Uvod v Python

Aritmetični izrazi

Videli smo, da lahko v Pythonu seštevamo, seveda pa so nam voljo tudi ostale osnovne računske operacije: - za odštevanje, \* za množenje in \*\* za potenciranje. Za deljenje Python pozna dve operaciji: običajno deljenje / in pa celoštevilsko deljenje //, ki zavrže morebitni ostanek. Če želimo izračunati samo ostanek, uporabimo %. Prioriteta operatorjev je določena tako kot običajno: najtesneje veže potenciranje, nato množenje in deljenji, nazadnje pa seštevanje in odštevanje. Če želimo vrstni red spremeniti, uporabimo običajne oklepaje. Še to: da je koda bolj berljiva, damo na vsaki strani operatorja po en presledek.

(1 + 5) \*\* (9 - 2)

279936

123 / 10

12.3

123 // 10

12

123 % 10

3

123 \*\* (45 + 67)

1173201153236117392747457141184065170953567764283837482787268119007036898684512512556572577156186549602764788041495818311329933349581701014867937205332087819177539156963702612817234021747525564287508352993790061063457990401206082438721

Vidimo, da velika števila Pythonu ne povzročajo velikih težav.

**Attention**

Razmislite, kako bi izračunali predzadnjih sto števk števila 212345?

Spremenljivke

Izračunane vrednosti si lahko shranimo tudi v spremenljivke, ki jih potem uporabljamo v kasnejših izračunih. Za to uporabimo *prireditveni stavek* oblike

ime\_spremenljivke = vrednost\_ki\_jo\_zelimo\_shraniti

na primer:

a = 3 + 3

Za razliko od aritmetičnih izrazov prireditveni stavek ne vrne nobene vrednosti, zato se izhodna celica ne prikaže. Vseeno pa lahko vidimo, da je Python vrednost izračunal in jo shranil, saj sledeči izraz upošteva, da je a enak 6:

7 \* a

42

V celici lahko zaporedoma izvedemo tudi več ukazov:

b = a + 8

c = 3 \* b

a + b + c

5 \* c

210

Vidimo, da se je v izhodni celici prikazala le vrednost zadnjega izraza. Če želimo, lahko v prireditvenem stavku nastavimo več vrednosti hkrati:

a, b = 10, 15

a + b

25

Vrednost spremenljivke lahko tudi povozimo z novo vrednostjo, vendar to na preostale spremenljivke ne vpliva, saj se vedno shrani tista vrednost, ki smo jo podali v prireditvenem stavku.

a = 10

b = a + 3

b

13

Na primer, če vrednost a povozimo, se vrednost b ne spremeni, saj je prireditveni stavek b = a + 3 najprej izračunal vrednost desne strani, torej 13, in v b shranil število samo.

a = 25

b

13

Za primer daljšega programa si oglejmo [Fermijevo oceno](https://sl.wikipedia.org/wiki/Fermijev_problem) števila učiteljev matematike v slovenskih osnovnih šolah:

stevilo\_slovencev = 2000000

pricakovana\_zivljenska\_doba = 75

velikost\_generacije = stevilo\_slovencev / pricakovana\_zivljenska\_doba

stevilo\_osnovnosolcev = 9 \* velikost\_generacije

stevilo\_razredov = stevilo\_osnovnosolcev / 25

stevilo\_ur\_matematike\_na\_teden = 4.5 \* stevilo\_razredov

stevilo\_uciteljev\_matematike = stevilo\_ur\_matematike\_na\_teden / 20

stevilo\_uciteljev\_matematike

2160.0

V Sloveniji bi tako moralo biti približno 2000 učiteljev matematike. Opazimo, da lahko imena spremenljivk vsebujejo več kot eno črko, česar smo navajeni v matematiki. V programiranju je zelo pomembno, da so imena čimbolj opisna, saj tako hitreje razumemo, kaj počne program.

Funkcije

Pod imeni pa se ne skrivajo samo števila temveč tudi funkcije. Na primer, Python poleg aritmetičnih operacij ponuja osnovne funkcije, kot na primer max in min za izračun maksimuma in minimuma. Pokličemo jih tako, da za imenom funkcije v oklepajih naštejemo argumente, ločene z vejicami:

max(3, 6)

6

min(12, -5)

-5

max(min(10, 20), 30 // 2)

15

Matematične funkcije so na voljo v ločeni knjižnici math. Do njih lahko dostopamo na dva osnovna načina:

Knjižnico uvozimo s stavkom import math, nato pa do funkcij in konstant dostopamo tako, da dodamo math. pred vsako ime:

**import** **math**

math.sqrt(2) / 2

0.7071067811865476

math.sin(math.pi / 4)

Drugi način pa je, da iz knjižnice s stavkom from math import ... uvozimo posamezne vrednosti, nato pa do njih dostopamo direktno:

**from** **math** **import** sqrt, sin, pi

sqrt(2) / 2

0.7071067811865476

sin(pi)

1.2246467991473532e-16

**Attention**

V zadnjem ukazu nismo dobili pričakovanega odgovora 0. Računalnik namreč ne dela s čisto pravimi realnimi števili, temveč z njihovimi približki, ki jim pravimo *števila s plavajočo vejico*. Za ta števila običajno najprej zapišemo decimalke (ki jim pravimo *mantisa*), nato pa še eksponent. Število, ki smo ga dobili, je tako enako približno 1,22⋅10−16, saj e-16 pomeni 10−16. Na primer 3.2445e2 pa označuje število 324,45=3,2445⋅102).

Obstaja tudi tretji način, ko iz knjižnice s stavkom from math import \* uvozimo vse naštete vrednosti, vendar je odsvetovan, ker potem nikoli ne vemo, kaj vse smo uvozili.

Preprečevanje napak

V programih lahko naredimo tri glavne vrste napak

Sintaktične napake

Pri teh napakah program napišemo drugače, kot določajo pravila. Na primer, če argumente funkcije ločimo s podpičjem namesto z vejico, ali pa če narobe pišemo oklepaje:

max(2; 4)

File **"/tmp/ipykernel\_1770/3958682054.py"**, line 1

max(2; 4)

^

SyntaxError: invalid syntax

max(2, 4))

File **"/tmp/ipykernel\_1770/309947572.py"**, line 1

max(2, 4))

^

SyntaxError: invalid syntax

Na take napake nas Python opozori, še preden začne z izvajanjem programa, zato jih ne moremo zgrešiti.

Napake ob izvajanju

Pri teh napakah program napišemo sintaktično pravilno, vendar ga Python ne zna ali ne more izvesti:

1 / 0

---------------------------------------------------------------------------

ZeroDivisionError Traceback (most recent call last)

/tmp/ipykernel\_1770/1455669704.py **in** <module>

----> 1 1 / 0

ZeroDivisionError: division by zero

mix(2, 4)

---------------------------------------------------------------------------

NameError Traceback (most recent call last)

/tmp/ipykernel\_1770/3879753672.py **in** <module>

----> 1 mix(2, 4)

NameError: name 'mix' is not defined

Opozorila o napakah si bomo še ogledali bolj podrobno, zaenkrat pa si zapomnimo le, da je ključna informacija o napaki v zadnji vrstici opozorila. V prvem primeru je bila napaka deljenje z 0, v drugem pa to, da ime mix ni definirano.

Na precej takih napak (npr. na to, da uporabimo nedefinirano ime) nas bo opozoril že urejevalnik, lahko pa se zgodi, da se bodo pojavile šele ob izvajanju. Pri Uvodu v programiranje to običajno ne bo težava, v praksi pa zna biti precej nerodno, sploh kadar gre za kritično pomemben program (npr. za nadzor jedrskega reaktorja) ali pa kadar s tem izgubimo veliko dela (recimo, da se računalnik po 10-urnem izračunu ustavi, preden izpiše rezultat). Lahko se tudi zgodi, da do napak pride šele ob kakšnih robnih pogojih, zato jih lahko precej časa sploh ne opazimo. Vseeno pa je njihova prednost vsaj ta, da jih opazimo, kadar se zgodijo (kot bomo videli, jih lahko včasih tudi naknadno rešimo).

Vsebinske napake

Pri vsebinskih napakah je vse videti v redu, vendar je koncu dobimo napačen odgovor, ker smo podali napačna navodila. Recimo, da želimo izračunati razdaljo med točkama (2, 3) in (5, 7):

((2 - 5) \*\* 2 + (3 - 7) \*\* 2) \*\* 1/2

Program smo napisali brez sintaktičnih napak in izvajanje je uspešno vrnilo rezultat, ki pa je žal napačen, ker nismo potencirali na 1/2, temveč potencirali na 1 in delili z 2, saj ima potenciranje prednost pred deljenjem. Take napake so še posebej zlobne, ker jih lahko precej dolgo časa ne opazimo. Znan primer te napake je [Mars Climate Orbiter](https://en.wikipedia.org/wiki/Mars_Climate_Orbiter), ki je po devetih mesecih potovanja proti Marsu prehitro vstopil v atmosfero in razpadel. Vzrok je bil v tem, da je del kode delal s SI merskimi enotami, del kode pa z imperialnimi. Škode je bilo za 300 milijonov dolarjev.

Na vsebinske napake nas računalnik ne more opozoriti, saj nam ne zna brati misli. Izognemo se jim lahko tako, da programe pišemo strukturirano iz manjših delov, ki jih sproti preverjamo. Ključno pa je, da jih pišemo berljivo:

* Na vsaki strani dvomestne operacije (=, +, \*\*, …) pišemo presledek (tako bi hitreje prepoznali, da \*\* 1/2 v resnici pomeni \*\* 1 / 2).
* Za ločili (na primer ,) pišemo presledek, pred njimi pa ne.
* Spremenljivkam dajemo opisna imena, ki jih pišemo z malimi črkami. Posamezne besede ločimo z znakom \_.

Na primer, računalnik bi razumel tudi sledeči izračun števila učiteljev matematike:

s,z=2000000,75

g=s/z

o=9\*g

r=o/25

m=4.5\*r

u=m/20

u

2160.0

ali celo:

s,z=2000000,75;g=s/z;o=9\*g;r=o/25;m=4.5\*r;u=m/20;u

2160.0

a pri takem pisanju se nam bo hitro pojavila napaka.

Definicije funkcij

Kako bi izračunali ploščino trikotnika s stranicami dolžin 4, 13 in 15? Pomagamo si lahko s Heronovo formulo

s(s−a)(s−b)(s−c)

kjer je s=(a+b+c)/2. Tako bi lahko napisali:

a, b, c = 4, 13, 15

s = (a + b + c) / 2

ploscina = math.sqrt(s \* (s - a) \* (s - b) \* (s - c))

ploscina

24.0

Kako pa bi izračunali površino tetraedra pa bi izračunali površino tetraedra, ki ima za lica štiri trikotnike? Načeloma bi lahko pisali:

a, b, c, d, e, f = 896, 1073, 1073, 990, 1073, 1073

s\_abc = (a + b + c) / 2

p\_abc = math.sqrt(s\_abc \* (s\_abc - a) \* (s\_abc - b) \* (s\_abc - c))

s\_aef = (a + e + f) / 2

p\_aef = math.sqrt(s\_aef \* (s\_aef - a) \* (s\_aef - e) \* (s\_aef - f))

s\_cde = (c + d + e) / 2

p\_cde = math.sqrt(s\_cde \* (s\_cde - c) \* (s\_cde - d) \* (s\_cde - f))

s\_bdf = (b + d + f) / 2

p\_bdf = math.sqrt(s\_bdf \* (s\_bdf - b) \* (s\_bdf - d) \* (s\_bdf - f))

povrsina = p\_abc + p\_aef + p\_bdf + p\_cde

povrsina

1816080.0

Kot vidimo, to ni najbolj pregledno. V taki kodi z veliko verjetnostjo naredimo kakšno napako. Bolje je, da uporabimo funkcije. Že prej smo uporabili nekaj vgrajenih funkcij, Python pa nam omogoča, da si funkcije definiramo tudi sami. Na primer, definicija funkcije za izračun ploščine trikotnika je sledeča:

**def** ploscina\_trikotnika(a, b, c):

*"""Vrne ploščino trikotnika z danimi stranicami."""*

s = (a + b + c) / 2

ploscina = math.sqrt(s \* (s - a) \* (s - b) \* (s - c))

**return** ploscina

Oglejmo si njene sestavne dele. Vsaka definicija funkcije se začne s ključno besedo def, ki ji sledi ime funkcije, v našem primeru ploscina\_trikotnika, tej pa v oklepajih našteti argumenti, ki jih funkcija sprejme. Funkcije lahko sprejmejo različno število argumentov. Naša sprejme tri argumente, ki jih bomo shranili v spremenljivke a, b in c. Za dvopičjem sledi glavni del: *telo funkcije*, torej ukazi, ki naj se izvedejo, ko funkcijo pokličemo. Telo moramo zamakniti za štiri presledke, da se jasno vidi, kaj vse obsega.

V prvo vrstico telesa lahko zapišemo neobvezni *dokumentacijski niz* oziroma *docstring*, v katerem na kratko opišemo, kaj funkcija počne. Dokumentacijski niz ni obvezen, je pa koristen, saj uporabniku prek ustrezne podpore v urejevalniku ali funkcije help pove, kaj funkcija počne. Drugo in tretjo vrstico telesa smo že videli, v zadnji pa z ukazom return povemo, katero vrednost naj vrne funkcija. Tako definirano funkcijo potem kličemo na enak način kot vgrajene funkcije.

ploscina\_trikotnika(4, 13, 15)

24.0

S pomočjo funkcije ploscina\_trikotnika lahko tudi na veliko bolj pregleden način zapišemo funkcijo za izračun površine tetraedra:

**def** povrsina\_tetraedra(a, b, c, d, e, f):

p\_abc = ploscina\_trikotnika(a, b, c)

p\_aef = ploscina\_trikotnika(a, e, f)

p\_bdf = ploscina\_trikotnika(b, d, f)

p\_cde = ploscina\_trikotnika(c, d, e)

**return** p\_abc + p\_aef + p\_bdf + p\_cde

povrsina\_tetraedra(896, 1073, 1073, 990, 1073, 1073)

1816080.0

Tako kot drugje v Pythonu, se tudi stavki v telesu funkcije izvajajo od prvega proti zadnjemu. Ko dosežemo stavek return, funkcija vrne vrednost danega izraza ter zaključi z izvajanjem. Tako tudi funkcija

**def** f(x):

**return** x \*\* 2

**return** 1000

vrne kvadrat števila x in ne števila 1000, saj se izvajanje ustavi ob prvem stavku return, zato do drugega sploh ne pride.

f(10)

100

Če stavka return ne napišemo, funkcija vrne posebno vrednost None, ki označuje manjkajočo vrednost. Pozorno se ji bomo posvetili kasneje, zaenkrat pa jo omenimo le zato, da bomo znali razumeti spodnjo (precej pogosto) napako:

**def** g(x):

x \*\* 2

2 \* g(10)

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

/tmp/ipykernel\_1770/89980391.py **in** <module>

2 x \*\* 2

3

----> 4 2 \* g(10)

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

Pričakovali bi, da bo rezultat klica 2 \* g(10) enak 200. Toda ker smo v funkciji g pozabili na return, je funkcija vrnila vrednost None. To lahko razberemo iz opozorila, v katerem približno piše, da operacije \* ne moremo uporabiti na celem številu in vrednosti None. Vsakič, ko dobite opozorilo TypeError, v katerem se pojavlja NoneType, posumite na to, da nekje manjka stavek return.

Argumenti funkcije in spremenljivke, ki jih definiramo v telesu funkcije, se izven funkcije ne vidijo. Pravimo, da so *lokalne*. Namen tega je, da funkcije ne motijo ena druge s spremenljivkami, ki jih uporabljajo. Na primer, če definiramo

**def** f(x):

y = 3 \* x

**return** y

tedaj tudi po klicu funkcije f ne x ne y ne bosta definirana:

f(4)

x

---------------------------------------------------------------------------

NameError Traceback (most recent call last)

/tmp/ipykernel\_1770/32546335.py **in** <module>

----> 1 x

NameError: name 'x' is not defined

y

---------------------------------------------------------------------------

NameError Traceback (most recent call last)

/tmp/ipykernel\_1770/3563912222.py **in** <module>

----> 1 y

NameError: name 'y' is not defined

Če pa je y na primer že definiran drugje, pa ga klic funkcije f ne zmoti:

y = 10

f(4)

12

y

10

Včasih imamo za nekatere argumente funkcij v mislih že prav določeno vrednost. Na primer, za izračun logaritma potrebujemo dve števili: osnovo in argument (tudi logaritmand). Toda velikokrat za osnovo vzamemo 10, zato namesto log10⁡x pišemo kar log⁡x. V Pythonu lahko neobvezne argumente določimo tako, da jim podamo privzeto vrednost:

**def** koren(x, n=2):

**return** x \*\* (1 / n)

Pozor, okoli enačajev pri neobveznih argumentih **ne pišemo presledkov**. Na podoben način take funkcije tudi kličemo:

koren(64, n=3)

3.9999999999999996

V primeru, da drugega argumenta ne podamo, bi Python uporabil privzeto vrednost:

koren(64)

8.0

Logične vrednosti

Poleg števil Python pozna tudi logični vrednosti True in False, ki označujeta resnico in neresnico. Logične vrednosti ponavadi dobimo kot rezultat primerjav, kot so enakost ==, neenakost != ali urejenostne relacije <, >, <=, >=, ter prek logičnih operacij and, or in not.

1 + 1 == 3

False

3 != 2

True

**True** **and** **False**

False

**not** (5 == 10)

True

3 < 5 **or** 10 > 20

True

Logične vrednosti uporabimo v *pogojnih stavkih* (oziroma *stavkih* if) oblike

**if** pogoj:

*# stavki, ki jih izvedemo,*

*# ko pogoj drži*

**else**:

*# stavki, ki jih izvedemo,*

*# ko pogoj ne drži*

Ključnima besedama if/else in pripadajočim stavkom pravimo tudi *veji pogojnega stavka*. Stavke v obeh vejah moramo zamakniti za štiri presledke tako kot v funkcijah.

Na primer, če izvedemo program

x = 5

**if** x < 10:

y = 2 \* x

**else**:

y = 3 \* x - 1

x = y + 7

se bo izvedla veja if, zato bo x na koncu enak 17, y pa 10. V primeru, da bi bila začetna vrednost x = 12, pa bi se izvedla veja else in vrednost x bi na koncu bila 42, vrednost y pa 35.

Če želimo, lahko vejo else v pogojnem stavku tudi izpustimo. V tem primeru se ob neizpolnjevanju pogoja ne zgodi nič.

x = 5

y = 8

**if** x > 10:

y = 2 \* x

y

8

Pogojne stavke lahko pišemo tudi v funkcijah. Na primer, funkcijo, ki računa absolutno vrednost, lahko s pomočjo pogojnega stavka napišemo kot:

**def** absolutna\_vrednost(x):

**if** x >= 0:

**return** x

**else**:

**return** -x

absolutna\_vrednost(-5)

5

absolutna\_vrednost(3)

3

Če bi želeli vrniti predznak števila, pa moramo ločiti tri primere: negativno število, nič in pozitivno število. To lahko storimo kot:

**def** predznak(x):

**if** x < 0:

**return** -1

**else**:

**if** x == 0:

**return** 0

**else**:

**return** 1

Zgornji pogojni stavek je malo nerodno zapisan. Ker se nam bo dostikrat zgodilo, da se ne bomo odločali le med dvema primeroma, temveč med večimi, nam Python omogoča splošnejše pogojne stavke oblike:

**if** pogoj1:

*# stavki, ki jih izvedemo,*

*# ko pogoj1 drži*

**elif** pogoj2:

*# stavki, ki jih izvedemo,*

*# ko pogoj1 ne drži, ampak drži pogoj2*

**elif** pogoj3:

*# stavki, ki jih izvedemo,*

*# ko tudi pogoj2 ne drži, ampak drži pogoj3*

**else**:

*# stavki, ki jih izvedemo,*

*# ko noben od pogojev ne drži*

Beseda elif je okrajšava za else-if. Funkcijo za izračun predznaka bi lepše zapisali kot

**def** predznak(x):

**if** x < 0:

**return** -1

**elif** x == 0:

**return** 0

**else**:

**return** 1

Izrazi & stavki

V Pythonovih programih ločimo med *izrazi* in *stavki*. Izrazi so vse, kar sestavimo iz funkcij in operacij ter uporabljamo kot argumente funkcij, desne strani prireditvenih izrazov ali pogoje v pogojnih stavkih. Stavki pa so osnovni gradniki Pythonovih programov in jih pišemo enega pod drugim. Zaenkrat smo videli tri vrste stavkov: prva so bili prireditveni stavki, drugi pogojni stavki (ki so potem spet sestavljeni iz gnezdenih stavkov), tretja in najmanj opazna pa so bili izrazi. Običajne izraze lahko prav tako pišemo v programe, vendar ne bodo imeli posebnega učinka. Če napišemo

x = 10

10 + 10

y = 20

se bo vsota 10 + 10 res izračunala, vendar se ne bo nikamor shranila in Python bo na njo hitro pozabil. Kmalu pa bomo srečali tudi izraze, ki bodo imeli vpliv na nadaljnje izvajanje programov.

Python poleg pogojnih stavkov podpira tudi pogojne izraze, s katerimi nekatere stvari napišemo malo elegantneje. Na primer, zgornjo določitev osnovnih točk bi lahko pisali kot:

**def** absolutna\_vrednost(x):

**return** x **if** x >= 0 **else** -x

V pogojnih izrazih moramo vedno napisati obe možnosti, prav tako pa ne moremo uporabiti elif-a. Načeloma lahko pogojne izraze verižimo kot:

**def** predznak(x):

**return** -1 **if** x < 0 **else** 0 **if** x == 0 **else** 1

samo to je preveč natlačeno, da bi bilo berljivo. Pogojni stavki so torej precej omejeni, ampak vseeno jih omenjamo, ker znajo včasih kakšno stvar narediti preglednejšo.

# Nizi

V programih seveda ne delamo le s števili, temveč tudi z drugimi podatki, na primer besedili. V ta namen Python podpira nize, ki so strnjena zaporedja znakov. Nize običajno pišemo v enojnih narekovajih, na primer 'to je primer niza'.

## Osnovne operacije na nizih

Nize lahko stikamo z operacijo + in množimo s celimi števili:

'zala' + 'gasper'

'zalagasper'

'tro' + 4 \* 'lo'

'trololololo'

Dolžino niza dobimo s funkcijo len:

len('lokomotiva')

10

Pogosto uporabljana funkcija je tudi str, ki dano vrednost pretvori v niz z njeno predstavitvijo:

str(110)

'110'

Nize lahko med seboj tudi primerjamo. Pri tem Python nize ureja leksikografsko, torej tako, kot bi bili urejeni v leksikonu ali kazalu: najprej primerja prvi črki, če sta ti dve enaki, pogleda drugi dve, in tako naprej. Pri tem velike črke pridejo na vrsto pred malimi, na šumnike pa se brez posebnih knjižnic Python ne spozna.

'beseda' == 'konj'

False

'abak' <= 'abeceda'

True

Na nizih imamo na voljo tudi predikat in, s katerim ugotovimo, ali se nek niz pojavlja kot podniz v drugem nizu. Na voljo je tudi not in, s katerim bolj berljivo zapišemo ravno nasprotno stvar:

'gram' **in** 'Uvod v programiranje'

True

'liter' **in** 'Uvod v programiranje'

False

**not** ('liter' **in** 'Uvod v programiranje')

True

'liter' **not** **in** 'Uvod v programiranje'

True

## Indeksi in rezine

Do posameznega znaka v nizu pridemo z indeksi. Z izrazom niz[i] dostopamo do i-tega znaka v danem nizu:

'REKURZIJA'[3]

'U'

'REKURZIJA'[0]

'R'

'REKURZIJA'[-1]

'A'

Indeksi se začnejo šteti z 0. Če uporabimo negativna števila, lahko štejemo tudi od zadaj, vendar tam začnemo šteti z -1 (saj je -0 = 0).

\ 0 1 2 3 4 5 6 7 8

R E K U R Z I J A

-9 -8 -7 -6 -5 -4 -3 -2 -1

Na podoben način lahko dostopamo tudi do podnizov. Če napišemo niz[i:j] bomo dobili niz, ki mu pravimo rezina in sega od vključno i-tega do vključno j - 1-tega znaka. Če kakšno od meja izpustimo, bomo vzeli vse znake od začetka oziroma do konca.

Videli bomo, da razpon od i do j v Pythonu pogosto pomeni vse od vključno i do vključno j−1. To je malo neobičajno, vendar ima nekaj prednosti: razpon od i do j vedno vsebuje j−i elementov, hkrati pa lahko razpone sestavljamo brez prekrivanja. Na primer, niz[i:j] + niz[j:k] je enako kot niz[i:k] (vsaj, če velja i < j < k).

'REKURZIJA'[2]

'K'

'REKURZIJA'[6]

'I'

'REKURZIJA'[2:6]

'KURZ'

'REKURZIJA'[:6]

'REKURZ'

'REKURZIJA'[2:]

'KURZIJA'

Pišemo lahko tudi niz[i:j:k], s čimer vzamemo le vsak k-ti znak:

'REKURZIJA'[1:8]

'EKURZIJ'

'REKURZIJA'[1:8:1]

'EKURZIJ'

'REKURZIJA'[1:8:2]

'EUZJ'

'REKURZIJA'[1:8:3]

'ERJ'

'REKURZIJA'[::-1]

'AJIZRUKER'

S pomočjo indeksov in rezin lahko napišemo (ne najbolj učinkovito) funkcijo, ki prešteje vse samoglasnike v danem nizu:

**def** stevilo\_samoglasnikov(niz):

**if** niz == '':

**return** 0

**elif** niz[0] **in** 'aeiouAEIOU':

**return** 1 + stevilo\_samoglasnikov(niz[1:])

**else**:

**return** stevilo\_samoglasnikov(niz[1:])

stevilo\_samoglasnikov('Uvod v programiranje')

7

Funkcija deluje tako, da najprej pogleda, če je niz prazen. Če je, v njem ni samoglasnikov, zato vrne 0. Če ni, pogleda prvi znak. Če je samoglasnik, potem je število samoglasnikov za ena večje od števila samoglasnikov v preostanku niza (ki ga dobimo s pomočjo rezine), sicer pa je enako številu samoglasnikov v preostanku.

## Zapisi nizov

Nize lahko pišemo tudi z dvojnimi narekovaji, ki jih ponavadi uporabimo takrat, kadar v nizu želimo uporabiti enojni narekovaj: "Tole je kr'neki!". V tem primeru niza ne moremo pisati med enojnimi narekovaji, saj bi Python po narekovaju za kr mislil, da je konec niza.

'Tole je kr'neki!'

File "/tmp/ipykernel\_1809/1353015095.py", line 1

'Tole je kr'neki!'

^

SyntaxError: invalid syntax

Včasih želimo uporabiti obe vrsti narekovajev. V tem primeru si pomagamo z ubežnimi znaki. To so znaki, ki jih na običajni način ne moremo zapisati, zato uporabimo poseben zapis, ki se začne z znakom \. Tedaj lahko pišemo '"Tole je kr\'neki," je rekla.' ali pa "\"Tole je kr'neki,\" je rekla.\". Ubežne znake brez težav lahko pišemo tudi tedaj, kadar ni treba '\"Grem v rudnik,\" je rekla.'. Z ubežnimi znaki lahko zapišemo tudi znak za novo vrstico \n, za tabulator \t in seveda tudi za poševnico \\, saj je ne moremo pisati le kot \, ker bi Python to razumel kot začetek ubežnega znaka.

Nize lahko pišemo tudi med tri enojne (''') ali tri dvojne (""") narekovaje (ki smo jih videli že pri dokumentacijskem nizu). V tem primeru za en sam narekovaj ne potrebujemo ubežnega znaka. Take nize lahko pišemo tudi čez več vrstic.

Različni zapisi ne vplivajo na vsebino. Tako '"Živjo!"', '\"Živjo!\"',"\"Živjo!\"", '''"Živjo!"''' ali """"Živjo!\"""" vsi predstavljajo enak niz z osmimi znaki.

## Vgrajene metode na nizih

Precej operacij na nizih pa lahko opravimo preko metod. To so funkcije, ki jih na poseben način kličemo na posamezni vrednosti. Na primer, za pretvarjanje niza v male črke pokličemo

'REKURZIJA'.lower()

'rekurzija'

Klic metod na nizih ima splošno obliko niz.metoda(...), kjer v oklepajih naštejemo argumente. Na primer, pojavitve posameznega znaka v nizu preštejemo z metodo count:

niz = 'Otorinolaringolog'

niz.count('o')

4

Klice metod lahko tudi verižimo:

niz = 'Otorinolaringolog'

niz.lower().count('o')

5

Vse metode, ki so na voljo na nizih lahko najdete v [uradni dokumentaciji](https://docs.python.org/3/library/stdtypes.html#string-methods), zato naštejmo le najbolj pogosto uporabljane:

* s.count(t) vrne število pojavitev podniza t v nizu s. Klic s.count(t, i) deluje podobno, le da začne šteti šele pri indeksu i, klic s.count(t, i, j) pa konča šteti pri indeksu j.
* s.index(t) vrne najmanjši indeks v nizu s, kjer se niz t pojavi kot podniz. Podobno kot prej klic s.index(t, i) začne iskati pri indeksu i, klic s.index(t, i, j) pa konča pri indeksu j. Če niza ni, metoda sproži napako. Metoda s.find se obnaša enako kot s.index, le da v primeru, ko podniza ne najde, vrne -1.
* s.join(sez) z ločilom s skupaj stakne vse nize iz seznama sez (te bomo spoznali [kmalu](https://matija.pretnar.info/uvod-v-programiranje/05-seznami-in-nabori.html#seznami-in-nabori)).
* s.replace(t1, t2) vrne niz s, v katerem smo vse pojavitve podniza t1 zamenjali s podnizi t2. Klic s.replace(t1, t2, n) pa zamenja le prvih n pojavitev.
* s.strip() vrne niz s, v katerem smo odstranili vse bele znake (presledke, tabulatorje, nove vrstice) z začetka in konca. Klic s.strip(t) z začetka in konca odstrani vse znake iz niza t.
* s.lower() / s.upper() / s.title() / s.capitalize() / s.swapcase() vrnejo niz s, v katerem smo vse črke zamenjali z malimi / vse črke zamenjamo z velikimi / vsem besedam damo veliko začetnico / na začetku niza damo veliko začetnico / male črke zamenjamo z velikimi in obratno.
* s.split() vrne seznam besed v nizu s (ločene glede na bele znake). Klic s.split(t) loči glede na podniz t. Klic s.split(t, n) vrne niz razbit na prvih n ločilih.
* s.isdigit() / s.isalpha() / s.islower() / s.isupper() / s.isalnum() / s.isspace() vrnejo True natanko takrat, kadar je niz s neprazen in so vsi znaki števke / črke / male črke / velike črke / črke ali številke / beli znaki.

## f-nizi

Včasih je bila v Pythonu zelo uporabna tudi metoda format. Ta vzame niz, v katerem so z zavitimi oklepaji označeni prostori, ki jih zapolnimo z argumenti metode. Na primer

'*{0}* ima *{1}*'.format('mama', 'stol')

'mama ima stol'

ali pa

niz = '*{0}* vzklika: "*{1}*, *{1}*, *{1}*!"'

niz.format('Mama', 'joj')

'Mama vzklika: "joj, joj, joj!"'

niz.format('Tone', 'FMF')

'Tone vzklika: "FMF, FMF, FMF!"'

Argumente lahko tudi oblikujemo in poimenujemo, kar je razloženo v [uradni dokumentaciji](https://docs.python.org/3/library/string.html#format-string-syntax). Po novem, od Pythona 3.6 naprej, pa lahko uporabljamo tudi *f*-nize. To so nizi, ki jih na začetku označimo z znakom f, nato pa v zavite oklepaje zapišemo vrednost, ki naj jo Python izračuna in vstavi v niz. Na primer:

kdo = 'Mama'

kaj = 'joj'

f'*{*kdo*}* vzklika: "*{*kaj*}*, *{*kaj*}*, *{*kaj*}*!"'

'Mama vzklika: "joj, joj, joj!"'

ali pa

a, b = 22, 7

f'*{*a*}*/*{*b*}* = *{*a / b*:*.5*}*'

'22/7 = 3.1429'

Tako kot pri metodi format lahko tudi pri f-nizih izračunane vrednosti oblikujemo:

a, b = 22, 7

f'*{*a*}*/*{*b*}* = *{*a / b*:*.5*}*'

'22/7 = 3.1429'

f'popust znaša *{*1 / 3*:*.2%*}*'

'popust znaša 33.33%'

f'*{*"NASLOV"*:*\*^30*}*'

'\*\*\*\*\*\*\*\*\*\*\*\*NASLOV\*\*\*\*\*\*\*\*\*\*\*\*'

Možnosti za oblikovanje je veliko, vse pa so opisane v [uradni dokumentaciji](https://docs.python.org/3/reference/lexical_analysis.html#f-strings).

## Pisanje na konzolo in branje s konzole

Seveda od uporabnikov ne pričakujemo, da bodo naše programe uporabljali le tako, da bodo v interaktivni zanki poganjali funkcije ter brali njihove rezultate. Raje bi imeli prijazen uporabniški vmesnik, v katerem računalnik uporabnika prijazno pozove za vnos, uporabnik potem nekaj natipka (ne nujno v Pythonu), računalnik pa se na to ustrezno odzove. V ta namen sta namenjeni dve funkciji print, ki dani niz izpiše na konzolo, ter input, ki vrne niz, ki ga je vtipkal uporabnik.

print('Pozdravljen, svet!')

Pozdravljen, svet!

print('"Tole je kr**\'**neki," je rekla.')

"Tole je kr'neki," je rekla.

Kot vidimo, je izpis funkcije print namenjen uporabnikom.

Zanke

Rekurzivni klici nam omogočajo, da se pri izvajanju poljubno mnogokrat vrnemo na poljubne konce programa. V veliki večini primerov pa nam je dovolj že to, da ponovimo določen del programa. Temu so namenjene zanke.

Zanka while

Z zanko while določene ukaze izvajamo toliko časa, dokler velja dani pogoj. Pišemo jo kot

**while** pogoj:

*# stavki, ki jih ponavljamo*

*# od prvega do zadnjega,*

*# dokler velja pogoj*

Na primer, program

n = 12

**while** n < 1000:

n = n \* 2

print(n)

24

48

96

192

384

768

1536

bo n nastavil na 12, nato pa ga toliko časa podvojeval in izpisoval, dokler njegova vrednost ne bo presegla 1000.

V programih bomo pogostokrat novo vrednost spremenljivke izračunali tako, da bomo spremenili staro (zato tudi govorimo o spremenljivkah). Na primer, ko bomo šteli vsa praštevila med 1 in 1000000, bomo imeli spremenljivko, ki bo imela na začetku vrednost 0, nato pa jo bomo ob vsakem praštevilu povečali za 1. V ta namen lahko namesto n = n \* 2 pišemo kar operator n \*= 2, saj je \*= operacija, ki spremenljivko na levi pomnoži z vrednostjo na desni. Tudi za ostale operacije obstajajo podobne bližnjice, na primer -=, \*=, //= in tako naprej.

x = 3

x += 2

x \*= 4

x

20

Zgornji program bi tako na krajše lahko napisali kot:

n = 12

**while** n < 1000:

n \*= 2

print(n)

24

48

96

192

384

768

1536

Za primer izračunajmo stopnjo največje potence števila 2, ki še deli število 1580160. To storimo tako, da število zaporedoma celoštevilsko delimo z 2 in ob vsakem deljenju števec stopnje povečamo za 1. Ukaze ponavljamo toliko časa, dokler je ostanek pri deljenju z 2 enak 0:

n = 1580160

stopnja = 0

**while** n % 2 == 0:

n //= 2

stopnja += 1

print(stopnja)

stopnja

1

2

3

4

5

6

7

7

Program bi lahko pretvorili tudi v splošnejšo funkcijo:

**def** celostevilski\_logaritem(n, k):

*"""Vrne stopnjo največje potence števila k, ki še deli n."""*

stopnja = 0

**while** n % k == 0:

n //= k

stopnja += 1

**return** stopnja

celostevilski\_logaritem(81, 3)

4

celostevilski\_logaritem(1580160, 2)

7

Isto funkcijo bi lahko napisali tudi z rekurzijo:

**def** celostevilski\_logaritem\_rek(n, k):

*"""Vrne stopnjo največje potence števila k, ki še deli n."""*

**if** n % k == 0:

**return** 1 + celostevilski\_logaritem\_rek(n // k, k)

**else**:

**return** 0

celostevilski\_logaritem\_rek(81, 3)

4

celostevilski\_logaritem\_rek(1580160, 2)

7

Vseeno v praksi za tiste programe, pri katerih neko stvar ponavljamo toliko časa, dokler velja določen pogoj, raje uporabimo zanko while, saj je učinkovitejša (vsaj v Pythonu, v drugih jezikih je rekurzija ravno tako učinkovita). Poleg tega se Python pritoži, če gremo pri rekurziji pregloboko. Običajno se to zgodi takrat, kadar smo rekurzijo napisali tako, da se ne ustavi. Vendar računalnik tega ne more vedeti, zato se Python ustavi po določenem številu klicev ustavi:

celostevilski\_logaritem\_rek(2 \*\* 2950, 2)

2950

celostevilski\_logaritem\_rek(2 \*\* 2960, 2)

2960

Pri zankah teh težav ni:

celostevilski\_logaritem(2 \*\* 2950, 2)

2950

celostevilski\_logaritem(2 \*\* 2960, 2)

2960

celostevilski\_logaritem(2 \*\* 10000, 2)

10000

Se pa lahko pri zanki while zgodi, da se njeno izvajanje nikoli ne zaključi. Na primer, če bi poklicali celostevilski\_logaritem(12345, 1), bi bil ostanek pri deljenju z 1 v pogoju vedno enak 0 in zanka bi tekla v neskončnost. Ko se naveličamo čakanja, lahko pritisnemo Ctrl-C in izvajanje prekinemo.

Zanka for

Zanko while torej uporabimo takrat, kadar želimo ukaze ponavljati, dokler velja nek pogoj. Včasih pa že vnaprej vemo, kolikokrat bomo te ukaze ponovili. Na primer, funkcijo za izračun fakultete bi lahko pisali kot:

**def** fakulteta(n):

*"""Vrne fakulteto naravnega števila n."""*

produkt = 1

i = 1

**while** i <= n:

produkt \*= i

i += 1

**return** produkt

vendar vemo, da se bo zanka izvedla natanko enkrat za vsako število od 1 do n. Poleg tega se nam hitro zgodi, da vrstico i += 1 po nesreči pozabimo ali napišemo kot i + 1 ali kot i = 1, zaradi česar se zanka izvaja v neskončnost. Za primere, ko vemo, kolikokrat izvedemo določeno kodo, raje uporabimo zanko for, ki jo pišemo kot:

**for** spremenljivka **in** vrednosti:

*# stavki, ki jih izvedemo*

*# po enkrat za vsako izmed*

*# podanih vrednost spremenljivke*

izvede pa se enako, kot če bi napisali

*# v spremenljivko shrani prvo vrednost*

*# izvedi vse stavke v telesu zanke*

*# v spremenljivko shrani drugo vrednost*

*# izvedi vse stavke v telesu zanke*

*# ...*

*# v spremenljivko shrani zadnjo vrednost*

*# izvedi vse stavke v telesu zanke*

Na primer, čez vse črke danega niza se sprehodimo kot:

**for** znak **in** 'to je en niz':

print(znak)

t

o

j

e

e

n

n

i

z

samoglasnike pa lahko preštejemo kot:

**def** stevilo\_samoglasnikov(niz):

stevilo = 0

**for** znak **in** niz:

**if** znak.lower() **in** 'aeiou':

stevilo += 1

**return** stevilo

stevilo\_samoglasnikov('Uvod v programiranje')

7

Če se želimo sprehoditi po številih, uporabimo funkcijo range. Ta zaporedoma vrača vsa števila v danem razponu:

**for** x **in** range(5, 10):

print(x \*\* 2)

25

36

49

64

81

Vidimo, da tako kot rezine tudi range ne doseže zgornje meje. Če funkciji range podamo en argument, bo začela šteti z 0, če pa ji podamo še tretji argument, ga bo uporabila za velikost koraka (kot pri rezinah).

**for** x **in** range(5):

print(x)

0

1

2

3

4

**for** x **in** range(5, 10, 2):

print(x)

5

7

9

S pomočjo funkcije range bi fakulteto torej napisali kot:

**def** fakulteta(n):

*"""Vrne fakulteto naravnega števila n."""*

produkt = 1

**for** i **in** range(1, n + 1):

produkt \*= i

**return** produkt

fakulteta(20)

2432902008176640000

Stavki break, continue in pass

V zankah lahko uporabimo tudi posebne ukaze, ki spreminjajo običajen potek programa. Stavek break prekine trenutno zanko. Na primer:

**for** n **in** range(1, 5):

print(n)

**if** n == 2 **or** n == 3:

**break**

print('x')

1

x

2

izmenično izpisuje števila od 1 do 4 ter znak x. V trenutku, ko pride do števila 2, ki zadošča pogoju n == 2 or n == 3, izvajanje zanke v celoti prekine in preneha z izpisovanjem.

Primer je napisan za zanko for, vendar enako deluje tudi za zanko while.

Stavek continue zanke ne ustavi, temveč le preskoči preostanek trenutnega obhoda zanke in gre nazaj na začetek z naslednjo vrednostjo. Na primer:

**for** n **in** range(1, 5):

print(n)

**if** n == 2 **or** n == 3:

**continue**

print('x')

1

x

2

3

4

x

pri številih 2 in 3, ki zadoščata pogoju, preskoči izpis znaka x, ki bi moral slediti.

Tudi stavek continue deluje tako za zanko for kot za zanko while.

Stavek pass pa ne stori ničesar. Na primer:

**for** n **in** range(1, 5):

print(n)

**if** n == 2 **or** n == 3:

**pass**

print('x')

1

x

2

x

3

x

4

x

pri številih 2 in 3, ki zadoščata pogoju, vstopi v pogojni stavek, vendar tam ne stori ničesar, zato je izpis enak, kakor bi bil za program brez pogojnega stavka.

Stavek pass lahko uporabljamo kjerkoli v Pythonu, ne le v zankah. Najpogosteje ga uporabimo takrat, kadar Python zahteva, da napišemo vsaj en ukaz, vendar ne želimo storiti ničesar. Recimo, da imamo program:

**for** x **in** range(100):

**if** x % 2 == 0:

print(x, 'je sod')

**else**:

print(x, 'je lih')

in za trenutek želimo izklopiti izpisovanje sodih števil. Če bi napisali le

**for** x **in** range(100):

**if** x % 2 == 0:

*# print(x, 'je sod')*

**else**:

print(x, 'je lih')

bi se Python pritožil, da je prva veja pogojnega stavka prazna, saj komentarje ignorira. Seveda bi lahko celoten program preuredili v

**for** x **in** range(100):

**if** x % 2 != 0:

print(x, 'je lih')

vendar tega ne želimo (sploh pri večjih programih). S pomočjo stavka pass pa lahko napišemo

**for** x **in** range(100):

**if** x % 2 == 0:

*# print(x, 'je sod')*

**pass**

**else**:

print(x, 'je lih')

Tudi če se odločimo, da bi zopet vklopili izpisovanje, lahko stavek pass pustimo v kodi, ker ne stori ničesar.

Seznami & nabori

Če želimo delati z zaporedjem podatkov, uporabimo sezname. Sename pišemo v oglatih oklepajih, med katerimi napišemo vrednosti, ločene z vejicami, na primer [10, 20, 30] je seznam, ki vsebuje tri števila, [] pa prazen seznam. Če želimo, lahko vejico pišemo tudi za zadnjim elementom. Seznami so lahko tudi gnezdeni. Na primer, matriko bi predstavili s seznamom seznamov [[1, 2, 3], [4, 5, 6], [7, 8, 9]]. V sezname lahko spravimo vrednosti različnih tipov, na primer: [1, True, [2, 5], "Niz", 3.14] Vendar običajno sezname uporabimo za predstavitev homogene zbirke podatkov, torej da so vse vrednosti istega tipa.

Osnovne operacije na seznamih

Precej operacij na seznamih je enakih kot na nizih. Sezname lahko stikamo z operacijo + in množimo s celimi števili:

[**True**, **False**] + [**True**]

[True, False, True]

3 \* ['x', 'y']

['x', 'y', 'x', 'y', 'x', 'y']

Dolžino seznama dobimo s funkcijo len:

len([100, 200, 300])

3

len([])

0

Sezname med seboj primerjamo leksikografsko: najprej prvi par, če sta ta dva enaka, naslednji par in tako naprej.

[1, 100, 10000] < [2, 0, 0]

True

Prav tako imamo na voljo predikata in in not in, s katerima ugotovimo, ali se nek element pojavlja v seznamu:

prastevila = [2, 3, 5, 7, 11, 13, 17]

5 **in** prastevila

True

15 **not** **in** prastevila

True

Predikat in na seznamih se razlikuje od tistega na nizih v tem, da preverja le pojavitev elementov, ne pa podseznamov:

[3, 5, 7] **in** prastevila

False

Na primer:

**def** stevilo\_dni(mesec, leto):

**if** mesec == 2:

**return** 29 **if** je\_prestopno(leto) **else** 28

**elif** mesec **in** [4, 6, 9, 11]:

**return** 30

**else**:

**return** 31

Indeksiranje in rezine na seznamih delujejo tako kot na nizih:

prastevila = [2, 3, 5, 7, 11, 13, 17]

prastevila[2]

5

prastevila[-2]

13

prastevila[len(prastevila) - 2]

13

prastevila[2:-2]

[5, 7, 11]

prastevila[::2]

[2, 5, 11, 17]

Če imamo gnezdene sezname, do elementov dostopamo z gnezdenimi indeksi:

mat = [[1, 0, 0], [0, -1, 2], [3, 1, 5]]

mat[1][-1]

2

Na primer, sled matrike bi lahko izračunali kot:

**def** sled(matrika):

*"""Izračuna sled dane matrike."""*

vsota\_diagonalnih = 0

**for** i **in** range(len(matrika)):

vsota\_diagonalnih += matrika[i][i]

**return** vsota\_diagonalnih

mat = [[5]]

sled(mat)

5

Sledi pa nikakor ne bomo izračunali na sledeči (pri študentih dostikrat videni) način:

**def** grozna\_sled(matrika):

*"""Na popolnoma napačen izračuna sled dane matrike."""*

vsota\_diagonalnih = 0

**for** i **in** range(len(matrika)):

**for** j **in** range(len(matrika)):

**if** i == j:

vsota\_diagonalnih += matrika[i][j]

**return** vsota\_diagonalnih

Funkcija sled matrike sicer izračuna pravilno, vendar na izjemno potraten način, saj se sprehodi čez celotno matriko, ne le čez diagonalne elemente. Na primer, pri matriki velikosti 1000×1000 bi druga funkcija pregledala tisočkrat več elementov (in posledično porabila tisočkrat več časa).

Zanke na seznamih

Tako kot se lahko z zanko for sprehodimo po vseh znakih v nizu, se lahko z njo sprehodimo tudi po vseh elementih danega seznama:

**for** x **in** [10, 20, 30]:

print(x)

10

20

30

Na primer, vsoto vseh elementov seznama bi definirali kot:

**def** vsota\_elementov(seznam):

*"""Vrne vsoto elementov v danem seznamu."""*

vsota = 0

**for** element **in** seznam:

vsota += element

**return** vsota

vsota\_elementov([10, 2, 4000, 300])

4312

Največji element v danem seznamu lahko poiščemo tako, da zaporedoma vsak element seznama primerjamo z do sedaj največjim videnim elementom. Če je trenutni element večji, do sedaj največji element popravimo. Ko pregledamo vse elemente v seznamu, je do sedaj največji element tudi na splošno največji element. Edina stvar, na katero moramo še paziti, je ta, da na začetku izberemo ustrezen največji element. Tu imamo dve dobri izbiri. (Slaba izbira bi bila, da bi za največji do zdaj viden element vzeli neko dovolj majhno število, na primer 0 ali -9999999 – ta izbira je očitno napačna!) Prva dobra izbira je kar prvi element v seznamu, pri čemer moramo potem poprej preveriti še to, da je seznam neprazen:

**def** najvecji\_element(seznam):

*"""Vrne največji element v danem seznamu. Če ga ni, vrne None"""*

**if** len(seznam) == 0:

**return**

najvecji\_do\_zdaj = seznam[0]

**for** element **in** seznam:

**if** element > najvecji\_do\_zdaj:

najvecji\_do\_zdaj = element

**return** najvecji\_do\_zdaj

najvecji\_element([10, 2, 4000, 300])

4000

Seveda lahko uporabimo tudi vgrajene funkcije:

sum([10, 2, 4000, 300])

4312

min([10, 2, 4000, 300])

2

max([10, 2, 4000, 300])

4000

Spreminjanje seznamov

Za razliko od nizov lahko vrednosti v seznamih tudi spreminjamo:

sez = [10, 20, 30]

sez[1]

20

sez[1] = 40

sez

[10, 40, 30]

Zamenjamo lahko tudi celotno **rezino**

sez = [10, 20, 30, 40]

sez[1:3]

[20, 30]

[20, 30]

sez[1:3] = [0, 0, 0]

sez

[10, 0, 0, 0, 40]

Če nadomestna rezina ni enake dolžine kot prvotna, se seznam ustrezno skrajša ali podaljša. Nadomestimo tudi prazno rezino, s čimer nove elemente vrinemo v seznam. Nadomeščanje prazne rezine ni isto kot nadomeščanje elementa z istim indeksom kot rezina:

sez = [10, 20, 30, 40]

sez[2:2] = [0, 0, 0]

sez

[10, 20, 0, 0, 0, 30, 40]

sez = [10, 20, 30, 40]

sez[2] = [0, 0, 0]

sez

[10, 20, [0, 0, 0], 40]

Tako elemente kot rezine lahko tudi **brišemo**

sez = [10, 20, 30, 40, 50]

**del** sez[1]

sez

[10, 30, 40, 50]

sez = [10, 20, 30, 40, 50]

**del** sez[2:4]

Pri spreminjanju seznamov je treba biti previden, saj ne deluje tako, kot smo navajeni pri spreminjanju vrednosti spremenljivk. Na primer, če pišemo

a = 5

b = a

a = 0

b

5

Vidimo, da se vrednost spremenljivke b ni spremenila, ko smo spremenili a, saj smo jo v drugi vrstici nastavili na število 5. Pri seznamih je stvar malo drugačna. Če pišemo

a = [1, 1, 1]

b = a

a = [2, 2, 2]

b

[1, 1, 1]

so stvari še vedno take, kot bi jih pričakovali. Vrednost b smo nastavili na isti seznam kot a, vendar smo potem rekli, da naj bo v a shranjen drugačen seznam, s čimer na vrednost v b nismo vplivali. Če pa pišemo

a = [1, 1, 1]

b = a

a[1] = 2

b

[1, 2, 1]

se je s tem, ko smo spremenili a, spremenil tudi b. Kaj se je zgodilo? Ko smo napisali b = a, smo povedali, naj bo v b shranjen isti seznam kot a. In z a[1] = 2 smo povedali, naj se na mesto 1 v seznamu, shranjenem v a, zapiše 2. Ker je v b shranjen isti (ne le enak) seznam kot v a, je s tem tudi seznam v b drugačen. Da je zmeda večja, += ni samo okrajšava za = in +, kot smo navajeni pri številih:

a = [1, 1, 1]

b = a

a = a + [2, 2, 2]

b

[1, 1, 1]

a = [1, 1, 1]

b = a

a += [2, 2, 2]

b

[1, 1, 1, 2, 2, 2]

V prvem primeru izračunamo nov seznam a + [2, 2, 2] in ga shranimo pod ime a, s čimer se b ne spremeni. V drugem primeru pa je += operacija, ki razširi obstoječi seznam. Ker gre pri a in b za isti seznam, se spremeni tudi seznam v b.

Pogosta past, v katero se na začetku ujamemo zaradi spremenljivosti seznamov, je izračun identične matrike. Vemo že, da lahko v Pythonu seznam pomnožimo s številom:

3 \* [0]

[0, 0, 0]

To nam da idejo, da bi lahko na isti način izračunali ničelno matriko:

3 \* [3 \* [0]]

[[0, 0, 0], [0, 0, 0], [0, 0, 0]]

Izračun je videti pravilen, vendar vse tri vrstice te matrike kažejo na isti seznam. To je tako, kot če bi pisali:

vrstica = [0, 0, 0]

matrika = [vrstica, vrstica, vrstica]

Poskusimo iz te matrike dobiti identično matriko tako, da po diagonali nastavimo enice. Najprej nastavimo prvi element v prvi vrstici:

matrika[0][0] = 1

matrika

[[1, 0, 0], [1, 0, 0], [1, 0, 0]]

Kaj se je zgodilo? Ker druga in tretja vrstica kažeta na isti seznam kot prva, smo tudi v njima prvi element popravili na 1. Če sedaj nastavimo še drugi element v drugi vrstici in tretjega v tretji vrstici se zgodba ponovi:

matrika[1][1] = 1

matrika[2][2] = 1

matrika

[[1, 1, 1], [1, 1, 1], [1, 1, 1]]

Če želimo identično matriko izračunati na pravilen način, moramo za predstavitev vsake vrstice podati svoj seznam, zato ne moremo uporabiti le pomnoževanja seznamov.

Vgrajene metode na seznamih

Kot pri nizih za večino pogosto uporabljanih stvari na seznamih obstajajo že vgrajene metode. Te povečini ne vračajo ničesar, temveč le spremenijo dani seznam. Izjemi sta metodi index in count, ki vrneta vrednost in seznam pustita pri miru, ter metoda pop ki tako spremeni seznam kot vrne vrednost. Vse metode so naštete v [uradni dokumentaciji](https://docs.python.org/3/tutorial/datastructures.html#more-on-lists), najbolj osnovne izmed njih pa so:

* sez.append(x) na konec seznama sez doda element x.

sez = [10, 20, 30]

sez.append(40)

sez

[10, 20, 30, 40]

Metodo append pogosto uporabljamo za izračun seznama ustreznih elementov. To storimo tako, da ustvarimo prazen seznam, nato pa vanj z metodo append dodamo vsak ustrezen element. To je podoben postopek kot pri izračunu vsote ustreznih elementov, kjer smo ustvarili spremenljivko z začetno vrednostjo 0, nato pa ji prištevali ustrezne elemente.

* sez.extend(sez2) na konec seznama sez doda vse elemente iz seznama sez2.

sez = [10, 20, 30]

sez.extend([40, 50])

sez

[10, 20, 30, 40, 50]

Okrajšava +=, ki smo jo videli prej, pokliče natanko metodo extend.

sez = [10, 20, 30]

sez += [40, 50]

sez

[10, 20, 30, 40, 50]

* sez.insert(i, x) pred element na mestu i v seznamu sez vstavi element x.

sez = [10, 20, 30]

sez.insert(1, 0)

sez

[10, 0, 20, 30]

[10, 0, 20, 30]

* sez.remove(x) iz seznama sez odstrani prvo pojavitev vrednosti x.

sez = [1, 2, 1, 2, 3]

sez.remove(2)

sez

[1, 1, 2, 3]

* sez.pop(i=-1) vrne element na mestu i v seznamu sez in odstrani ta element iz seznama. Če indeksa i ne podamo, metoda odstrani zadnji element.

sez = [10, 20, 30]

sez.pop()

30

sez

[10, 20]

* sez.clear() iz seznama sez pobriše vse elemente.

sez = [1, 2, 1, 2, 3]

sez.clear()

sez

[]

* sez.index(x) vrne prvo mesto, na katerem se v seznamu sez nahaja vrednost x.
* sez.count(x) vrne število pojavitev vrednosti x v seznamu sez.
* sez.sort(key=None, reverse=False) na mestu uredi seznam glede na vrednosti funkcije key. Če parameter reverse nastavimo na True, bo seznam urejen padajoče.

sez = [4, 1, 8, 2, 16]

sez.sort()

sez

[1, 2, 4, 8, 16]

sez = [4, 1, 8, 2, 16]

sorted(sez)

[1, 2, 4, 8, 16]

[1, 2, 4, 8, 16]

* sez.reverse() obrne seznam sez na glavo.

sez = [1, 2, 4, 8, 16]

sez.reverse()

sez

[16, 8, 4, 2, 1]

TODO: primerjava z rezinami

sez = [1, 2, 4, 8, 16]

sez[::-1]

sez

[1, 2, 4, 8, 16]

Izpeljani seznami

Sezname dostikrat lepše zgradimo z *izpeljanimi seznami*, ki so oblike [izraz for spremenljivka in mozne\_vrednosti]. Podobno kot v matematiki množice pišemo kot 2⋅n∣n∈1,…,9, lahko v Pythonu napišemo seznam:

[2 \* n **for** n **in** range(1, 10)]

[2, 4, 6, 8, 10, 12, 14, 16, 18]

ali pa na primer:

potence = [2 \*\* n **for** n **in** range(10)]

potence

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

[n - 1 **for** n **in** potence]

[0, 1, 3, 7, 15, 31, 63, 127, 255, 511]

[int(stevka) **for** stevka **in** str(3141592)]

[3, 1, 4, 1, 5, 9, 2]

Če želimo, lahko v izpeljani seznamih oblike [izraz for spremenljivka in mozne\_vrednosti if pogoj] s pogojem določimo, katere elemente želimo:

[2 \* n **for** n **in** range(1, 10) **if** n % 3 == 1]

[2, 8, 14]

**def** vsota\_pozitivnih\_elementov(seznam):

*"""Vrne seznam vseh pozitivnih elementov danega seznama."""*

**return** sum([element **for** element **in** seznam **if** element > 0])

vsota\_pozitivnih\_elementov([10, -20, 50])

60

Zanke v izpeljanih seznamih so lahko gnezdene na dva načina. Prvi je, da kot elemente izpeljanega seznama spet podamo izpeljane sezname. Na primer, identično matriko lahko naredimo kot:

**def** identicna\_matrika(n):

**return** [[1 **if** i == j **else** 0 **for** j **in** range(n)] **for** i **in** range(n)]

Lahko pa v izpeljanem seznamu naštejemo več zank, pri čemer ob vsakem koraku v obhodu prve spremenljivke naredimo obhod vseh preostalih:

**def** razlicni\_pari(n):

**return** [(i, j) **for** i **in** range(n) **for** j **in** range(i, n)]

razlicni\_pari(5)

[(0, 0),

(0, 1),

(0, 2),

(0, 3),

(0, 4),

(1, 1),

(1, 2),

(1, 3),

(1, 4),

(2, 2),

(2, 3),

(2, 4),

(3, 3),

(3, 4),

(4, 4)]

Izpeljani seznami pa niso vedno rešitev, tudi če so krajši. Na primer, če je vsaka vrednost v seznamu odvisna od prejšnjih, jih ne moremo uporabiti.

Nabori

Nabori so prav tako kot seznami sestavljeni iz več elementov, pišemo pa jih med običajne oklepaje: (1, 2, 3). Nabor z enim elementom pišemo kot (1, ) (razmislite, zakaj ga ne pišemo kot (1)). Namen naborov je hranjenje heterogenih podatkov. To pomeni, da imajo elementi na različnih mestih različne pomene. Na primer, če vemo da vsako število v seznamu [103, 111, 98, 106] predstavlja rezultat neke meritve, bi podobno sklepali tudi, če bi v seznam dodali še kakšno drugo številko. Pri datumu (25, 6, 1991) pa imajo različne komponente različne pomene: prva predstavlja dan, druga mesec in tretje leto. Če bi dodali še četrto številko, bi morali povedati, kaj pomeni. Python nam seveda omogoča, da bi meritve zapisali kot (103, 111, 98, 106), datum pa kot [25, 6, 1991], vendar bomo sezname uporabljali za **poljubno število podatkov z enakim pomenom**, nabore pa za **fiksno število podatkov z različnimi pomeni**.

S tem namenom so nabori v Pythonu nespremenljivi:

x = (1, 1, 1)

x[2] = 2

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

/tmp/ipykernel\_1848/3425797582.py **in** <module>

1 x = (1, 1, 1)

----> 2 x[2] = 2

TypeError: 'tuple' object does not support item assignment

Sicer pa za nabore veljajo podobne lastnosti kot za sezname: lahko jih stikamo in množimo; lahko izračunamo njihovo vsoto, minimum, maksimum in dolžino; s predikatom in lahko pogledamo, ali vsebujejo dani element; lahko jih indeksiramo in delamo rezine; po njih se lahko sprehodimo z zanko for; od metod pa sta na voljo le count in index, saj ti dve edini ne spreminjata ničesar.

(**True**, **False**) + (**True**,)

(True, False, True)

3 \* ('x', 'y')

('x', 'y', 'x', 'y', 'x', 'y')

5 **in** (1, 2, 3)

False

(1, 2, 5, 10, 20, 50)[3]

10

(1, 2, 5, 10, 20, 50)[3:]

(10, 20, 50)

Če imamo nabor, ga lahko razstavimo na posamezne spremenljivke, na primer

datum = (25, 6, 1991)

dan, mesec, leto = datum

f'*{*dan*}*. *{*mesec*}*. *{*leto*}*'

'25. 6. 1991'

V resnici gre pri hkratnem prireditvenem stavku kot je

x, y = 10, 20

za to, da najprej naredimo torej nabor z dvema komponentama oziroma par (10, 20), nato pa ga razstavimo na dve spremenljivki.

Funkciji enumerate in zip

Dostikrat želimo hkrati dostopati do elementov seznama in njihovih indeksov.

Predstavimo polinome s seznamom koeficientov, urejenim od prostega proti vodilnemu členu. Polinom 3−x2 bi tako predstavili s seznamom [3, 0, -1]. Pri izračunu vrednosti polinoma želimo hkrati dostopati tako do koeficientov kot do njihovih indeksov, ki ustrezajo potenci. To lahko storimo na več načinov. Lahko se vozimo po indeksih in prek njih dostopamo do koeficientov:

**def** vrednost\_polinoma(polinom, x):

vsota = 0

**for** i **in** range(len(polinom)):

koef = polinom[i]

vsota += koef \* x \*\* i

**return** vsota

vrednost\_polinoma([3, 0, 1], 1)

4

vrednost\_polinoma([3, 0, 1], 2)

7

Lahko se vozimo po koeficientih in hkrati povečujemo števec indeksa:

**def** vrednost\_polinoma(polinom, x):

vsota = 0

i = 0

**for** koef **in** polinom:

vsota += koef \* x \*\* i

i += 1

**return** vsota

**Warning**

Paziti moramo, da indeksa ne računamo s pomočjo metode index, saj ta vrne indeks prve pojavitve iskane vrednosti, kar je narobe (pa še počasno):

```{code-cell}

def napacna\_vrednost\_polinoma(polinom, x):

vsota = 0

for koef in polinom:

i = polinom.index(koef)

vsota += koef \* x \*\* i

return vsota

```{code-cell}

vrednost\_polinoma([0, 2, 0, 2], 3)

```

```{code-cell}

napacna\_vrednost\_polinoma([0, 2, 0, 2], 3)

```

Ker je v spodnjem klicu funkcije metoda `index` za indeks prve pojavitve vrednosti 2 obakrat vrnila 1, je funkcija vrnila $2 \cdot 3^1 + 2 \cdot 3^1 = 6$ namesto $2 \cdot 3^1 + 2 \cdot 3^3 = 60$.

Najbolj enostavno pa je, da uporabimo funkcijo `enumerate`, ki zaporedoma vrača pare, v katerih so druge komponente vrednosti danega seznama, prve komponente pa njihovi indeksi:

```{code-cell}

for x in enumerate('abc'):

print(x)

Pare, ki nam jih podaja enumerate lahko v dve spremenljivki razstavimo tudi v zanki for:

**for** i, x **in** enumerate('abc'):

print(i, x)

0 a

1 b

2 c

S pomočjo funkcije enumerate lahko vrednost polinoma izračunamo kot:

**def** vrednost\_polinoma(polinom, x):

vsota = 0

**for** i, koef **in** enumerate(polinom):

vsota += koef \* x \*\* i

**return** vsota

ali še krajše z izpeljanim seznamom:

**def** vrednost\_polinoma(polinom, x):

**return** sum([koef \* x \*\* i **for** i, koef **in** enumerate(polinom)])

Podobno kot enumerate deluje funkcija zip, ki sprejme več seznamov, vrne pa zaporedje naborov istoležnih elementov:

**for** x **in** zip('xyz', [10, 20, 30], [4, 5, 6]):

print(x)

('x', 10, 4)

('y', 20, 5)

('z', 30, 6)

Funkciji se reče zip, ker združuje elemente različnih seznamov tako kot zadrga. Vrnjeno zaporedje ima toliko elementov, kot najkrajši argument funkcije:

**for** x **in** zip('xyz', [10, 20, 30, 40]):

print(x)

('x', 10)

('y', 20)

('z', 30)

S pomočjo funkcije zip lahko enostavno izračunamo skalarni produkt:

**def** skalarni\_produkt(vektor1, vektor2):

*"""Vrne skalarni produkt dveh vektorjev iste dimenzije."""*

**assert** len(vektor1) == len(vektor2)

vsota = 0

**for** x1, x2 **in** zip(vektor1, vektor2):

vsota += x1 \* x2

**return** vsota

ali kot:

**def** skalarni\_produkt(vektor1, vektor2):

*"""Vrne skalarni produkt dveh vektorjev iste dimenzije."""*

**assert** len(vektor1) == len(vektor2)

**return** sum([x1 \* x2 **for** x1, x2 **in** zip(vektor1, vektor2)])

skalarni\_produkt([1, -2, 5], [-2, 5, 2])

-2