki ga je treba takoj izvesti", temveč funkcija. Z drugimi besedami, `def delitelji` pomeni: "*kadar bo kdo poklical funkcijo* ***`delitelji`****, naredi naslednje:*".

Imenu sledijo oklepaji, v katerih navedemo *imena argumentov funkcije*. V našem primeru bo funkcija zahtevala en argument. Torej, ta, ki bo poklical funkcijo, bo moral v oklepaje napisati eno reč (upamo, da bo napisal število, sicer pa naj si sam pripiše posledice).

Tista reč, ki jo bomo ob klicu funkcije podali kot argument, se bo znotraj funkcije pojavila kot spremenljivka z imenom `n`. Takšno ime smo namreč uporabili v prvi vrstici, v "glavi" funkcije. Vrednosti ji ne bomo priredili: ko bo nekdo poklical funkcijo, bo Python tej "spremenljivki" kar sam od sebe priredil vrednost, ki jo bo "klicatelj" napisal kot argument funkcije. Če torej nekdo pokliče

[4]:
```
delitelji(35)
```

[4]:
```
[1, 5, 7]
```

bo imel `n` vrednost 35 in če pokliče

[5]:
```
delitelji(13)
```

[5]: [1]

`n` ima vrednost argumenta.

Za `def delitelji(n)` sledi dvopičje in zamik. Vse, kar sledi takole zamaknjeno, je koda funkcije. Kaj mora narediti le-ta? No, tisto, kar pač dela funkcija: sestaviti seznam deliteljev `n`. To pa ne le znamo narediti, temveč smo celo ravno prejle tudi zares naredili in lahko le skopiramo.

Na koncu (ali tudi že kje vmes - bomo že videli primer) funkcija pove, kaj naj klicatelj dobi kot rezultat. To stori s stavkom `return s`.

1.1.2 Geometrija

Napišimo funkcijo, ki dobi kot argument dolžine stranic trikotnika in vrne njegovo ploščino. Ta funkcija bo imela tri argumente; poimenujmo jih `a`, `b` in `c`.

[6]:
```
from math import *
def ploscina_trikotnika(a, b, c):
    s = (a + b + c) / 2
    return sqrt(s * (s - a) * (s - b) * (s - c))
```

Ko smo pri tem, napišimo še funkcijo funkcijo za obseg trikotnika ter za obseg in ploščino kroga.

[7]:
```
from math import *

def obseg_trikotnika(a, b, c):
    return a + b + c

def ploscina_kroga(r):
    return pi * r ** 2

def obseg_kroga(r):
    return 2 * pi * r
```

1.1.3 Vsota števil v seznamu

Zdaj napišimo drugo funkcijo: funkcijo, ki dobi seznam števil in izračuna njihovo vsoto.

[8]:
```
def vsota(s):
    vsota = 0
    for e in s:
        vsota += e
    return vsota
```

Funkcija ima spet en argument, tokrat smo ga poimenovali `s`. Ta argument bo seznam števil, ki jih je potrebno sešteti. Funkcija gre - z zanko `for` - prek tega seznama in sešteva, kot smo počeli že prejšnji teden, le brez funkcij. Na koncu rezultata ne izpiše (`print`), kot smo delali doslej, temveč ga vrne (`return`).

(Mimogrede povejmo še, da nam funkcije `vsota` ne bi bilo potrebno napisati, saj obstaja: imenuje se `sum`.)

Za napredno misleče Nekateri se sprašujejo, kako funkcija ve, kakšnega tipa bodo argumenti, ki jih bo dobila. Nekateri se zdaj, ko so izvedeli, da se nekateri sprašujejo o tem, sprašujejo, zakaj bi se kdo to spraševal. Vsebina tega razdelka je namenjena predvsem prvim.

V nekaterih jezikih je potrebno za vsako spremenljivko, preden jo uporabimo, povedati, kakšnega tipa bo. Python ni eden izmed njih. Podobno je z argumenti funkcij. V nekaterih jezikih bi morali povedati, kakšnega tipa bodo argumenti funkcije in kakšen bo rezultat. Tudi takšen Python ni. Funkcija sprejme, kar sprejme in vrača, kar vrača.

Za primer vzemimo preprostejšo funkcijo `sestej`, ki ji podamo dva argumenta, funkcija pa vrne njuno vsoto.

[9]:
```
def sestej(a, b):
    return a + b
```

Preskusimo jo.

[10]:
```
sestej(1, 5)
```

[10]:
```
6
```

[11]:
```
sestej("Ana", "marija")
```

[11]:
```
'Anamarija'
```

[12]:
```
sestej(1, "Ana")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-27cd999b26a0> in <module>
----> 1 sestej(1, "Ana")

<ipython-input-9-7e63af8aa86d> in sestej(a, b)
      1 def sestej(a, b):
----> 2     return a + b

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

Funkcija se ne ukvarja s tipom argumentov. Pač pa se s tem ukvarja `+`. Če ga pokličemo s čim, česar se ne da seštevati, javi napako.

V resnici je stvar še lepša. Ko Python pride do `a + b`, reče objektoma `a` in `b`, naj se seštejeta. Če se znata, je to v redu, če ne, pa javita napako in Python jo izpiše. Python v resnici ne zna seštevati, seštevati (se) znajo objekti. Operator `+` ne obstaja.

Vrnimo se k funkciji `vsota`, oni od prej. Kaj bi se zgodilo, če bi dali funkciji namesto seznama kaj drugega, recimo terko ali kaj podobnega? No, preskusimo jo.

[13]:
```
vsota([1, 2, 3])
```

[13]:
```
6
```

[14]:
```
vsota((1, 2, 3))
```

[14]:
```
6
```

[15]:
```
vsota(range(4))
```

[15]:
```
6
```

S seznamom dela. Prvi klic pomeni isto, kot če bi napisali:

[16]:
```
v = 0
for e in [1, 5, 3]:
    v += e
```

Drugi klic je podoben, le da je `s` zdaj terka `(1, 5, 3)`, torej dobimo

[17]:
```
v = 0
for e in (1, 5, 2):
    v += e
```

Tretji klic pa je isto, kot če bi napisali

[18]:
```
v = 0
for e in range(4):
    v += e
```

Kaj pa, če bi namesto seznama celih števil podali seznam necelih?

[19]:
```
vsota([1.3, 2.2, 3.1])
```

[19]:
```
6.6
```

Dela, jasno. Zakaj pa ne bi?

Kaj pa, če bi dali seznam nizov? Poglejmo, poučno bo.

[20]:
```
vsota(["Ana", "Berta", "Cilka"])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-3951a9717356> in <module>
----> 1 vsota(["Ana", "Berta", "Cilka"])
```

```
<ipython-input-8-829f2145b967> in vsota(s)
      2     vsota = 0
      3     for e in s:
----> 4         vsota += e
      5     return vsota

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

Zgodi se tole. Najprej je `v` enak 0. V prvem koraku zanke poskuša izračunati `0 + "Ana"`, kar se ne da.

Funkcijo je mogoče popraviti, da bo delovala tudi za nize.

[21]:
```
def vsota(s):
    v = None
    for e in s:
        if v == None:
            v = e
        else:
            v += e
    return v
```

`v`-ju moramo v začetku dati neko vrednost, ki bo povedala, da še nima nobene vrednosti. Tipično delo za `None`. V zanki preverimo, ali je `v` še vedno `None` in mu v tem primeru *priredimo* `e`, sicer pa mu *prištejemo* `e`.

Zdaj bo `v` gotovo pravega tipa in funkcija bo znala seštevati vse živo.

[22]:
```
vsota([1, 2, 3])
```

[22]:
```
6
```

[23]:
```
vsota(range(4))
```

[23]:
```
6
```

[24]:
```
vsota(["Ana", "Berta", "Cilka"])
```

[24]:
```
'AnaBertaCilka'
```

Kaj pa tole?

[25]:
```
vsota("Benjamin")
```

[25]:
```
'Benjamin'
```

Zanka `for` gre prek niza in sešteje njegove črke. Rezultat je, seveda, spet isti niz.

Pač pa funkcija ne deluje, če ji damo seznam, ki vsebuje reči, ki jih ni mogoče sešteti.

[26]:
```
vsota([1, "a"])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-cb5caf960439> in <module>
----> 1 vsota([1, "a"])

<ipython-input-21-31fd070ab5a0> in vsota(s)
      5             v = e
      6         else:
----> 7             v += e
      8     return v

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

1.1.4 Popolno število

Vrnimo se k popolnim številom. Rekli smo, da je število popolno, če je enako vsoti svojih deliteljev. Lahko bi torej rekli

[28]:
```
stevilo = int(input("Vnesi število: "))
d = delitelji(stevilo)
v = vsota(d)
if v == stevilo:
    print("Število", stevilo, "je popolno")
else:
    print("Število", stevilo, "ni popolno")
```

```
Vnesi število: 28
Število 28 je popolno
```

vendar ne bomo. Napisali bomo funkcijo, ki vrne `True`, če je število popolno in `False`, če ni.

[29]:
```
def popolno(n):
    d = delitelji(n)
    v = vsota(d)
    return v == n
```

Pazite, tule nismo pisali

[30]:
```
def popolno(n):
    d = delitelji(n)
    v = vsota(d)
    if v == n:
        return True
    else:
        return False
```

Lahko bi, vendar bi bilo smešno. Pač pa lahko funkcijo še skrajšamo (in navadno bi jo tudi res), takole

[31]:
```
def popolno(n):
    return vsota(delitelji(n)) == n
```

Končno ostane še program, ki bo z uporabo gornje funkcije sestavil seznam vseh popolnih števil manjših od 1000. V njem z zanko `for` preštejemo do 1000, za vsako število posebej preverimo, ali je popolno in če je, ga dodamo na seznam.

[32]:
```
s = []
for i in range(1, 1001):
    if popolno(i):
        s.append(i)
print(s)
```

```
[6, 28, 496]
```

Lepo prosim, ne pišite `if popolno(i) == True:`. Smo že povedali, zakaj, ne?

1.2 Klici funkcij

Najprej zberimo vse skupaj:

[33]:
```
def delitelji(n):
    s = []
    for i in range(1, n):
        if n % i == 0:
            s.append(i)
    return s

def vsota(s):
    v = 0
    for e in s:
        v += e
    return v

def popolno(n):
    d = delitelji(n)
    v = vsota(d)
    return v == n


s = []
for i in range(1, 1001):
    if popolno(i):
        s.append(i)
print(s)
```

```
[6, 28, 496]
```

Kako se izvaja tako napisan program? Malo drugače, kot smo vajeni. Začetek programa - vse do mesta `s = []`, so definicije funkcij. Python tega dela programa ne izvede, le zapomni si funkcije, da jih bo kasneje lahko poklical. Stvari se začnejo zares dogajati šele, pri `s = []`. Ko program pride do klica funkcije `popolno`, skoči v to funkcijo – vendar si zapomni, odkod je skočil, tako da se bo kasneje lahko vrnil na to mesto.

Prav. Zdaj smo v funkciji `popolno` in `n` je neka številka (najprej 1, naslednjič bo 2 in tako naprej). Že takoj, v prvi vrstici skoči izvajanje v funkcijo `delitelji`. Ta sestavi seznam deliteljev in ga vrne - tistemu, ki jo je poklical, funkciji `popolno`. Nato se nadaljuje izvajanje funkcije `popolno`: ta v naslednji vrsti pokliče funkcijo `vsota`. Funkcija `vsota` izračuna in vrne vsoto. Spet smo v funkciji `popolno`, ki izračuna vrednost izraza `v == n`; vrednost izraza je `True` ali `False`. Funkcija `popolno` ga vrne tistemu, ki jo je klical, se pravi oni zanki na koncu skripte. Če je rezultat `True`, dodamo število v seznam, sicer ne.

Ta program je lep zato, ker je pregleden. Razdelili smo ga na tri funkcije, vsaka opravlja svoje delo. Ko pišemo eno funkcijo, se ne ukvarjamo s celo sliko, temveč le s tem, kar počne ta funkcija. Če mislimo le na eno stvar naenkrat, nam bo lažje programirati in manj se bomo motili.

1.3 ## Rezultat sredi funkcije

Napišimo funkcijo, ki pove, ali je dano število praštevilo. Vrnila bo `True` (je) ali `False` (ni).

Prejšnjič smo videli, da lahko zanko prekinemo z `break`. Prekinemo jo lahko tudi z `return`.

[34]:
```
def prastevilo(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Prvi `return` je znotraj stavka `if`. Če odkrijemo, da kakšno število med 2 in n-1 deli n (se pravi, če je ostanek po deljenju `n` z `i` enak 0), dano število ni praštevilo in vrnemo `False`. S tem se izvajanje funkcije prekine, funkcija vrne rezultat in konec. Nobenega `break` ali česa podobnega ne potrebujemo. `return` vedno konča izvajanje funkcije. Do drugega `return`a tako pridemo le, če se ni izvedel prvi `return`. To pa seveda pomeni, da ni bilo nobenega števila, ki bi delilo dano število `n`.

Kot so odkrili študenti, funkcija meni, da je 1 praštevilo. To počne iz zgodovinskih razlogov. Če funkcija vrne `False`, si bo še kdo mislil, da je 1 sestavljeno število, ne? Če smo pripravljeni tvegati, pa lahko funkcijo popravimo tako

[35]:
```
def prastevilo(n):
    if n == 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

ali pa celo tako

[36]:
```
def prastevilo(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return n != 1
```

V drugi različici se zanka for ne bo izvedla nikoli, torej ne bo vrnila `False`, `return` na koncu pa bo vrnil `True` le, če število ni 1.

1.4 Funkcije (navadno) vračajo rezultate, ne izpisujejo

Iz neznanega razloga so študentom - kot opažam na izpitih in v domačih nalogah - veliko bolj pri srcu funkcije, ki nekaj izpišejo, kot funkcije, ki vračajo rezultat. Tako bi jih mikalo, recimo, funkcijo za `vsoto` napisati tako, da bi na koncu namesto `return v` pisalo `print(v)`.

Vendar to ni prav! S takšno funkcijo si nimamo kaj pomagati! Ne potrebujemo funkcije, ki *izpiše* vsoto; potrebujemo funkcijo, ki *vrne* vsoto, da bomo lahko s to vsoto še kaj počeli. Če jo bomo hoteli izpisati, bomo pač poklicali funkcijo in izpisali, kar vrne. Funkcija naj jo le izračuna. Si predstavljate, kako neuporabna bi bila funkcija `sin`, če bi le izpisala sinus, namesto da ga vrne? Bi si lahko pri programiranju topov z njo sploh kaj pomagali?

To seveda ne pomeni, da funkcije ne smejo ničesar izpisovati. Smejo; nekatere so pač namenjene temu. Najbolj očiten primer takšne funkcije je očitno kar `print`.

1.5 Funkcije, ki ne vračajo ničesar

Vse funkcije v Pythonu vračajo rezultat. Če funkcija ne vrača ničesar, vrača nič, torej `None`. To se bo zgodilo, kadar funkcija nima `return` ali kadar se ta ne izvede.

Napišimo, na primer, funkcijo, ki kot argument dobi seznam in vrne prvo sodo število v njem.

[37]:
```
def prvo_sodo(s):
    for e in s:
        if e % 2 == 0:
            return e
```

In zdaj jo preskusimo.

[38]:
```
print(prvo_sodo([1, 2, 3, 4, 5]))
```

```
2
```

[39]:
```
print(prvo_sodo([1, 3, 5]))
```

```
None
```

Prvi klic je jasen: funkcija vrne 2. V drugem primeru pa nikoli ne pride do `return`a, zato funkcija pač ne vrne ničesar. Ko smo jo poklicali, se tudi ni nič izpisalo, saj Python, kadar ga poganjamo v načinu za čvekanje, ne izpiše `None` … razen, kadar ga k temu prisilimo s `print`, kot smo storili v zadnji vrstici.

Konstanta `None` se šteje za neresnično. To je praktično, saj jo lahko uporabljamo v pogojnih stavkih. Imejmo nek seznam `s` in izpišimo prvo sodo število ali pa povejmo, da v seznamu ni sodih števil.

[40]:
```
sodo = prvo_sodo([1, 3, 5])
if sodo:
    print("Prvo sodo število je", sodo)
else:
    print("V seznamu ni sodih števil.")
```

```
V seznamu ni sodih števil.
```

Vendar moramo biti pri takšnem početju majčkeno previdni. Kaj, če je prvo sodo število v seznamu 0? Ker je tudi 0 neresnično, bo program v tem primeru napisal, da ni sodih števil. Pravilno bi bilo torej

[41]:
```
sodo = prvo_sodo([1, 3, 5])
if sodo != None:
    print("Prvo sodo število je", sodo)
else:
    print("V seznamu ni sodih števil.")
```

```
V seznamu ni sodih števil.
```

Iz nekega razloga, ki ga bomo pojasnili čez dva tedna, pa običajno pišemo

[42]:
```
sodo = prvo_sodo([1, 3, 5])
if sodo is not None:
    print("Prvo sodo število je", sodo)
else:
    print("V seznamu ni sodih števil.")
```

```
V seznamu ni sodih števil.
```

Pa še nekaj iz lepopisa. Če pišemo funkcijo, ki nikoli ne vrne ničesar, pač ne napišemo `return`-a. 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

[43]:
```
def prvo_sodo(s):
    for e in s:
        if e % 2 == 0:
            return e
    return None
```

1.6 Več rezultatov

Funkcija lahko vrača tudi "več rezultatov", tako kot tale funkcija, ki vrne, kvadrat in kub števila.

[44]:
```
def kk(x):
    return x ** 2, x ** 3
```

Pokličemo jo lahko takole.

[45]:
```
kvad, kub = kk(5)
```

Pozoren študent je najbrž že spregledal: v resnici funkcija vrača en sam rezultat, namreč terko in ob klicu funkcije to terko razpakiramo. Vendar nam o tem, tehničnem vidiku zadeve, ni potrebno razmišljati. Mirno se lahko vedemo, kot da smo dobili dve vrednosti.

V tem primeru navadno ne pišemo oklepajev okrog terk. Funkcija bo delovala tudi, če pišemo `return (x ** 2, x ** 3)`, vendar to ni običajen zapis za vračanje dveh vrednosti.

1.7 Več returnov

Funkcija ima seveda lahko več `return`ov. Izvede se tisti, na katerega naleti. En primer smo že videli (določanje praštevilskosti). Naredimo še enega: funkcijo, ki vrne absolutno vrednost danega števila.

[46]:
```
def abs(x):
    if x < 0:
        return -x
    else:
        return x
```

Funkcija ima torej dva `return`. Eden se izvede, če je število negativno in drugi, če pozitivno.

1.8 Funkcije brez argumentov

Funkcija je lahko tudi brez argumentov. Tule je funkcija, ki nima argumentov, vrne pa 42. Vedno.

[47]:
```
def odgovor():
    return 42
```

Tako funkcijo je potrebno tudi poklicati brez argumentov - oklepajev pa vseeno ne smemo pozabiti.

[48]:
```
odgovor()
```

[48]:
```
42
```

Zdaj, ko znamo definirati funkcije, bo postalo naše življenje veliko lažje in preglednejše. Naši programi so, z vsem kar znamo, začeli postajati dolgi in razvlečeni. Zdaj jih bomo lahko razsekali v posamezne funkcije in se v njih tako lažje znašli.