INTERAKTIVNI MEDIJI 1

Izr. prof. dr. Aleš Hladnik

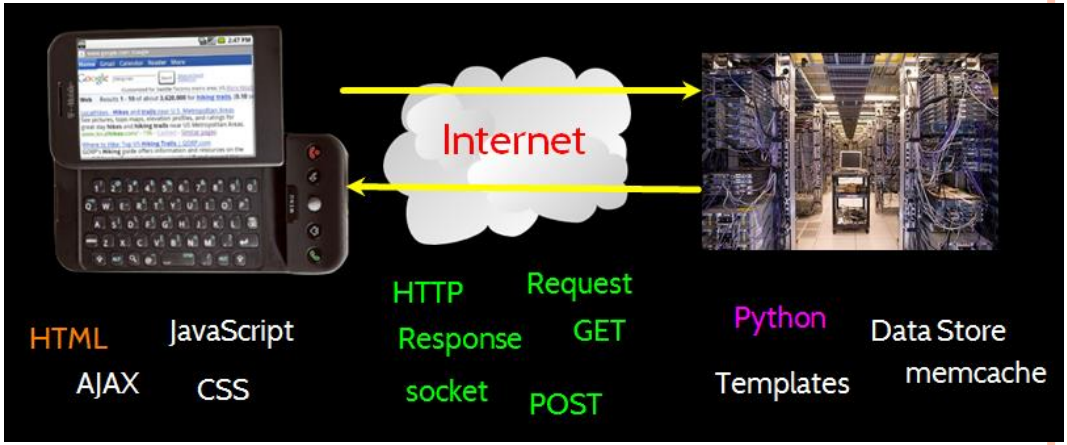
Šolsko leto 2023/2024

VSEBINA PREDMETA

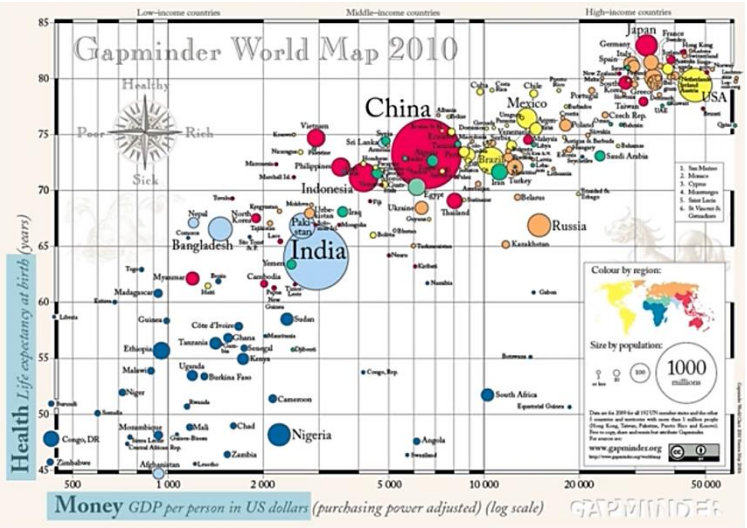
Python - osnove

sintaksa in semantika
spremenljivke
podatkovni tipi
izrazi
operatorji
zanke (IF) in iteracije (FOR)
programi
funkcije
nizi
datoteke
seznami
knjižnice in moduli
slovarji

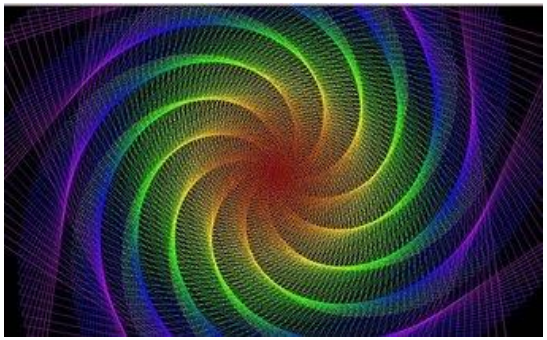
Dostopanje do spletnih podatkov in njihovo luščenje (*web scraping*)

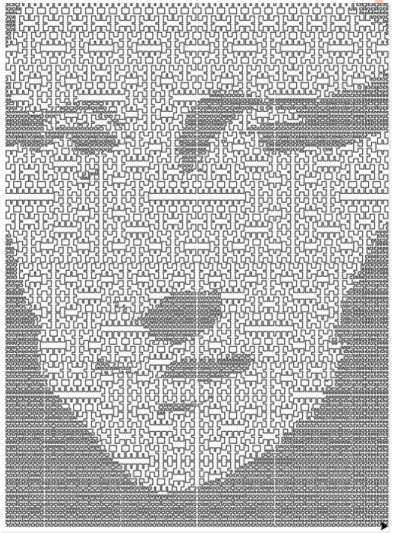


Vizualizacija in analiza (velikih količin) podatkov



Generativna umetnost in računalniške igre





... in še več !

1: Osnove programiranja v Pythonu

Prirejeno po e-učbeniku za informatiko v gimnaziji, 2015:
https://lusy.fri.uni-lj.si/ucbenik/book/index.html

Osnovni koncepti programiranja

Računalnik je naprava, ki nam omogoča reševanje najrazličnejših problemov: od najpreprostejšega računanja do krmiljenja vesoljske rakete. Z računalnikom danes pravzaprav počnemo mnogo več, kot samo rešujemo probleme: urejamo besedila, iščemo najrazličnejše informacije, komuniciramo, se izobražujemo, si dopisujemo, se zabavamo, telefoniramo (da, tudi napravice, ki jih nosite v žepu, so dejansko računalniki!), ... Težko je najti področje, kamor računalniki še niso posegli.

Kako lahko računalnik pripravimo do tega, da zmore početi vse naštete reči (in še mnogo več)? Tako, da zanj napišemo ustrezne **programe**. Program ni nič drugega kot zaporedje navodil za reševanje določenega problema. Računalnik **izvršuje** programe, ki mu jih napišemo. Z besedo **programiranje** pa označujemo dejavnost pisanja programov. Zaradi razširjenosti računalnikov in računalniških naprav je programiranje danes ena od ključnih veščin.

Računalniki so zanesljivi: vsak program bodo vedno izvedli na enak način. V nasprotju z nami se zlepa ne utrudijo. Vendar pa so tudi razmeroma »neumni«, saj zahtevajo izjemno natančna navodila.

Ljudje se med seboj sporazumevamo v različnih jezikih. Tudi za podajanje navodil računalniku — torej programiranje — uporabljamo posebne jezike. Imenujemo jih **programski jeziki**. Tako kot naravnih (človeških) jezikov je tudi programskih cela vrsta. V tem učbeniku bomo programirali v jeziku **python**, saj je sodoben programski jezik, primeren tako za začetnike kot za poklicne programerje. Vendar pa večina programskih jezikov (npr. java, C++, C#, pascal ipd.) temelji na podobnih idejah kot python. Zato se ti v prihodnje po potrebi ne bo težko naučiti še kakšnega programskega jezika.

Namestitev Pythona ver. 3

- Python3 na OS Windows namestimo tako, da na naslovu https://www.python.org/downloads/windows/ izberemo najnovejši namestitveni paket (trenutno 3.12.2), poženemo namestitev ter sledimo navodilom
- Na voljo so tudi paketi za druge operacijske sisteme – MacOS, Linux/UNIX, itd.
- Po namestitvi zaženemo Pythonov **interaktivni tolmač IDLE** (Integrated DeveLopment Environment) oziroma njegovo lupino *(shell)*
- Spletna pomoč za Python3: dokumentacija na https://docs.python.org/3/
- Python2 in Python3 se med seboj precej razlikujeta, zato odsvetujemo uporabo Pythona ver. 2 !

Uporaba interaktivnega tolmača

Zažene se pythonov **interaktivni tolmač**, program, ki sproti bere naše ukaze in jih izvršuje. Prikaže se pozivnik (trije znaki `>>>` s sledečim presledkom), ki čaka na vnos ukaza. Interaktivni tolmač lahko uporabljamo kot kalkulator:

```
>>> 3 + 4
7
>>> 7 - 12
-5
```

Za množenje uporabljamo znak `*`:

```
>>> 5 * 2
10
```

Python pozna kar dve različni vrsti deljenja, realnoštevilsko (znak `/`) in celoštevilsko (znak `//`). Realnoštevilsko deljenje deluje tako, kot smo vajeni na kalkulatorjih: rezultat izračuna na določeno število decimalnih mest natančno. Celoštevilsko deljenje pa števke za decimalno piko odreže in tako ohrani samo celi del količnika.

```
>>> 27 / 4
6.75
>>> 27 // 4
6
```

Pogosto nam bo prišel prav tudi ostanek pri celoštevilskem deljenju, ki ga v pythonu označimo z znakom `%`:

```
>>> 27 % 4
3
```

Včasih nam bo prišlo prav tudi potenciranje (znak `**`), na primer 5^2:

```
>>> 5 ** 2
25
```

IDLE: interaktivni tolmač (lupina, *shell*)

Spisek bližnjic: *Options > Configure IDLE > Keys*

Vrstni red izvajanja operacij

Množenje, deljenje in računanje ostanka imajo prednost pred seštevanjem in odštevanjem, operatorji iz iste skupine pa se — tako kot v matematiki — obravnavajo od leve proti desni. Z oklepaji lahko spreminjamo vrstni red izvajanja operacij:

```
>>> 3 + 4 * 5
23
>>> (3 + 4) * 5
35
```

Vaje

Izračunaj vrednosti sledečih izrazov najprej na papirju, nato pa rezultate preveri še s pythonovim interaktivnim tolmačem:

- `1 + 2 * (3 - 4 * (5 - 6))`
- `23 // 6 + 23 % 6`
- `23 // (6 + 23) % 6`
- `32 // 2 * 4`
- `32 // 2 % 4`
- `32 // (2 ** 4)`
- `32 // (2 % 4)`
- `32 // (2 // 4)`

Izraz, spremenljivka, prireditev

Doslej smo pythonov interaktivni tolmač uporabljali kot navaden kalkulator. Vanj smo vtipkovali matematične **izraze**, tolmač pa je računal njihove **vrednosti**. S pojmom **izraz** v programiranju označujemo zaporedje simbolov, ki ima neko vrednost. Kot bomo videli kasneje, izraz ni nujno matematičen, pa tudi njegova vrednost ni nujno število.

Vrednosti izrazov lahko tudi poimenujemo in jih kasneje uporabimo. Na ta način se izognemu večkratnemu vtipkovanju istih izrazov.

```
>>> a = 4 + 5
>>> b = 11 - 9
>>> a
9
>>> a * b
18
```

V prvi vrstici smo uvedli **spremenljivko** z imenom `a` in ji **priredili** vrednost izraza `4 + 5` (torej 9). V drugi vrstici smo uvedli spremenljivko `b` in ji priredili vrednost izraza `11 - 9` (torej 2). V tretji vrstici smo vrednost spremenljivke `a` izpisali, v četrti pa smo izpisali vrednost zmnožka spremenljivk `a` in `b`.

Spremenljivka je poimenovan prostor v pomnilniku, ki lahko hrani poljubno vrednost. Z zapisom

```
>>> spremenljivka = izraz
```

spremenljivki na levi strani enačaja **priredimo** vrednost izraza na desni strani enačaja. Prireditev se izvrši tako, da se najprej izračuna izraz, nato pa se njegova vrednost vpiše v spremenljivko.

Izraz, spremenljivka, prireditev (nad.)

Oglejmo si še en primer:

```
>>> x = 5
>>> x = x + 1
>>> x
6
```

Zapis `x = x + 1` se zdi nenavaden. Kako je lahko `x` enak `x + 1`? Vendar ga ne smemo brati po matematično! Ko pythonov tolmač prebere zapis `x = x + 1`, najprej izračuna vrednost izraza na desni strani enačaja (vrednost izraza `x + 1` znaša 6), nato pa dobljeno vrednost ponovno vpiše v spremenljivko `x`. Spremenljivka `x` ima tako po izvedbi gornjih vrstic vrednost `6`.

Ime spremenljivke je lahko sestavljeno iz poljubnega zaporedja malih in velikih črk, števk in podčrtajev (znakov `_`), začeti pa se mora s črko ali podčrtajem.

Vaja

Označi veljavna imena spremenljivk:

- [ ] `abc123`
- [ ] `obseg_pravokotnika`
- [ ] `1krog`
- [ ] `polmerKroga`

Prireditveni stavek

Zapis ***spremenljivka = izraz*** imenujemo **prireditveni stavek**. Prireditveni stavek je eden od več različnih *stavkov*, ki jih ponuja python. Stavek je osnovni element programskega jezika, ki »nekaj naredi« (ima nek učinek).

Vaja

Kakšne vrednosti imajo spremenljivke `a`, `b` in `c` po izvedbi sledečih stavkov? Nalogo najprej reši na papir, nato pa rezultate preizkusi s pythonovim interaktivnim tolmačem.

```
>>> a = 5
>>> b = a + 1
>>> b = b * a
>>> c = b // 3
>>> c = a + b + c
```

Ko se spremenljivka prvič pojavi na levi strani prireditvenega stavka, jo **definiramo**. Dokler spremenljivka ni definirana, je seveda ne moremo uporabiti. Tolmač nas opozori, da spremenljivka `ploscina` ni definirana:

```
>>> ploscina
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ploscina' is not defined
```

Vaja

Razvrsti sledeča navodila v smiselno zaporedje:

```
>>> ploscina
>>> ploscina = a * b
>>> b = 4
>>> a = 3
```

ŠTEVILSKA PODATKOVNA TIPA

Poženimo interaktivni tolmač in izvedimo sledeča stavka:

```
>>> a = 10 // 2
>>> b = 10 / 2
```

Ker je število 10 deljivo s številom 2, bi moral biti rezultat obakrat enak 5, kajne?

```
>>> a
5
>>> b
5.0
```

type(a) = int

type(b) = float

Hm, zakaj python števila 5.0 ne zapiše enostavno kot 5? Mar ni to dvoje enako? Odgovor se glasi: da in ne. Da, matematično sta števili enaki. Sta pa tudi različnih **tipov**. Število 5 je **celoštevilskega** tipa, število 5.0 pa **realnoštevilskega**. Python ločuje med celimi in realnimi števili. Realna števila vedno zapisuje z decimalno piko, tudi če je decimalni del enak 0.

Če v izrazu nastopa vsaj eno realno število ali pa realnoštevilsko deljenje, bo rezultat realno število, v nasprotnem pa celo:

```
>>> 3 + 5.0 + 7
15.0
>>> 3 + 10 // 2
8
>>> 3 + 10 / 2
8.0
```

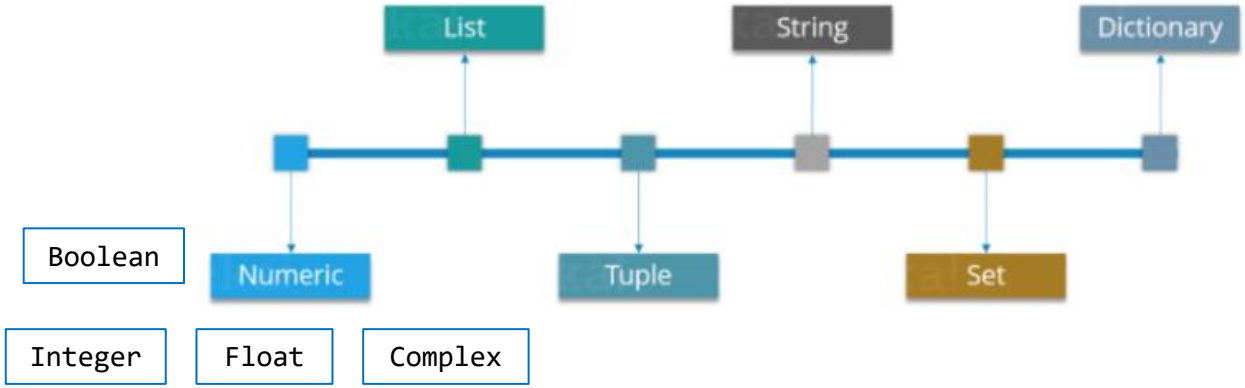
Številska podatkovna tipa (nad.)

Realno število je možno pretvoriti v celo in obratno. Prvo pretvorbo izvršimo z operatorjem `int`, drugo pa z operatorjem `float`.

```
>>> int(3.0)
3
>>> int(3.9)
3
>>> int(-3.9)
-3
>>> float(-6)
-6.0
```

Operator `int` pretvori realno število v celo enostavno tako, da odreže decimalni del.

Podatkovni tipi v Pythonu



Znakovni podatkovni tip

Poleg števil bomo pogosto delali tudi z **nizi**. Niz predstavlja poljubno zaporedje znakov. Zapišemo ga znotraj enojnih ali dvojnih navednic. Primeri nizov so `'Dober dan!'`, `"leto 2014"`, `'+-*/'` in `"123"`. Z nizi sicer ne moremo računati, a kot bomo videli kasneje, so pri programiranju prav tako nepogrešljivi.

Tudi spremenljivke imajo svoj tip. Na primer, po izvedbi stavkov

```
>>> a = 5
>>> b = 7.3
>>> obseg = 2 * a + b
>>> dejstvo = 'obožujem python'
```

type(dejstvo) = str

je spremenljivka `a` celoštevilskega tipa, spremenljivki `b` in `obseg` realnoštevilskega, spremenljivka `dejstvo` pa je niz.

V pythonu lahko tip vrednosti ali spremenljivke ugotovimo z ukazom `type`. Na primer:

```
>>> type(-17)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type('abc')
<class 'str'>
>>> a = 10 // 6
>>> type(a)
<class 'int'>
>>> b = 10 / 6
>>> type(b)
<class 'float'>
```

Znakovni podatkovni tip (nad.)

Vaja

Kakšnega tipa so spremenljivke `a`, `b`, `c`, `d` in `e` po izvedbi sledečih stavkov?

```
>>> a = 5.3
>>> b = int(7.2 * a)
>>> c = 5 * b + (2.0 - 2.0)
>>> d = 'a + b + int(c)'
>>> e = float(b) + int(b + c)
```

Nize združujemo z operatorjem `+`. Funkcija `print` izpiše niz brez navednic. Bodimo pozorni, da operator `+` za nize deluje drugače kot za števila. Seštevanje nizov seveda nima pomena.

```
>>> a = 'Pozdravljen, '
>>> b = 'svet!'
>>> c = a + b
>>> c
'Pozdravljen, svet!'
```

Operator `str` pretvori celo ali realno število v niz, kar je večkrat koristno — predvsem kadar želimo števila izpisati s funkcijo `print`.

```
>>> str(3)
'3'
>>> str(3.14159)
'3.14159'
```

2: Izdelava samostojnih programov in pogojni stavki

Program in pogojni stavki

V prejšnji učni enoti smo pisali in izvajali posamezne stavke, v tej pa jih bomo združili v samostojen program. Naučili se bomo, kako se s programom »pogovarjamo« — kako mu posredujemo podatke in kako nam program prikaže rezultate. Spoznali bomo tudi pogojne stavke, s katerimi lahko računalniku povemo, naj določen del programa izvede samo v primeru, če je izpolnjen določen pogoj.

Če je zunaj mrzlo in sneži, potem obleci šal, kapo in rokavice. Če dežuje, obleci dežni plašč, sicer pa jopico.
S takšnimi in podobnimi vprašanji se srečujemo vsakodnevno. Morda se ne zavedamo, da gre pri tem v bistvu za pogojne stavke.

Od zaporedja stavkov do programa

Eva se je odpravila na 30-kilometrsko kolesarsko pot. Ko je odšla od doma, je bila ura 16:35, ko se je vrnila, pa so kazalci kazali 18:05. Zanima jo, s kakšno povprečno hitrostjo (v km/h) je kolesarila, zato odpre pythonov interaktivni tolmač in svoje podatke najprej zapiše v spremenljivke:

```
>>> pot = 30
>>> uraZacetka = 16
>>> minZacetka = 35
>>> uraKonca = 18
>>> minKonca = 5
```

Eva dobro ve, da se povprečna pot v km/h izračuna kot količnik prepotovane poti v kilometrih in porabljenega časa v urah. Dolžino poti pozna, za izračun porabe časa pa se mora malo potruditi. Najprej mora izračunati razliko med končno in začetno uro, nato pa še razliko med končno in začetno minuto. Porabo časa v minutah izračuna tako, da razliko v urah pomnoži s 60, k temu pa prišteje še razliko v minutah. Če rezultat deli s 60, dobi porabo časa v urah, ki jo potrebuje za izračun hitrosti:

```
>>> razlikaUr = uraKonca - uraZacetka
>>> razlikaMinut = minKonca - minZacetka
>>> porabaCasaVMin = 60 * razlikaUr + razlikaMinut
>>> porabaCasaVUrah = porabaCasaVMin / 60
>>> porabaCasaVUrah
1.5
```

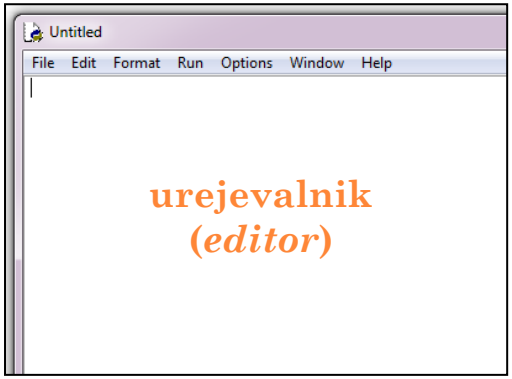
Drži, kolesarila je natanko poldrugo uro, v kar se sicer zlahka prepriča tudi brez pomoči računalnika, kljub temu pa se ji zdi imenitno, da zna že toliko programirati. Do iskanega rezultata jo loči le še en preprost stavek:

```
>>> povprecnaHitrost = pot / porabaCasaVUrah
>>> povprecnaHitrost
20.0
```

Od zaporedja stavkov do programa

Eva, navdušena kolesarka in čedalje bolj zagnana programerka, postopek ponovi še za nekaj naslednjih tur, kmalu pa se naveliča vnašanja venomer istih vrstic, ki se razlikujejo le v podatkih. K sreči pa nam python omogoča, da zaporedje stavkov zapišemo v datoteko in ga nato samodejno izvajamo. Tako dobimo **program** — zaporedje stavkov, ki ga lahko poženemo (izvršimo). V tem razdelku si bomo ogledali, kako program zapišemo v datoteko

* Tolmač: *File* > *New File*



V urejevalniku ustvarimo datoteko* z imenom `hitrost.py` (datoteke s pythonovimi programi morajo vedno imeti končnico `.py`) in vanjo zapišimo stavke za izračun in izpis povprečne hitrosti:

```
# definiramo spremenljivke
pot = 30
uraZacetka = 16
minZacetka = 35
uraKonca = 18
minKonca = 5

# izračunamo porabo časa v urah
razlikaUr = uraKonca - uraZacetka
razlikaMinut = minKonca - minZacetka
porabaCasaVMin = 60 * razlikaUr + razlikaMinut
porabaCasaVUrah = porabaCasaVMin / 60

# izračunamo in izpišemo povprečno hitrost v km/h
povprecnaHitrost = pot / porabaCasaVUrah
print(povprecnaHitrost)
```

Način, ki smo ga uporabljali v interaktivnem tolmaču, v programu namreč ne deluje; če bi namesto `print(povprecnaHitrost)` zapisali samo `povprecnaHitrost`, program ne bi izpisal ničesar. Vrstice, ki se pričnejo z znakom #, so **komentarji**. Pythonov tolmač komentarje preprosto preskoči, za naše razumevanje programa pa so pogosto nepogrešljivi.

PROGRAM, ZAPISAN V DATOTEKI

Datoteko `hitrost.py` moramo seveda shraniti.* Priporočljivo jo je shraniti v posebno mapo, ki bo namenjena samo pythonovim programom. Programa nato ne poženemo v pythonovem interaktivnem tolmaču, ampak v terminalu (konzoli) operacijskega sistema.**

V terminalu se moramo nato premakniti v mapo, kamor smo shranili program `hitrost.py`. Kako to naredimo? Recimo, da smo mapo poimenovali `mojiProgrami`. Če smo jo ustvarili na namizju (`Desktop` na angleških sistemih), potem se vanjo premaknemo takole:

```
> cd Desktop
> cd mojiProgrami
```

Znaka `>` (oziroma `$` na Linux in OS X) ne pišemo; z njim zgolj označujemo, da smo v terminalu. Po vsaki vrstici moramo seveda pritisniti tipko `Enter`. Namesto `cd Desktop` bomo morda morali napisati `cd Namizje`. Če nismo prepričani, poženemo ukaz `ls -l` na sistemih Linux ali OS X oziroma `dir` na sistemu windows. Ta ukaz nam izpiše vsebino trenutne mape. Če v izpisu najdemo ime `Desktop`, poženemo ukaz `cd Desktop`, če vidimo ime `Namizje`, pa poženemo ukaz `cd Namizje`.

Ko se v terminalu pomaknemo v mapo, v katero smo shranili datoteko `hitrost.py`, jo poženemo s sledečim ukazom:

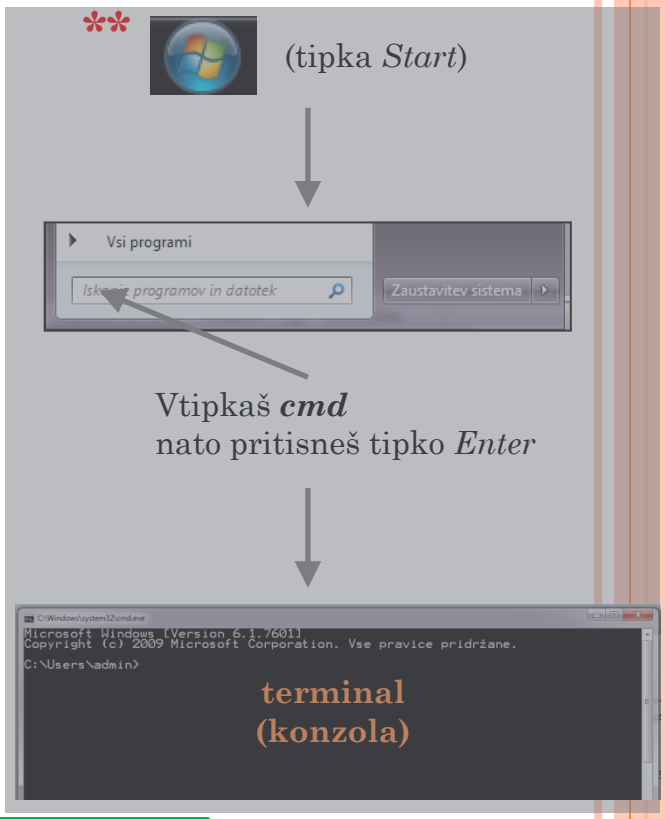
```
> python3 hitrost.py
```

Program `hitrost.py` se izvede stavek za stavkom, torej natanko tako, kot če bi stavke v istem vrstnem redu vtipkali v pythonov interaktivni tolmač. Po izvedbi funkcije `print` se na zaslon izpiše izračunana povprečna hitrost:

```
> 20.0
```

Kolesarki Evi smo prihranili veliko truda. Odslej ji ne bo več treba po vsakem izletu vtipkovati stavkov za izračun povprečne hitrosti, ampak bo samo ustrezno spremenila vrednosti spremenljivk v programu `hitrost.py` in ga ponovno pognala. To je vse!

* Urejevalnik: *File* > *Save As* ...



Shranjeni program, napisan v urejevalniku, lahko poženemo tudi kar v interaktivnem tolmaču (urejevalnik: *Run* > *Run Module*; bližnjica: F5)

Vhod in izhod

Naš program za izračun hitrosti ni najbolj prijazen. Če želimo izračunati povprečno hitrost za drugačne podatke, kot so trenutno navedeni v programu, moramo program spremeniti. Veliko bolje bi bilo, če bi program **povprašal** uporabnika po vhodnih podatkih. To dosežemo s pythonovo funkcijo `input`, ki jo v interaktivnem tolmaču poženemo takole:

```
>>> pot = input('Koliko km si prevozil(a)? ')
Koliko km si prevozil(a)? 30
>>> pot
'30'
```

Funkcija `input` izpiše besedilo, ki ji ga podamo (zakaj potrebujemo navednice?), nato pa počaka, da uporabnik vnese podatek. Denimo, da vnesemo število 30. Uporabnikov vnos smo podčrtali (30); tako bomo počeli tudi v nadaljevanju. Ker smo stavek zapisali kot `pot = input(...)`, se uporabnikov vnos (rezultat funkcije `input`) priredi spremenljivki `pot`.

Ko spremenljivko `pot` izpišemo, vidimo, da jo je python shranil kot **niz**, ne kot število. Funkcija `input` namreč ne more vedeti, kakšen tip podatkov naj pričakuje, zato uporabnikov vnos vedno obravnava kot niz.

K sreči pa niz, ki predstavlja število, zlahka pretvorimo v dejansko število. Uporabimo operator `int` (za pretvorbo v celo število) oziroma `float` (za pretvorbo v realno število), ki smo ju spoznali že v prejšnji učni enoti.

Vhod in izhod (nad.)

```
>>> a = '3.2'
>>> b = '-47'
>>> c = '13 je srečna številka.'
>>> float(a)
3.2
>>> int(b)
-47
>>> int(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '13 je srečna številka.'
```

Pri spremenljivki `c` se nam je zalomilo, saj operatorja `int` in `float` delujeta samo, če je celotno vsebino niza možno razumeti kot eno samo število.

Vhod in izhod (nad.)

Skopirajmo naš program `hitrost.py` v program `hitrostVhod.py` in ga spremenimo tako, da bo vse podatke vnesel uporabnik:

```
1  # definiramo spremenljivke.
2  pot = float(input('Koliko km si prevozil(a)? '))
3  uraZacetka = int(input('Vnesi uro začetka: '))
4  minZacetka = int(input('Vnesi minuto začetka: '))
5  uraKonca = int(input('Vnesi uro konca: '))
6  minKonca = int(input('Vnesi minuto konca: '))
7
8  # izračunamo porabo časa v urah.
9  razlikaUr = uraKonca - uraZacetka
10 razlikaMin = minKonca - minZacetka
11 porabaCasaVMin = 60 * razlikaUr + razlikaMin
12 porabaCasaVUrah = porabaCasaVMin / 60.0
13
14 # izračunamo in izpišemo povprečno hitrost v km/h.
15 povprecnaHitrost = pot / porabaCasaVUrah
16 izpis = 'Vozil(a) si s povprečno hitrostjo '
17 print(izpis + str(povprecnaHitrost) + ' km/h.')
```

Prevožena pot je v splošnem lahko realno število, zato smo rezultat funkcije `input` (uporabnikov vnos) pretvorili v vrednost tipa `float`. Tudi izpis je sedaj nekoliko bolj prijazen: najprej izpišemo niz Vozila(a) si s povprečno hitrostjo, nato vrednost spremenljivke `povprecnaHitrost`, nazadnje pa še niz `km/h`. Posamezne dele izpisa med seboj ločimo z znakom `+`.

Poženimo naš novi program v terminalu operacijskega sistema:

```
> python3 hitrostVhod.py
Koliko kilometrov si prevozil(a)? 20
Vnesi uro ob začetku vožnje: 8
Vnesi minuto ob začetku vožnje: 50
Vnesi uro ob koncu vožnje: 10
Vnesi minuto ob koncu vožnje: 0
Vozil(a) si s povprečno hitrostjo 17.1428571429 km/h.
```

Python nam rezultat izpiše na veliko preveč decimalk natančno, a se s tem problemom ne bomo ukvarjali. Pomembneje je, da se naš program zna "pogovarjati" z uporabnikom: uporabnik programu poda vhod, program pa mu izpiše svoj izhod. V splošnem je vhod v program množica podatkov, ki jih program prejme. Vhod lahko — tako kot v našem primeru — vnese uporabnik, lahko pa ga program pridobi iz datoteke ali z interneta ali pa mu ga posreduje nek drug program. Izhod pa je skupno ime za vse podatke, ki jih program proizvede pri danem vhodu. V našem primeru smo izhod izpisali na zaslon, lahko pa ga program tudi zapiše v datoteko ali posreduje nekemu drugemu programu.

Izpis (funkcija *print*)

V interaktivnem tolmaču funkcije `print` običajno ne potrebujemo, kljub temu pa jo lahko uporabljamo.

```
>>> stevilo = 42
>>> stevilo
42
>>> print(stevilo)
42
>>> niz = 'tralala'
>>> niz
'tralala'
>>> print(niz)
tralala
```

Vidimo, da funkcija print pri izpisu nizov izpusti navednice.

Če želimo s funkcijo `print` v isti vrstici izpisati več vrednosti, si lahko pomagamo z vejicami:

```
>>> a = 3
>>> b = 5
>>> vsota = a + b
>>> print(a, '+', b, '=', vsota)
3 + 5 = 8
```

Python po vsakem izpisu vrednosti samodejno doda še presledek. Če dodatnih presledkov ne želimo, lahko vrednosti s pomočjo operatorja + sestavimo v enoten niz, nato pa dobljeni niz izpišemo:

```
>>> a = 37
>>> b = 5
>>> k = a // b
>>> niz = 'Število ' + b + ' gre ' + k + '-krat v število ' + a + '.'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

!!!

Python nam ne pusti združevanja nizov in števil z operatorjem + . Ta operator lahko uporabimo bodisi za seštevanje števil ali pa za združevanje (lepljenje) nizov. Če želimo niz sestaviti na podlagi vrednosti različnih tipov, moramo vse vrednosti, ki niso nizi, s pomočjo operatorja str pretvoriti v nize – torej uporabiti: str(b), str(k) in str(a)!

Pogojni stavek

Računalniku pogosto želimo povedati nekaj v tem smislu: »če je izpolnjen nek pogoj, naredi to, sicer pa naredi nekaj drugega«. **Pogojno izvrševanje stavkov** v pythonu izrazimo takole:

```
if pogoj:
  stavek_A1
  stavek_A2
  ...
else:
  stavek_B1
  stavek_B2
  ...
```

Pogojni stavek je sestavljen iz dveh **blokov** — skupkov stavkov. Če je `pogoj` izpolnjen, se bodo po vrsti izvršili stavki v bloku `if` (`stavek_A1`, `stavek_A2` itd.), stavki v bloku `else` (`stavek_B1`, `stavek_B2` itd.) pa se bodo preskočili, sicer pa bo ravno obratno. Blok `else` lahko tudi izpustimo. Besedi `if` in `else` imata v pythonu poseben pomen. Pravimo, da gre za **rezervirani besedi**.* To pomeni, da ju ne moremo uporabiti za imena spremenljivk.

Vsi stavki znotraj blokov `if` in `else` morajo biti zamaknjeni za enako število presledkov oz. tabulatorskih premikov. Ponavadi uporabljamo dva ali štiri presledke ali en tabulatorski premik. Zamikanje močno izboljša preglednost kode. V pythonu je obvezno, v večini drugih programskih jezikov pa se moramo vanj prisiliti sami.

*

```
Here is a list of the Python keywords

False       def         if          raise
None        del         import      return
True        elif        in          try
and         else        is          while
as          except      lambda      with
assert      finally     nonlocal    yield
break       for         not
class       from        or
continue    global      pass
```

Pogojni stavek (nad.)

Oglejmo si primer uporabe pogojnega stavka. Za dnevno vstopnico plavalni bazen plačajo polnoletni polno ceno, mladoletni pa jo dobijo pol ceneje. Napišimo program `bazen.py`, ki v realnoštevilsko spremenljivko `cena` prebere polno ceno, v celoštevilsko spremenljivko `starost` pa starost osebe, nato pa izpiše ustrezno ceno vstopnice:

```
1 cena = float(input('Koliko stane vstopnica? '))
2 starost = int(input('Koliko let imaš? '))
3 if starost >= 18:
4   c = cena
5 else:
6   c = cena / 2
7 print('Vstopnica zate stane ' + str(c) + ' EUR.')
```

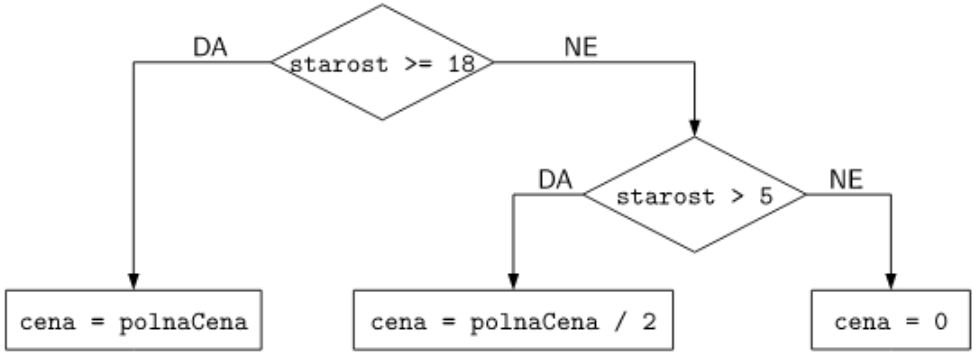
Preizkusimo program:

```
> python3 bazen.py
Koliko stane vstopnica? 6.5
Koliko let imaš? 21
Vstopnica zate stane 6.5 EUR.

> python3 bazen.py
Koliko stane vstopnica? 6.5
Koliko let imaš? 17
Vstopnica zate stane 3.25 EUR.
```

Operator `>=` pomeni enako kot v matematiki ≥, torej **večje ali enako kot**. Uporabljamo še sledeče **primerjalne operatorje**: `>` (večje od), `<` (manjše od), `<=` (manjše ali enako kot), `==` (enako kot) in `!=` (različno od). Bodimo pozorni na razliko med operatorjema `=` in `==`. Operator `=` vpiše vrednost desne strani v spremenljivko na levi strani, operator `==` pa ugotovi, ali sta vrednosti na levi in desni strani enaki.

Pogojni stavek (nad.)

Vrnimo se k našemu primeru s plavalnim bazenom. Recimo, da bo vstopnica za otroke do vključno 5. leta starosti odslej zastonj. Polnoletni še vedno plačajo polno ceno, tisti vmes pa polovično. Nalogo sedaj rešimo tako, da se najprej vprašamo, ali je starost enaka najmanj 18 let. Če je, nastavimo spremenljivko `cena` na polno ceno, sicer pa preverimo še, ali je starost večja od 5 let. Če je, je cena polovična, sicer pa znaša 0:



Popravimo program:

```
cena = float(input('Koliko stane vstopnica? '))
starost = int(input('Koliko let imaš? '))
if starost >= 18:
  c = cena
else:
  if starost > 5:
    c = cena / 2
  else:
    c = 0
print('Vstopnica zate stane ' + str(c) + ' EUR.')
```

Zaporedje `else-if` se v programiranju pogosto uporablja, zato nam python ponuja okrajšavo `elif`:

Pogojni stavek (nad.)

```
1 cena = float(input('Koliko stane vstopnica? '))
2 starost = int(input('Koliko let imaš? '))
3 if starost >= 18:
4   c = cena
5 elif starost > 5:
6   c = cena / 2
7 else:
8   c = 0
9 print('Vstopnica zate stane ' + str(c) + ' EUR.')
```

Program sedaj beremo takole: Če je starost enaka najmanj 18, priredi spremenljivki `c` vrednost spremenljivke `cena`. V nasprotnem primeru preveri, ali je starost večja od 5. Če to drži, je cena polovična, sicer pa se bomo kopali zastonj.

V stavku `if-elif-elif-...-else` se vedno izvede **natanko en blok** (skupek stavkov). Če je pogoj izpolnjen že pri bloku `if`, se izvedejo stavki v tem bloku, vsi ostali bloki pa se preskočijo. V nasprotnem primeru se preveri pogoj pri prvem bloku `elif`; če je izpolnjen, se izvedejo pripadajoči stavki, preostali bloki pa se preskočijo. V nasprotnem primeru se preveri pogoj pri drugem bloku `elif` itd. Blok `else` se izvede le v primeru, če nobeden od pogojev pri blokih `if` in `elif` ni izpolnjen.

Vaja

Kaj izpiše sledeči program, če je vrednost spremenljivke `tocke` enaka 3? Kaj se izpiše, če je enaka 7, 0, 10 oziroma 15?

```
tocke = int(input('Vnesi število točk: '))
if tocke > 10:
  print('A')
if tocke >= 5:
  print('B')
elif tocke > 0:
  print('C')
else:
  print('D')
```

Logični izrazi

Matematični izrazi, kot so npr. `1 + 2` ali `(3.7 - 2.6) / 5.4`, so nam že precej domači. Pri programiranju pa nam pogosto pridejo prav tudi **logični** izrazi. Primera logičnih izrazov sta, denimo, `3 <= 4` in `5 > 9`. Logični izraz ima samo dve možni vrednosti: **resnico** (`True` v pythonu) in **neresnico** (`False` v pythonu). Vrednost izraza `3 <= 4` je torej resnica, vrednost izraza `5 > 9` pa neresnica, saj število 5 ni večje od števila 9. Sledi še nekaj primerov:

```
>>> 2 <= 2
True
>>> 2 < 2
False
>>> 3 == 4
False
>>> 3 != 4
True
```

Vrednosti logičnih izrazov lahko shranjujemo v spremenljivke **logičnega tipa** (tip `bool` v pythonu):

```
>>> a = (2 < 1)
>>> b = (3 == 3)
>>> a
False
>>> b
True
>>> type(a)
<class 'bool'>
>>> type(b)
<class 'bool'>
```

type(a) = bool

Logični izrazi (nad.)

Logične izraze lahko med seboj povezujemo z **logičnimi operatorji** `and`, `or` in `not`. Izraz `a and b` je resničen samo v primeru, če sta podizraza `a` in `b` oba resnična. Izraz `a or b` je resničen, če je vsaj eden od podizrazov resničen; neresničen je samo tedaj, ko sta oba podizraza neresnična. Izraz `not a` pa je resničen, če je podizraz `a` neresničen, in obratno.

a	*b*	*a* and *b*	*a* or *b*	not *a*	not *b*
True	True	True	True	False	False
True	False	False	True	False	True
False	True	False	True	True	False
False	False	False	False	True	True

Tabela logičnih operatorjev

Na primer, vrednost izraza `(3 <= 5) and (6 == 3)` je enaka `False`, saj je prvi podizraz resničen, drugi pa neresničen. Vrednost izraza `(3 <= 5) or (6 == 3)` pa je enaka `True`.

Oglejmo si primer uporabe logičnih operatorjev. Leto je prestopno, če je deljivo s 400 ali pa če je deljivo s 4, vendar ne s 100. Zapišimo to pravilo v obliki programa `prestopnoLeto.py`. Spomnimo se, da je število *a* deljivo s številom *b* natanko tedaj, ko je ostanek pri deljenju *a* z *b* enak 0.

```
1 L = int(input('Vnesi leto: '))
2 if L % 400 == 0 or L % 4 == 0 and L % 100 != 0:
3   print('Leto ' + str(L) + ' je prestopno.')
4 else:
5   print('Leto ' + str(L) + ' ni prestopno.')
```

Podobno kot ima množenje prednost pred seštevanjem, ima tudi operator `and` prednost pred operatorjem `or`, zato nam oklepajev v pogoju pri bloku `if` ne bi bilo treba pisati. Operator `not` pa ima prednost pred obema.

Hrošči in razhroščevanje

Programiranje je človeško opravilo, ljudje pa smo zmotljivi. Napakam, ki se prikradejo v naše programe, pravimo **hrošči**. Izraz izvira še iz časov, ko so bili računalniki veliki kot omare. Nekoč je v enega od njih prilezel dejanski hrošč, ki je povzročil pregrevanje vezja in s tem napačno delovanje.

Hrošč ni nujno posledica neprevidnosti. Lahko zgolj pozabimo na kakšen poseben primer, kot se nam je zgodilo pri programu za računanje povprečne hitrosti `hitrost2.py`, ko nismo upoštevali (sicer neobičajne, a povsem smiselne) možnosti nočnega kolesarjenja:

```
Koliko km si prevozil(a)? 10
Vnesi uro začetka: 23
Vnesi minuto začetka: 50
Vnesi uro konca: 0
Vnesi minuto konca: 20
Vozil(a) si s povprečno hitrostjo -0.425531914894 km/h.
```

Razhroščevanja (odpravljanja napak) se najlažje lotimo z vstavljanjem **vmesnih izpisov**. Pri našem programu bi lahko izpisali vrednost spremenljivke `porabaCasaVMin` in bi hitro videli, da je nekaj narobe, saj bi bila negativna (glej program na desni, vrstica 12).

Vmesni izpisi pridejo prav tudi pri pogojnih stavkih:

```
if vseJeVRedu:
  print('Juhuhu, vse je v redu!')
  ...
else:
  print('Ojej, nekaj smrdi!')
  ...
```

Smiselno postavljeni vmesni izpisi so izjemno močno programersko orodje, včasih pa nam vendarle koristijo posebni programi za razhroščevanje, ki nam omogočajo izvajanje programa po korakih, spremljanje vrednosti spremenljivk med izvajanjem, pa še marsikaj.

```
1  # definiramo spremenljivke
2  pot = float(input('Koliko km si prevozil(a)? '))
3  uraZacetka = int(input('Vnesi uro začetka: '))
4  minZacetka = int(input('Vnesi minuto začetka: '))
5  uraKonca = int(input('Vnesi uro konca: '))
6  minKonca = int(input('Vnesi minuto konca: '))
7
8  # izračunamo porabo časa v urah
9  razlikaUr = uraKonca - uraZacetka
10 razlikaMin = minKonca - minZacetka
11 porabaCasaVMin = 60 * razlikaUr + razlikaMin
12 print(porabaCasaVMin)
13 porabaCasaVUrah = porabaCasaVMin / 60
14
15 # izračunamo in izpišemo povprečno hitrost
16 povprecnaHitrost = pot / porabaCasaVUrah
17 izpis = 'Vozil(a) si s povprečno hitrostjo '
18 print(izpis + str(povprecnaHitrost) + ' km/h.')
```

Tolmač: ***Debug > Debugger***

3: Zanke

Zanke

Zanka je del programa, ki ga računalnik izvede po večkrat zaporedoma. Uporabljamo jo v primerih, ko hočemo po večkrat izvesti enake (ali vsaj podobne) operacije; na primer če hočemo nekaj narediti z vsakim členom nekega zaporedja, z vsakim številom z nekega območja in podobno. Delovanje zanke nadziramo tako, da predpišemo ustavitveni pogoj (zanka se konča, ko je ta pogoj izpolnjen), ogledali pa si bomo tudi mehanizme za predčasni izhod iz zanke.

Vsako oblačilo iz škafa stresi, raztegni in obesi na vrv za sušenje perila. To je samo eden izmed primerov uporabe zanke v resničnem življenju. Zagotovo pa jih lahko najdemo še precej več.

Zanka *while*

Doslej smo videli, da računalnik načeloma izvaja stavke našega programa po vrsti, kot so zapisani v programu, vsakega po enkrat. S pogojnimi stavki smo se tudi naučili doseči, da izvajanje nek del programa preskoči (če pogoj ni izpolnjen). Kaj pa, če hočemo, da se nek del programa izvede po večkrat? Tedaj lahko uporabimo **zanko**.

Primer

Naslednja zanka izpiše števila od 1 do 10.

```
n = 1 # spremenljivka n je števec, ki
      # predstavlja naslednje število,
      # ki ga moramo še izpisati.
while n <= 10:
  print(n)
  n = n + 1
```

Stavek `while` označuje, da gre za zanko; za njim pride pogoj za nadaljevanje — v našem primeru je to `n <= 10`, torej se zanka izvaja, dokler je pogoj `n <= 10` izpolnjen. Pri vsaki ponovitvi zanke se izvedeta stavka `print(n)` (ki izpiše trenutno vrednost `n`) in `n = n + 1` (ki poveča `n` za 1).

Oglejmo si še en primer problema, ki ga je veliko lažje rešiti z zanko kot brez nje.

ZANKA *WHILE* (NAD.)

Primer

Legenda pravi, da je iznajditelj šaha od kralja svoje dežele zahteval naslednje plačilo: eno zrno riža za prvo polje šahovnice; dve zrni za drugo polje; štiri zrna za naslednje polje; in tako naprej, za vsako polje dvakrat toliko zrn kot za prejšnje polje. Šahovnica ima 64 polj. Koliko zrn riža mora kralj v tej zgodbi plačati?

Če poskušamo to nalogo reševati ročno, se bomo hitro utrudili in najbrž tudi zmotili: računati moramo $2 \cdot 1 = 2$; $2 \cdot 2 = 4$; $2 \cdot 4 = 8$; $2 \cdot 8 = 16$; $2 \cdot 16 = 32$; in tako naprej še kar nekaj časa; in na koncu bi morali vse te zmnožke še sešteti. Dolgo, ponavljajoče se zaporedje enih in istih operacij je kot nalašč za reševanje z računalnikom. Poskusimo napisati program `sahovnica.py`, ki reši našo nalogo:

```
skupajZrn = 0
zrnNaPolju = 1 # število zrn na 1. polju
skupajZrn = skupajZrn + zrnNaPolju
zrnNaPolju = zrnNaPolju * 2 # število na 2. polju
skupajZrn = skupajZrn + zrnNaPolju
zrnNaPolju = zrnNaPolju * 2 # število na 3. polju
skupajZrn = skupajZrn + zrnNaPolju
zrnNaPolju = zrnNaPolju * 2 # število na 4. polju
skupajZrn = skupajZrn + zrnNaPolju
...
print(skupajZrn)
```

V spremenljivki `zrnNaPolju` hranimo število zrn na trenutnem polju, v `skupajZrn` pa skupno število zrn na vseh doslej obdelanih poljih. Ko se premaknemo na naslednje polje, moramo `zrnNaPolju` pomnožiti z 2, da dobimo število zrn na tem naslednjem polju (ki je dvakrat tolikšno kot na prejšnjem), nato pa to število prištejemo k skupnemu številu zrn na vseh poljih doslej (torej k spremenljivki `skupajZrn`).

ZANKA *WHILE* (NAD.)

Ta dva koraka moramo zdaj ponoviti za vsako polje šahovnice. V prejšnjem primeru bi to pomenilo, da se morata isti dve vrstici ponoviti še 60-krat (tam, kjer so v primeru tri pike). V praksi si seveda ne želimo tolikokrat pisati istih dveh vrstic, zato je bolje uporabiti zanko. Takšno rešitev kaže spodnji primer, podrobnejši opis pa sledi spodaj:

```
1  skupajZrn = 0
2  polje = 1
3  zrnNaPolju = 1 # število zrn na prvem polju
4  skupajZrn = skupajZrn + zrnNaPolju
5  while polje < 64:
6    # premaknimo se na naslednje polje
7    polje = polje + 1
8    # izračunajmo število zrn na njem
9    zrnNaPolju = zrnNaPolju * 2
10   skupajZrn = skupajZrn + zrnNaPolju
11 print(skupajZrn)
```

Z **zanko** `while` računalniku naročimo, naj neko skupino stavkov (ki ji pravimo **telo zanke**) izvaja, dokler je izpolnjen določen pogoj za nadaljevanje. Posamezni izvedbi telesa zanke pravimo **iteracija**. Pogoj za nadaljevanje zanke se preveri le pred začetkom vsake iteracije. Če torej pogoj za nadaljevanje preneha veljati nekje na sredi iteracije, se bo ta iteracija še vseeno izvedla do konca. Mogoče je tudi, da se telo zanke ne izvede niti enkrat (če pogoj za nadaljevanje že na začetku, pred prvo iteracijo, ni izpolnjen).

V našem primeru na prejšnji strani s spremenljivko `polje` štejemo, koliko polj smo že obdelali. Zanko moramo torej ponavljati, dokler je ta števec manjši od 64, zato imamo pogoj `polje < 64`. V vsaki iteraciji zanke povečamo števec `polje` za 1, izračunamo število zrn na naslednjem polju (`zrnNaPolju`) in ga prištejemo k skupni vsoti (`skupajZrn`).

Vaja

Recimo, da ima kralj iz naše zgodbe »le« milijardo riževih zrn. Pri katerem polju šahovnice mu zmanjka riža?

ZANKA *FOR*

Zanke pogosto uporabljamo za to, da se z neko spremenljivko sprehodimo po vseh elementih nekakšnega zaporedja in z vsakim od njih nekaj naredimo. Pri nalogi s šahovnico smo se na primer sprehodili po vseh poljih šahovnice, pri čemer je spremenljivka `x` po vrsti dobila vrednosti 2, 3, ... , 64.

Ta primer, torej da hočemo iti z neko spremenljivko po vrsti skozi več zaporednih celih števil, je v praksi zelo pogost. Zato v večini programskih jezikov obstaja poseben tip zanke, s katerim lahko takšno zanko napišemo krajše in lažje kot z `while`. To je **zanka** `for`:

```
1 for x in range(2, 65):
2   print(x)
```

S to zanko smo računalniku naročili, naj spremenljivko `x` na začetku postavi na 2, po vsaki iteraciji naj jo poveča za 1, zanko pa naj ponavlja, dokler je `x < 65`. Naša zanka bi torej izpisala števila od vključno 2 do vključno 64.

Pomembna razlika v primerjavi z zanko `while` je torej ta, da smo morali tam sami poskrbeti za povečevanje števca, tukaj pa nam tega ni treba, saj zanj poskrbi tolmač.

Zanka *for* (nad.)

V primerih, ko hočemo zanko začeti pri 0, lahko začetno vrednost števca izpustimo. Spodnji program izpiše števila od vključno 0 do vključno 64.

```
1 for x in range(65):
2   print(x)
```

Lahko pa tudi zahtevamo, naj se števec ne povečuje za 1, ampak z nekim drugim korakom. V tem primeru moramo v `range(...)` dodati še tretji parameter. Spodnji program povečuje števec `x` v korakih po 10, tako da izpiše le števila 2, 20, 12, 120, 22, 220, 32, 320, 42, 420, 52, 520, 62, 620.

```
1 for x in range(2, 65, 10):
2   print(x)
3   x = x * 10
4   print(x)
```

Vaja

Napiši program, ki od uporabnika prebere število *n*, nato prebere *n* ocen (od 1 do 5) in izpiše njihovo povprečje.

Doslej smo videli, da se lahko z zanko `for` sprehodimo po več zaporednih celih številih. Obstaja pa tudi splošnejša oblika te zanke, s katero se lahko sprehodimo po poljubnem zaporedju elementov, ki sploh niso nujno števila. Spodnji program izpiše najprej 10, nato `abc` in nazadnje še 34.5.

```
1 for x in [10, "abc", 34.5]:
2   print(x)
```

Zanke - napredno

- Gnezdenje zank
- Predčasni izhod iz zanke
  - stavek *break*
- Skok na konec trenutne iteracije
  - stavek *continue*

→ Glej e-učbenik:
https://lusy.fri.uni-lj.si/ucbenik/book/1203/index5.html

4: Tabele (= seznami)

Seznam je interno shranjen kot tabela kazalcev na objekte (elemente), a v praksi se za ta podatkovni tip skoraj vedno uporablja ime 'seznam' (angl.: *list*)!

Tabele

Včasih potrebujemo veliko množico spremenljivk — na primer, rezultate atletskega tekmovanja za celotno šolo ali pa imena in priimke vseh prijateljev na Facebooku. Namesto da za vsak podatek definiramo ločeno spremenljivko, lahko množico podatkov s sorodnim pomenom shranimo v tabelo in s tem pridobimo možnost enostavnega dostopa in obdelave podatkov.

Prvi primer je polica, kjer je vsaka kravata v svojem predalčku. Drugi primer so dijaške omarice, kjer ima vsak dijak svojo omarico.
Oboje si lahko predstavljamo kot tabelo.

TABELA

Enaintrideseti december. Liza Bonbonc, lastnica trgovine s sladkarijami, pregleduje podatke o prodaji sladkih pregreh po posameznih mesecih. Letos si prvič pomaga s pythonom. Kako bi izračunala prodajo v celotnem letu?

```
# prodaja v kilogramih po posameznih mesecih
prodajaJan = 182
prodajaFeb = 250
prodajaMar = 225
prodajaApr = 315
prodajaMaj = 328
prodajaJun = 160
prodajaJul = 51
prodajaAvg = 36
prodajaSep = 340
prodajaOkt = 280
prodajaNov = 378
prodajaDec = 320

prodaja = prodajaJan + prodajaFeb + prodajaMar + \
          prodajaApr + prodajaMaj + prodajaJun + \
          prodajaJul + prodajaAvg + prodajaSep + \
          prodajaOkt + prodajaNov + prodajaDec

print(prodaja)
```

Po dolgotrajnem tipkanju Liza sicer dobi želeni podatek, ampak to je šele vsota; rada bi naredila še veliko drugih izračunov. Kako bo šele naslednje leto, ko namerava zbirati podatke za vsak *dan* posebej ...

Pomagajmo ji! Za tovrstne primere je skoraj v vseh programskih jezikih možno uporabiti **tabelo**. Tabela je zaporedje podatkov, ki imajo (praviloma) soroden pomen. V pythonu jo zapišemo tako, da njene podatke navedemo znotraj oglatih oklepajev in jih ločimo z vejicami:

```
>>> prodaja = [182, 250, 225, 315, 328, 160,
               51, 36, 340, 280, 378, 320]
```

type(prodaja) = list

Posamezne podatke v tabeli imenujemo **elementi**. Prvi element (182) v našem primeru predstavlja število prodanih kilogramov sladkarij v januarju, drugi (250) v februarju itd. Zapis `[]` predstavlja prazno tabelo (tabelo brez elementov).

Dostop do elementov tabele

Dostop do elementov tabele

Elementi tabele so dostopni preko **indeksov**. Indeks elementa je njegova zaporedna številka minus ena. Prvi element ima tako indeks 0, drugi indeks 1, tretji indeks 2 itd. Z izrazom *tabela*[*indeks*] pridobimo element tabele na podanem indeksu:

```
>>> prodaja = [182, 250, 225, 315, 328, 160,
               51, 36, 340, 280, 378, 320]
>>> prodaja[0]
182
>>> prodaja[7]
36
>>> prodaja[11]
320
>>> prodaja[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Element z indeksom 12 seveda ne obstaja, zato nam python sporoči napako.

0	1	2	3	4	5	6	7	8	9	10	11
180	250	225	315	328	160	51	36	340	280	378	320

Indeks elementa tabele: 0
Element tabele: 180

Vaja

Ustvarimo tabelo `t = [30, 50, 20, 60]`. Kakšni so rezultati sledečih izrazov?

- `t[1]`:
- `t[3]`:
- `t[len(t) - 1]`:

Odsek tabele

Z izrazom *tabela*[*a*:*b*] dobimo **odsek tabele**, ki je sestavljen iz elementov z indeksi od *a* do vključno *b* − *1*. Z izrazom *tabela*[*a*:] dobimo odsek, sestavljen iz elementov z indeksi od *a* naprej, z izrazom *tabela*[:*b*] pa odsek, sestavljen iz elementov z indeksi od 0 do vključno *b* − *1*:

```
>>> prodaja = [182, 250, 225, 315, 328, 160,
               51, 36, 340, 280, 378, 320]
>>> prodaja[5:8]
[160, 51, 36]
>>> prodaja[5:]
[160, 51, 36, 340, 280, 378, 320]
>>> prodaja[:8]
[182, 250, 225, 315, 328, 160, 51, 36]
```

Vaja

Kaj izpiše pythonov interaktivni tolmač pri izvajanju sledečih ukazov? Najprej odgovori, nato preveri!

```
>>> t = [70, 80, -60, -100, 50, 20, -40, 90, 60, -50]
>>> t[1:3]
>>> t[4:]
>>> t[:4]
>>> t[6:6]
>>> len(t[2:])
>>> len(t[:6])
>>> len(t[3:7])
>>> len(t[5:5])
```

Sprehod po elementih tabele

Če želimo sešteti elemente v tabeli, ugotoviti, kateri je največji ipd., se moramo znati po tabeli **sprehoditi** — obiskati vsak element posebej. Osnovni sprehod po tabeli smo si na kratko že ogledali v prejšnji učni enoti:

```
for element in tabela:
  # element vsebuje trenutni element tabele
```

V prvem obhodu zanke se v spremenljivko `element` zapiše prvi element tabele, v drugem drugi itd. Oglejmo si primer. Koliko kilogramov sladkarij je Liza Bonbonc prodala v celotnem letu? Na to vprašanje lahko odgovorimo s sprehodom po tabeli in pomožno spremenljivko, ki hrani skupno količino sladkarij do trenutnega meseca. Pomožno spremenljivko (imenovali jo bomo `prodajaLeto`) najprej postavimo na 0, v vsakem koraku zanke pa ji prištejemo trenutni element tabele. Napišimo program:

```
1 prodaja = [182, 250, 225, 315, 328, 160,
2            51, 36, 340, 280, 378, 320]
3 prodajaLeto = 0
4 for prodajaMesec in prodaja:
5   prodajaLeto = prodajaLeto + prodajaMesec
6 print(prodajaLeto)
```

V prvem obhodu zanke ima spremenljivka `prodajaMesec` vrednost 182, v drugem 250, v tretjem 225 itd. Spremenljivka `prodajaLeto` pa ima po prvem obhodu zanke vrednost 0 + 182 = 182, v drugem 182 + 250 = 432 (torej vsoto po prvih dveh mesecih), v tretjem 432 + 225 = 657 (vsoto po prvih treh mesecih) itd. Ko se zanka zaključi, spremenljivka `prodajaLeto` vsebuje skupno količino prodanih sladkarij v celotnem letu.

Sprehod po elementih tabele (nad.)

Vaja

Program za izračun vsote prepiši v datoteko (npr. `vsota.py`) in ga poženi iz terminala tvojega operacijskega sistema.

Kolikšna je bila največja mesečna prodaja sladkarij? Te naloge se lotimo tako, da se sprehodimo po tabeli, sproti pa vzdržujemo spremenljivko, ki podaja največjo prodajo **do trenutno obravnavanega meseca**. Spremenljivko — imenujmo jo `najProdaja` — na začetku postavimo na vrednost prvega elementa tabele. Nato jo v vsakem koraku zanke primerjamo s trenutnim elementom tabele. Če je trenutni element tabele večji od trenutne vrednosti spremenljivke `najProdaja`, priredimo trenutni element spremenljivki `najProdaja`, sicer pa ne naredimo ničesar. Tako dosežemo, da bo spremenljivka `najProdaja` po vsakem koraku zanke vsebovala doslej največji element, po koncu zanke pa največji element celotne tabele.

```
1 prodaja = [182, 250, 225, 315, 328, 160,
2            51, 36, 340, 280, 378, 320]
3 najProdaja = prodaja[0]
4 for prodajaMesec in prodaja:
5   if prodajaMesec > najProdaja:
6     najProdaja = prodajaMesec
7 print(najProdaja)
```

0	1	2	3	4	5	6	7	8	9	10	11
180	250	225	315	328	160	51	36	340	280	378	320

Vrednost spremenljivke `prodajaMesec`: 180

Vrednost spremenljivke `najProdaja`: 180

▶ Sprehod po elementih tabele Ponastavi

Sprehod po indeksih tabele

Pri sprehodu po tabeli včasih poleg trenutnega elementa potrebujemo tudi njegov indeks. V tem primeru lahko uporabimo sledečo zanko:

```
for indeks in range(len(tabela)):
  # indeks je indeks trenutnega elementa tabele
  # tabela[indeks] je trenutni element tabele
```

Spremenljivka `indeks` prepotuje vse veljavne indekse tabele, od 0 do `len(tabela) - 1`. Do trenutnega elementa tabele dostopamo z izrazom `tabela[indeks]`.

Vrnimo se k naši Lizi. V katerih mesecih je Liza prodala več kot 300 kilogramov sladkarij? Da bo naloga enostavnejša, bomo najprej izpisali indekse mesecev (njihove zaporedne številke minus ena). Januar ima torej indeks 0, februar 1 itd. Indeksi mesecev ustrezajo indeksom v tabeli, saj element na indeksu 0 predstavlja prodajo v januarju, element na indeksu 1 prodajo v februarju itd. Zato lahko naš problem rešimo tako, da se sprehodimo po indeksih tabele in vsakokrat pogledamo element na trenutnem indeksu. Če je večji od 300, potem vemo, da je Liza v mesecu s tem indeksom prodala več kot 300 kg sladkarij, zato indeks izpišemo.

```
1 prodaja = [182, 250, 225, 315, 328, 160,
2            51, 36, 340, 280, 378, 320]
3 for indeksMeseca in range(len(prodaja)):
4   if prodaja[indeksMeseca] > 300:
5     print(indeksMeseca)
```

Gornji program izpiše števila 3, 4, 8, 10 in 11, ki predstavljajo mesece april, maj, september, november in december.

Spreminjanje, dodajanje in odstranjevanje elementov

Element na nekem indeksu v tabeli spremenimo s preprostim prirejanjem:

```
>>> t = [1, 2, 3, 4]
>>> t[1] = 10
>>> t
[1, 10, 3, 4]
```

S stavkom *tabela*.`append`(*element*) dodamo element na konec tabele, s stavkom *tabela*.`insert`(*indeks*, *element*) pa element vstavimo pred element s podanim indeksom:

```
>>> t = [10, 20, 30, 40]
>>> t.append(100)
>>> t
[10, 20, 30, 40, 100]
>>> t.insert(1, 60)
[10, 60, 20, 30, 40, 100]
```

objektno programiranje (OP) – na *objektu* uporabimo *metodo*: *objekt.metoda*

Element lahko iz tabele odstranimo na dva načina. Ukaz

```
tabela[indeks : indeks + 1] = []
```

odstrani element na indeksu `indeks`, ukaz

```
tabela.remove(vrednost)
```

pa odstrani prvi element s podano vrednostjo. Če element ne obstaja, sproži izjemo.

Pomoč glede *metod* na *objektih*:
- v Pythonu: vtipkaš dir(list)
- na spletu: Google > "python3 list methods" (podobno glede iskanja metod za ostale Pythonove strukture)

Spreminjanje, dodajanje in odstranjevanje elementov (nad.)

```
>>> t = [30, 50, 20, 80, 60, 70, 50]
>>> t[2:3] = []
>>> t
[30, 50, 80, 60, 70, 50]
>>> t.remove(50)
>>> t
[30, 80, 60, 70, 50]
>>> t.remove(20)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Vaja

1

Kaj se izpiše po izvedbi sledečih stavkov? Najprej odgovori, nato preveri:

```
>>> t = [5, 6, 7]
>>> t.append(1)
>>> t
>>> t.remove(5)
>>> t
>>> t[1:2] = []
>>> t
```

5: Funkcije

Funkcije

Funkcija je del programa (skupina stavkov), ki smo mu dali neko ime in ga lahko po tem imenu od drugod v programu »pokličemo«, torej od računalnika zahtevamo, naj izvede stavke, ki tvorijo funkcijo. Na ta način lahko problem, ki ga rešujemo, razdelimo na manjše podprobleme in vsakega rešimo s funkcijo, ki jo potem pokličemo, ko jo potrebujemo pri reševanju glavnega problema. Program tako razdelimo na manjše enote, zaradi česar ga je lažje pisati, testirati in vzdrževati.

Želimo zložiti perilo, vendar je vsako vrsto oblačila (npr. hlače, majice, nogavice) potrebno zložiti malce drugače. Zlaganje vseh oblačil lahko razbijemo na manjše probleme (kako zložiti hlače, kako zložiti majico itd.).
To predstavlja uporabo funkcij pri reševanju danega problema.

FUNKCIJA

Funkcija ali **podprogram** je zaporedje stavkov, ki smo mu nadeli neko ime. Od drugod v programu lahko potem to funkcijo **pokličemo** in s tem od računalnika zahtevamo, naj takrat izvede zaporedje stavkov, ki tvorijo funkcijo. Pri klicu lahko funkcija dobi tudi vhodne podatke (parametre) in na koncu vrne nek rezultat, ki ga lahko klicatelj funkcije uporablja pri nadaljnjem delu.

Funkcije pridejo prav, ker lahko z njimi razdelimo program na manjše in bolj obvladljive dele. Ko funkcijo napišemo in se prepričamo, da deluje pravilno, jo lahko odtlej uporabljamo kot nekakšno »črno škatlico«, ne da bi morali vsakič znova razmišljati o podrobnostih tega, kako ta funkcija opravi svoje delo.

Funkcija ponavadi rešuje nek zaokrožen in jasno določen podproblem znotraj širšega problema, ki ga rešuje celoten program. Funkcije torej razvijamo tako, da znotraj problema, ki ga rešujemo, opazimo nek primeren podproblem, razmislimo o tem, kakšne vhodne podatke bomo potrebovali pri reševanju tega podproblema, in nato napišemo zaporedje korakov (stavkov), ki rešijo ta podproblem.

Oglejmo si konkreten primer: naslednja zelo preprosta funkcija dobi kot parametra dve števili `a` in `b`, pogleda, katero od njiju je večje, in to večje število vrne kot rezultat funkcije.

```
def Vecje(a, b):
  if a > b:
    return a
  else:
    return b
```

Stavek `def` pove, da gre za **deklaracijo** funkcije. Z njo pythonu povemo, kako naj bo naši funkciji ime (v tem primeru je to `Vecje`) in kateri stavki naj jo sestavljajo (to je blok zamaknjenih stavkov, ki sledijo stavku `def`; tej skupini stavkov pravimo **telo funkcije**). Tolmač na tem mestu teh stavkov še ne bo poskušal izvesti, pač pa si jih le zapomni za kasneje.

Funkcija (nad.)

Za imenom funkcije smo v oklepajih tudi navedli, da ta funkcija pri klicu zahteva dva parametra, ki ju bo videla pod imenoma `a` in `b`. Funkcija lahko s parametri dela podobne stvari kot s spremenljivkami. V našem primeru ima funkcija dva parametra, v splošnem pa jih imajo lahko funkcije tudi več ali manj (lahko tudi nobenega).

Kot vidimo v zgornjem primeru, se lahko v funkciji pojavlja stavek `return`. Ko tolmač pri izvajanju funkcije pride do tega stavka, se takoj vrne iz funkcije (preostanka telesa funkcije ne izvede) in vrne vrednost, ki je bila navedena za besedo `return`. V našem primeru torej naša funkcija vrne bodisi vrednost `a` bodisi `b`, odvisno od tega, katera je večja.

Zdaj ko imamo funkcijo deklarirano, jo lahko iz drugih delov programa pokličemo tako, da navedemo njeno ime in nato v oklepajih še parametre, na primer takole:

```
1 def Vecje(a, b):
2   if a > b:
3     return a
4   else:
5     return b
6
7 c = Vecje(10, 20) + 30
8 print(c) # izpiše 50
```

Podizraz `Vecje(10, 20)` je **klic funkcije**; tolmač bo, da izračuna njegovo vrednost, izvedel telo funkcije `Vecje` in pri tem kot parameter `a` uporabil vrednost 10, kot parameter `b` pa vrednost 20. V tem konkretnem primeru funkcija vrne 20, kar potem tolmač vzame kot vrednost podizraza `Vecje(10, 20)` in s to vrednostjo računa naprej — v našem primeru se to vrednost sešteje s 30 in rezultat priredi spremenljivki `c`, tako da ima ta na koncu vrednost 50.

Enostaven primer

enost_funk.py - C:/Users/alhladnik/Documents/Ales - OTGO/Ales/Predmeti/IM1/2016-17/Programiranje Python/...

File Edit Format Run Options Window Help

```
# Enostavna funkcija: izračun ploščine trikotnika
# ----------

def ploscina_trikotnika(osn, vis):
    return osn * vis / 2
```

Funkcijo napišemo v urejevalniku, shranimo v datoteko *enost_funk.py* (CTRL + s) in jo po ukazu *Run > Run Module* (F5) lahko uporabljamo – kličemo – kar iz tolmača

Python 3.5.2 Shell

File Edit Shell Debug Options Window Help

```
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5
2016, 11:41:13) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more inf
ormation.
>>>
 RESTART: C:/Users/alhladnik/Documents/Ales - OTGO/Ales
/Predmeti/IM1/2016-17/Programiranje Python/Data/enost_f
unk.py
>>> ploscina_trikotnika(10, 4)
20.0
```

Malo večji primer

Primer

Vremenska postaja vsak dan opoldne izmeri temperaturo. V obdobju *n* dni se nam tako nabere zaporedje *n* meritev. Napiši program, ki prebere tako zaporedje in poišče v njem najtoplejši dan; izpiše naj indeks tega dne in povprečno temperaturo 7-dnevnega obdobja, ki obsega ta dan in še tri dni pred njim in po njem (lahko tudi manj, če bi drugače padli ven iz zaporedja). Nato naj enako naredi še za najhladnejši dan.

Predpostavimo zdaj, da imamo podatke že v spremenljivki `n` in tabeli `meritve`, in se lotimo naslednjega dela programa:

```
# Poiščimo dan z najvišjo temperaturo.
naj = 0
for d in range(n):
  if meritve[d] > meritve[naj]:
    naj = d
# Izpišimo ga, kot zahteva besedilo naloge.
danOd = naj - 3
danDo = naj + 3
if danOd < 0:
  danOd = 0
if danDo >= n:
  danDo = n - 1
vsota = 0
for d in range(danOd, danDo + 1):
  vsota += meritve[d]
print(d, vsota / (danDo - danOd + 1))
# Poiščimo dan z najnižjo temperaturo.
naj = 0
for d in range(n):
  if meritve[d] < meritve[naj]:
    naj = d
# Izpišimo ga, kot zahteva besedilo naloge.
...
```

Ta del je smiselno zapisati kot novo funkcijo Izpisi()

skrajšan zapis za: vsota = vsota + meritve[d]

Program za branje zaporedja *n* meritev

```
n = int(input("Vnesi število dni: "))
meritve = []
for i in range(n):
  t = int(input("Vnesi temperaturo: "))
  meritve.append(t)

#print(n)
#print(meritve)
```

kompl_funk1.py

Malo večji primer (nad.)

Manjka nam še izpis najhladnejšega dneva — to je tam, kjer smo na koncu pustili tri pike. Naloga zahteva pri najhladnejšem dnevu enak izpis kot pri najtoplejšem, tako da bi morali pravzaprav kar skopirati stavke, ki smo jih uporabili za izpis najtoplejšega dneva. V prejšnjem programu se s tem ukvarja kar sedem vrstic programa.

Toda takšno kopiranje stavkov je neugodno. Program je zaradi tega daljši in manj pregleden; človek, ki ga pogleda, ne more takoj vedeti, ali sta obe kopiji res enaki ali pa je med njima kakšna drobna (in mogoče pomembna) razlika. In če bomo hoteli kasneje pri izpisu kaj spremeniti ali pa odpraviti kakšno napako, bomo morali enako popraviti obe kopiji.

Namesto kopiranja takšne skupine stavkov je bolje iz njih narediti funkcijo, ki jo lahko potem pokličemo tako za izpis najtoplejšega kot za izpis najhladnejšega dneva.

Naša nova rešitev (glej program na desni) se začne s stavkom `def`, ki pomeni **deklaracijo funkcije**. Z njo pythonu povemo, kako naj bo naši funkciji ime (v tem primeru je to `Izpisi`) in kateri stavki naj jo sestavljajo (to je blok zamaknjenih stavkov, ki sledijo stavku `def`; tej skupini stavkov pravimo **telo funkcije**). Tolmač na tem mestu teh stavkov še ne bo poskušal izvesti, pač pa si jih le zapomni za kasneje.

V nadaljevanju programa imamo dvakrat vrstico `Izpisi()` — temu pravimo **klic funkcije**. Funkcijo torej pokličemo tako, da navedemo njeno ime in za njim zapišemo par oklepajev (v oklepajih lahko navedemo tudi parametre; več o tem na naslednji strani). Ko pride tolmač pri izvajanju programa do takšne vrstice, bo po vrsti izvedel stavke iz telesa funkcije, nato pa se izvajanje **vrne** iz funkcije nazaj v glavni blok programa in se nadaljuje z naslednjim stavkom za klicem.

```
1  def Izpisi():
2    danOd = naj - 3
3    danDo = naj + 3
4    if danOd < 0: danOd = 0
5    if danDo >= n: danDo = n - 1
6    vsota = 0
7    for d in range(danOd, danDo + 1):
8      vsota += meritve[d]
9    print(d, vsota / (danDo - danOd + 1))
10
11 # Preberimo vhodne podatke
12 n = int(input("Vnesi število dni: "))
13 meritve = []
14 for i in range(n):
15   t = int(input("Vnesi temperaturo: "))
16   meritve.append(t)
17
18 # Poiščimo dan z najvišjo temperaturo.
19 naj = 0
20 for d in range(n):
21   if meritve[d] > meritve[naj]: naj = d
22 # Izpišimo ga, kot zahteva besedilo naloge.
23 Izpisi()
24
25 # Poiščimo dan z najnižjo temperaturo.
26 naj = 0
27 for d in range(n):
28   if meritve[d] < meritve[naj]: naj = d
29 # Izpišimo ga, kot zahteva besedilo naloge.
30 Izpisi()
```

kompl_funk2.py

Parametri

Naša funkcija `Izpisi` iz prejšnjega primera je predpostavila, da so meritve shranjene v tabeli `meritve` in da je indeks dneva, ki nas zanima, shranjen v spremenljivki `naj`. Takšne predpostavke so neugodne, ker ustvarijo zelo tesno medsebojno odvisnost med klicateljem in funkcijo. Na primer, če bi hoteli nekoč kasneje na podoben način izpisati kakšen dan, ki ga nimamo ravno v spremenljivki `naj`, nam dosedanja funkcija `Izpisi` ne bi pri tem nič pomagala. Podobno, če bi nekoč kasneje v glavnem bloku programa preimenovali tabelo `meritve` v kaj drugega, naša funkcija ne bi več delovala (oz. bi morali tudi njo ustrezno popraviti).

Zato je bolje, če funkciji podatke, s katerimi bo morala delati, podamo kot **parametre**. Pri deklaraciji lahko v oklepajih navedemo, koliko parametrov funkcija zahteva in kakšna so njihova imena.

V deklaraciji (stavek `def`) zdaj piše, da zahteva `Izpisi` dva parametra z imenoma `dan` in `tabela`. Stavki v telesu funkcije ju lahko uporabljajo na enak način kot spremenljivke. Pri klicu pa smo v oklepajih navedli konkretni vrednosti, ki naj ju funkcija v tem konkretnem klicu dobi kot parametra: v našem primeru bo parameter `dan` dobil vrednost spremenljivke `naj`, parameter `tabela` pa bo dobil vrednost spremenljivke `meritve`.

Parametrom pod imeni, kot jih vidi funkcija, pravimo včasih tudi **formalni parametri**; konkretnim vrednostim, ki jih kot parametre podamo ob klicu, pa pravimo **dejanski parametri** ali **argumenti**. Formalne parametre lahko uporabljamo le znotraj funkcije, zunaj nje pa se jih ne vidi.

```
1  def Izpisi(dan, tabela):
2    danOd = dan - 3
3    danDo = dan + 3
4    if danOd < 0: danOd = 0
5    if danDo >= len(tabela): danDo = len(tabela) - 1
6    vsota = 0
7    for d in range(danOd, danDo + 1):
8      vsota += tabela[d]
9    print(d, vsota / (danDo - danOd + 1))
10
11 # Preberimo vhodne podatke
12 n = int(input("Vnesi število dni: "))
13 meritve = []
14 for i in range(n):
15   t = int(input("Vnesi temperaturo: "))
16   meritve.append(t)
17
18 # Poiščimo dan z najvišjo temperaturo.
19 naj = 0
20 for d in range(n):
21   if meritve[d] > meritve[naj]: naj = d
22 # Izpišimo ga, kot zahteva besedilo naloge.
23 Izpisi(naj, meritve)
24
25 # Poiščimo dan z najnižjo temperaturo.
26 naj = 0
27 for d in range(n):
28   if meritve[d] < meritve[naj]: naj = d
29 # Izpišimo ga, kot zahteva besedilo naloge.
30 Izpisi(naj, meritve)
```

Funkcije - napredno

- Lokalne vs. globalne spremenljivke

https://lusy.fri.uni-lj.si/ucbenik/book/1205/index6.html

- Vračanje rezultata iz funkcije

https://lusy.fri.uni-lj.si/ucbenik/book/1205/index7.html

- Gnezdenje klicev in rekurzivne funkcije

https://lusy.fri.uni-lj.si/ucbenik/book/1205/index9.html

- Funkcije in strukturirano programiranje

https://lusy.fri.uni-lj.si/ucbenik/book/1205/index11.html

Nekaj standardnih funkcij

Vsak programski jezik ima tudi nekakšno standardno knjižnico funkcij (angl.: *libraries*), ki so že del jezika in jih lahko uporabljamo, ne da bi jih morali napisati sami. Poleg že videnih funkcij kot sta ***input*** in ***print***, je tule še nekaj Pythonovih funkcij, ki pogosto pridejo prav:

`abs(x)`

vrne absolutno vrednost števila x.

`float(x)`

pretvori vrednost x v realno število.

`int(x)`

pretvori vrednost x v celo število.

`len(x)`

če je x niz, vrne dolžino tega niza (število znakov v njem); če je x zaporedje, vrne število elementov v njem.

`max(t)`

vrne največji element zaporedja t.

`min(t)`

vrne najmanjši element zaporedja t.

`round(x)`

vrne tisto celo število, ki je najbližje številu x.

`str(x)`

pretvori vrednost x v niz.

`sum(t)`

vrne vsoto elementov v zaporedju t.

6: Nizi

Nizi

Z računalnikom želimo včasih obdelovati podatke, ki jih sestavljajo znaki, besede, besedila in podobno. Takrat pridejo prav nizi; niz je podatkovni tip, pri katerem posamezno vrednost sestavlja zaporedje osnovnih elementov, ki jim pravimo znaki. Posamezni znak je lahko črka, števka, presledek ali še marsikaj drugega. Ogledali si bomo pogoste operacije na nizih, med drugim spreminjanje nizov, urejanje, iskanje podnizov. Nizi pridejo prav tudi pri branju vhodnih podatkov in izpisu rezultatov.

Niz je sestavljen iz zaporedja osnovnih elementov, ki jim pravimo znaki. Primer uporabe v resničnem življenju je npr. naslov osebe iz katerega želimo izluščiti kraj oz. mesto (to je t.i. podniz).

Niz

Doslej smo v naših programih večinoma delali s števili, v praksi pa pogosto želimo delati tudi s tekstovnimi podatki (besedami in besedili). Takrat pride prav podatkovni tip **niz**. Niz je zaporedje osnovnih elementov, ki jim pravimo **znaki** — to so črke, števke, presledki, ločila in podobno.

V programu zapišemo nize tako, da vsebino niza postavimo v narekovaje. Uporabimo lahko enojne ali dvojne narekovaje, važno je le, da se začetni in končni narekovaj ujemata. Če želimo kot znak v nizu uporabiti tudi narekovaj, moramo pred njega zapisati `\`; če pa hočemo kot znak v nizu uporabiti `\`, ga moramo zapisati dvojno:

```
1  s1 = "Ena dve"
2  print(s1) # izpiše: Ena dve
3  s2 = 'tri'
4  print(s2) # izpiše: tri
5  s3 = "a\"b\\c"
6  print(s3) #izpiše: a"b\c
7
8  a = "ena dve"
9  print(a)   # izpiše: ena dve
10 print("a") # izpiše: a
```

Narekovaji so pomembni, ker drugače tolmač ne bi mogel vedeti, kdaj skušamo zapisati niz, in bi vsebino niza poskusil izvesti oz. interpretirati (glej zadnje tri vrstice gornjega primera).

Vaja

Kaj se izpiše ob klicu `print("\\///\\/\\\\/")`?

Niz (nad.)

Niz lahko v mnogih pogledih uporabljamo podobno kot tabelo: če je `s` nek niz, lahko do znaka na indeksu `i` pridemo z izrazom `s[i]` (pri tem `i = 0` pomeni prvi znak niza, `i = 1` je drugi znak in podobno). Za razliko od tabele pa posameznih znakov v nizu ne moremo spreminjati.

Niz ima lahko tudi dolžino 0; takemu pravimo **prazen niz**. Dolžino niza `s` dobimo z `len(s)`.

```
1 s = "ababc"
2 print(s[1])   # izpiše: b
3 print(len(s)) # izpiše dolžino niza: 5
4 s[1] = "c" # napaka, niza se ne da spreminjati
```

Na nizih lahko uporabljamo nekatere operatorje, ki smo jih sicer navajeni uporabljati na številih. Operacija `s + t` izvede **stik** dveh nizov (dobimo nov niz, ki vsebuje najprej znake niza `s` in nato še znake niza `t`), z operacijo `s * n` pa dobimo niz, ki ga sestavlja `n` kopij niza `s`.

```
1 s = "abc"
2 t = "bd"
3 print(s + t) # izpiše: abcbd
4 s = s + t * 4
5 print(s) # izpiše: abcbdbdbdbd
```

trik_piram_funk.py

Niz (nad.)

Vaja

Napiši funkcijo `Trikotnik(n)`, ki izpiše trikotnik iz zvezdic: v prvi vrstici eno zvezdico, v drugi dve in tako naprej do *n*-te vrstice. Primer: `Trikotnik(4)` naj izpiše

```
*
**
***
****
```

Vaja

Napiši funkcijo `Piramida(n)`, ki izpiše piramido iz zvezdic: v prvi vrstici eno zvezdico, v drugi tri, v tretji pet in tako naprej do *n*-te vrstice. Primer: `Piramida(4)` naj izpiše

```
   *
  ***
 *****
*******
```

podniz_funk.py

Podniz

Strnjenemu podzaporedju znakov nekega niza pravimo tudi **podniz** tega niza. Če je `s` nek niz, je `s[i:j]` podniz tega niza, ki obsega vse znake na indeksih od `i` do `j - 1`. Če hočemo podniz, ki se začne na začetku niza, lahko `i` tudi opustimo; podobno, če hočemo podniz, ki se konča na koncu niza, lahko opustimo `j`. Uporabljati smemo tudi negativne indekse, pri čemer -1 pomeni zadnji znak niza, -2 predzadnji znak niza in tako naprej.

```
s = "abcde"
print(s[2:4]) # izpiše: cd
print(s[:4])  # izpiše: abcd
print(s[3:])  # izpiše: de
print(s[-2])  # izpiše: d
print(s[:-1]) # izpiše: abcd
print(s[-3:]) # izpiše: cde
```

Vaja

Napiši funkcijo `JePodniz(s, t)`, ki ugotovi, ali se `t` pojavlja kot podniz v nizu `s`. Primer: `JePodniz("banana", "ana")` mora vrniti `True`; `JePodniz("banana", "jana")` pa mora vrniti `False`.

Znaki in urejenost nizov

Posamezni znak niza ima tudi svojo številsko kodo. Med znakom (v resnici nizom dolžine 1) in njegovo kodo lahko pretvarjamo s funkcijama `chr` in `ord`.

To, katera koda pripada kateremu znaku, je določeno s standardom Unicode, o katerem bomo izvedeli več v enem od kasnejših poglavij tega učbenika.

```
1 print(ord("A")) # izpiše: 65
2 print(ord("B")) # izpiše: 66
3 print(ord("Z")) # izpiše: 90
4 print(ord("a")) # izpiše: 97
5 print(chr(65))  # izpiše: A
```

Vaja

(a) Napiši funkcijo `Kode(s)`, ki kot parameter dobi niz `s` in izpiše zaporedje številskih kod njegovih znakov.
(b) Ugotovi, katere kode pripadajo velikim črkam, malim črkam (tudi šumnikom), števkam in presledku.
(c) Izpiši po vrsti znake s kodami od 32 do 127.

S pomočjo teh številskih kod znakov lahko nize postavimo v nekakšen vrstni red, ki mu pravimo **leksikografski vrstni red**. V tem vrstnem redu primerjamo dva niza tako, da primerjamo istoležne znake obeh nizov, dokler ne naletimo na prvo neujemanje; tisti niz, ki ima tam znak z manjšo kodo, je v leksikografskem vrstnem redu pred drugim nizom. (Če sta niza različno dolga in pridemo prej do konca krajšega niza kot do kakšnega neujemanja, pa rečemo, da je krajši niz v leksikografskem vrstnem redu pred daljšim.)

Nize lahko glede na ta vrstni red primerjamo z operatorji `<`, `>`, `<=` in `>=`. Z operatorjema `==` in `!=` lahko tudi preverimo, če sta dva niza enaka oz. različna.

```
1 print("abc" < "abd")      # izpiše: True
2 print("abcd" > "abc")     # izpiše: True
3 print("aBc" < "abc")      # izpiše: True
4 print("acc" > "aBc")      # izpiše: True
5 print("cab" < "abd")      # izpiše: False
6 print("102" < "12")       # izpiše: True
7 print("12" < "ab")        # izpiše: True
8 print("ab cde" < "abcde") # izpiše: True
```

Operacije na nizih – Iskanje podnizov

Nizi v pythonu imajo precej funkcij, ki jih pokličemo tako, da navedemo niz, na katerem bi jo radi izvedli, nato zapišemo piko in ime funkcije. Tako imamo na primer funkciji `upper` in `lower`, ki vrneta nov niz, ki ga sestavljajo enaki znaki kot prvotnega, le da so vse črke spremenjene v velike oz. v male:

```
s = "aBc1"
print(s.upper()) # izpiše: ABC1
print("DeF2".lower()) # izpiše: def2
```

```
s = "abcdbca"
print(s.find("bc")) # izpiše 1
print(s.rfind("bc")) # izpiše 4
print(s.find("bc", 2)) # izpiše 4 (2. pojavitev)
print(s.find("bc", 5)) # izpiše -1
print(s.find("bce")) # izpiše -1
print(s.count("bc")) # izpiše 2
print("bc" in s) # izpiše True
print("bce" not in s) # izpiše True
print(s.startswith("ab")) # izpiše True
print(s.endswith("ab")) # izpiše False
```

Funkcija `find` vrne prvi indeks, na katerem se dani podniz pojavi v našem nizu (oz. vrne -1, če se sploh ne pojavi v njem). Načeloma išče pojavitve od začetka niza naprej, lahko pa ji podamo parameter, s katerim povemo, pri katerem indeksu naj začne iskati.

Obstaja tudi `rfind`, ki deluje enako kot `find`, le da išče od desne proti levi namesto od leve proti desni.

S funkcijo `count` lahko preštejemo pojavitve podniza v nizu. Če želimo le preveriti, ali se podniz sploh pojavlja v nizu, lahko uporabimo operatorja `in` oz. `not in`.

S funkcijama `startswith` in `endswith` pa lahko preverimo, ali se niz začne oz. konča z nekim podnizom.

Spet gre za objektno programiranje (glej poglavje Tabele): *objekt.metoda*

Vaja

Napiši funkcijo `JeSamoglasnik(s)`, ki predpostavi, da je kot parameter dobila niz dolžine 1, in vrne `True`, če je ta niz samoglasnik, sicer pa vrne `False`.

Operacije na nizih – Preverjanje znakov

Pogosto želimo preveriti, ali je nek znak (ali niz) črka, števka in podobno.

Nizi imajo precej funkcij, ki nam pri tem pomagajo:

- `isalpha` preveri, če je znak črka;
- `isdigit` preveri, če je števka;
- `isupper` in `islower` preverita, če je velika oz. mala črka;
- `isspace` preveri, če je presledek (sem štejejo poleg »pravega« presledka še nekateri drugi znaki, na primer tabulator in konec vrstice);
- `isalnum` preveri, če je znak črka ali števka.

Lahko jih uporabimo tudi na nizih, dolgih več kot 1 znak; tedaj preverijo, če so vsi znaki niza taki.

```
1  s = "Ab1"
2  print(s[0].isupper()) # True, A je velika črka
3  print(s[1].islower()) # True, b je mala črka
4  print(s[2].isdigit()) # True, 1 je števka
5  print(s.isalnum()) # True, vsi znaki črke/števke
6  print(" ".isspace()) # True
7
8  def PrestejCrke(s):
9    n = 0
10   for i in range(len(s)):
11     if s[i].isalpha():
12       n += 1
13   return n
14
15 print(PrestejCrke("Abeceda123"))
```

Operacije na nizih – Preverjanje znakov (nad.)

V funkciji `PrestejCrke` imamo zanko, ki se sprehodi po vseh znakih niza. Pravzaprav se s spremenljivko `i` sprehodi po vseh indeksih od `0` do `len(s) - 1`, nato pa do posameznih znakov pride z izrazom `s[i]`. Če nam indeksi niso pomembni, lahko gremo z zanko tudi neposredno po znakih niza:

```
def PrestejCrke(s):
  n = 0
  for c in s:
    if c.isalpha():
      n += 1
  return n
```

Če na primer pokličemo `PrestejCrke("1ab")`, bi bil `c` v prvi iteraciji zanke enak `"1"`, v drugi iteraciji `"a"` in v tretji `"b"`.

Vaja

Napiši funkcijo `NizVStevilo(s)`, ki kot parameter dobi niz z zapisom nekega naravnega števila v desetiškem zapisu; funkcija naj vrne to število. Primer: `NizVStevilo("123")` mora vrniti `123` (kot celo število, ne kot niz).

(Za to v pythonu seveda že obstaja funkcija `int` — namen te naloge pa je, da nekaj podobnega napišeš sam(a) brez uporabe te funkcije.)

Operacije na nizih – Zamenjave podnizov

Videli smo že, da lahko nize stikamo z operatorjem `+`. Če hočemo stakniti več nizov naenkrat, pride prav tudi funkcija `join`. Z njo staknemo vse nize iz dane tabele, mednje pa se vrine po ena kopija niza, na katerem smo funkcijo poklicali:

```
1 s = ", "
2 t = ["ena", "dve", "tri"]
3 u = "ena, dve, tri"
4 print(u == s.join(t)) # izpiše: True
```

Obratna operacija pa je `split`, ki razcepi niz pri vseh pojavitvah danega podniza; vrne nam tabelo tako dobljenih kosov prvotnega niza. Če jo pokličemo brez parametrov, razcepi niz pri presledkih.

```
1 u = "ena, dve, tri"
2 print(u.split(", "))
3 print(u.split())
```

Funkcija `strip` poreže presledke z začetka in konca niza; `lstrip` jih poreže le z začetka, `rstrip` pa le s konca.

```
1 s = "   abc   "
2 print(s.strip() == "abc")     # izpiše: True
3 print(s.lstrip() == "abc   ") # izpiše: True
4 print(s.rstrip() == "   abc") # izpiše: True
```

Funkcija `replace` pa zamenja vse pojavitve nekega podniza z nekim drugim podnizom. Kot vidimo v spodnjem primeru, `replace` pregleduje niz od leve proti desni in po vsaki zamenjavi nadaljuje šele za koncem pravkar zamenjanega podniza.

```
1 s = "banana in ananas"
2 print(s.replace("an", "xy"))
3 print(s.replace("ana", "xy"))
4 print(s.replace("ana", "xna"))
```

7: Razno – N-terice, Slovarji in Moduli

Elementi množice: n-terice

V pythonu so elementi tabele lahko kakršnegakoli tipa, elementi množic pa morajo biti **nespremenljivega tipa** (*ang. immutable type*). Spremenljivke nespremenljivega tipa lahko zgolj ustvarjamo, ne moremo pa spreminjati njihovih vrednosti. Tip `int`, denimo, je nespremenljiv, čeprav python dopušča zaporedje stavkov

```
a = 7
a = a + 3
```

ki navidez spremeni vrednost spremenljivke `a`. Vendar pa se v resnici ustvari **nova** spremenljivka (nov prostorček v pomnilniku), ki se nato ponovno poimenuje kot `a`, stara spremenljivka `a` pa se izgubi. Tipi `int`, `float`, `bool` in `str` (nizi) so nespremenljivi, tabele pa niso, saj, denimo, stavek `t[1] = 7` spremeni element **obstoječe** tabele. Zato tabele ne morejo nastopati kot elementi pythonove množice.

Vendar pa python ponuja nespremenljivo alternativo tabelam: ***n*-terice** (podatkovni tip `tuple`). *n*-terice se obnašajo enako kot tabele, le da jih zapisujemo v okroglih oklepajih, njihovih vrednosti pa ne moremo spreminjati:

```
1 a = (50, 20, 70, 60)
2 print(a[1])   # izpiše 20
3 print(a[1:3]) # izpiše (20, 70)
4 # Spodnja dva poskusa spremembe n-terice
5 # sprožita izjemo.
6 a[1] = 10
7 a.append(30)
```

type(a) = tuple

Ob poskusu spremembe *n*-terice se sproži izjema. Zaradi svoje nespremenljivosti *n*-terice lahko shranjujemo v množico.

n-terico z dvema elementoma bomo imenovali **par**, *n*-terico s tremi elementi **trojica** itd. *n*-terici z enim elementom ne bomo dali posebnega imena, omeniti pa velja, da jo zapišemo tako:

```
(element,)
```

Če vejico na koncu izpustimo, bo python izraz tolmačil kot vrednost *element* v (odvečnih) oklepajih.

Vaja

Napiši izraz, ki ustvari množico s štirimi elementi: številom `-23.4`, nizom `'juha'`, trojico prvih treh praštevil in *n*-terico z elementom `True`.

SLOVAR

```
# h je slovar s tremi ključi
h = {"ena": 1, "dve": 2, "tri": 3}
# g je prazen slovar (brez ključev)
g = {}
```

type(h) = dict

Spremenljivka `h` po tem primeru vsebuje slovar s tremi ključi, in sicer nizi `"ena"`, `"dve"` in `"tri"`. Pripadajoča vrednost pri ključu `"ena"` je celo število 1 in tako naprej.

Slovar je podatkovna struktura, ki si jo lahko predstavljamo kot razširitev tabele in množice. Slovar vzdržuje množico parov oblike ⟨*ključ*, *vrednost*⟩, pri čemer so vsi ključi v slovarju različni. Vsak par je zato enolično določen že s svojim ključem in vse operacije na njem izvajamo prek tega ključa. Ključe (in pripadajoče vrednosti) lahko dodajamo v slovar, brišemo iz njega, lahko pa za dani ključ preverimo, če je prisoten v slovarju, in preberemo ali spremenimo njegovo pripadajočo vrednost.

Slovar je zato zelo podoben množici, pravzaprav ga dobimo tako, da v množici ob vsakem elementu (ključu) shranimo še pripadajočo vrednost. Po načinu uporabe pa je slovar zelo podoben tudi tabeli: pri tabeli imamo zbirko vrednosti, do katerih dostopamo prek indeksov, indeksi pa so majhna cela števila od 0 naprej. Če poznamo indeks, lahko hitro in učinkovito preberemo ali zapišemo vrednost na tistem indeksu v tabeli. Pri slovarju je podobno, le da namesto indeksov za dostopanje do vrednosti uporabljamo ključe, ki so načeloma lahko poljubnega tipa, ne le majhna cela števila. Ključi so lahko na primer nizi (npr. imena in priimki), velika cela števila (kot je npr. EMŠO, ki je 13-mestno število), datumi, urejene *n*-terice in tako naprej.

Ime »slovar« izvira iz dejstva, da v tej strukturi lahko na podlagi ključa poiščemo pripadajočo vrednost, podobno kot v običajnem slovarju na podlagi besede poiščemo njeno razlago ali prevod v drug jezik. Še eno ime za to podatkovno strukturo pa je »preslikava«, ker nam preslika ključ v pripadajočo vrednost podobno kot preslikave v matematiki.

Operacije nad slovarjem

Podobno kot pri množici tudi pri slovarju ključi nimajo nobenega posebnega vrstnega reda. V naslednjem primeru sta slovarja `h` in `g` na koncu enaka:

```
1 h = {"ena": 1, "dve": 2, "tri": 3}
2 g = {"dve": 2, "tri": 3, "ena": 1}
3 print(h == g) # izpiše True
```

Tudi če slovar izpišemo s `print`, se ključi ne izpišejo nujno v kakšnem posebnem vrstnem redu.

Do vrednosti, ki v slovarju pripada določenemu ključu, lahko pridemo z oglatimi oklepaji. Na enak način lahko to vrednost tudi spremenimo:

```
1 h = {"ena": 1, "dve": 2, "tri": 3}
2 a = h["dve"]
3 print(a) # izpiše 2
4
5 h["dve"] = 222
6 print(h["dve"]) # izpiše 222
```

Če poskušamo brati vrednost ključa, ki ga ni v slovarju, se bo sprožila izjema:

```
1 h = {"ena": 1, "dve": 2, "tri": 3}
2 a = h["štiri"] # napaka, ključa "štiri"
3                # ni v slovarju!
```

Če pa poskušamo vpisati vrednost za ključ, ki ga še ni v slovarju, ga bo računalnik ob tej priliki dodal v slovar:

```
1 h = {"ena": 1, "dve": 2, "tri": 3}
2 h["štiri"] = 44
3 print(h) # izpiše vsebino seznama
```

Operacije nad slovarjem (nad.)

Še en način za branje vrednosti v slovarju pa je funkcija `get`, ki ji iskani ključ podamo kot parameter. Če ključa ni v slovarju, funkcija ne sproži napake, pač pa vrne `None`. Podpira pa tudi dodatni drugi parameter, s katerim ji povemo, kaj naj nam vrne, če zahtevanega ključa v slovarju ni:

```
1 h = {"ena": 1, "dve": 2, "tri": 3}
2 print(h.get("dve"))      # izpiše 2
3 print(h.get("dve", 123)) # izpiše 2
4 print(h.get("pet"))      # izpiše None
5 print(h.get("pet", 456)) # izpiše 456
```

Še ena koristna operacija je brisanje ključa iz slovarja. V pythonu to naredimo s stavkom `del`:

```
1 h = {"ena": 1, "dve": 2, "tri": 3}
2 del h["ena"]
3 print(h)     # izpiše {"dve": 2, "tri": 3}
4 a = h["ena"] # napaka, ključa "ena" ni v slovarju
```

Z operatorjema `in` in `not in` lahko preverimo, če je nek ključ prisoten v slovarju ali ne, podobno kot pri množicah. S funkcijo `len` pa lahko dobimo število ključev v slovarju:

```
1 h = {"ena": 1, "dve": 2, "tri": 3}
2 print("ena" in h) # izpiše True
3 print("štiri" in h) # izpiše False
4 print("štiri" not in h) # izpiše True
5
6 print(len(h)) # izpiše 3
```

Pogosto pride prav tudi sprehod po vseh ključih slovarja. To lahko naredimo z zanko `for`:

```
1 h = {"ena": 1, "dve": 2, "tri": 3}
2 for k in h:
3   print("Ključ: " + str(k))
4   print("Vrednost: " + str(h[k]))
```

PYTHONOV MODUL

V tej učni enoti se bomo prvič srečali s pythonovimi moduli. Modul je pythonova datoteka, ki je prvenstveno namenjena uvažanju v druge pythonove datoteke, ne samostojnemu zaganjanju. Če modul uvozimo, lahko kličemo funkcije, ki so zapisane v njem. Modul nam tako omogoča, da iste funkcije kličemo v več različnih pythonovih datotekah (samostojnih programih ali drugih modulih). Oglejmo si preprost primer. V datoteko `matematika.py` zapišimo sledeči funkciji:

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

def razlika(a, b):
  return a - b
```

V datoteko `test.py`, ki se mora nahajati v isti mapi kot datoteka `matematika.py`, pa zapišimo sledeče stavke:

```
# Uvozimo modul matematika.py
import matematika
# Sedaj lahko kličemo funkcije iz modula matematika.py
print(matematika.vsota(3, 4))
print(matematika.razlika(7, 2))
```

Če program `test.py` poženemo, se seveda izpišeta števili 7 in 5.

Stavek `import` ima več možnih oblik. Oblika

```
import matematika as mat
```

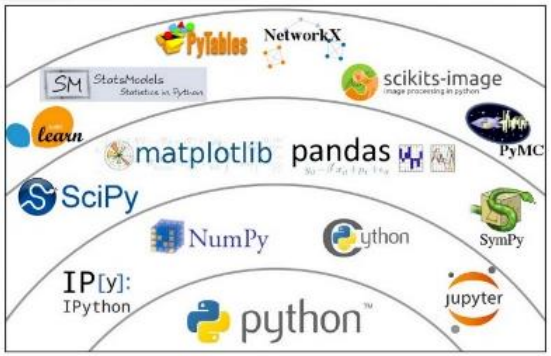
nam omogoča, da namesto `matematika.vsota(...)` in `matematika.razlika(...)` pišemo kar `mat.vsota(...)` in `mat.razlika(...)`.

Če pa uporabimo obliko

```
from matematika import vsota, razlika
```

lahko pišemo kar `vsota(...)` in `razlika(...)`.

Trije moduli v okviru standardne knjižnice (Standard Library; glej Global Module Index v Pythonovi dokumentaciji): random, math, turtle



Python v praksi – primeri aplikacij

8: Generativna umetnost in računalniške igre

Računalniška umetnost

- Nekaj primerov:
  - http://blog.wispeo.com/featured-creative/glenn-marshall-creating-art-from-the-python-language/
  - https://labs.ideo.com/2014/06/04/painting-with-code/

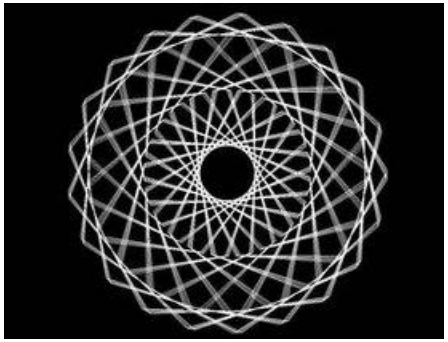
- Za profesionalno (odprtokodno) delo se uporablja program Processing …
  - https://processing.org/
  - Možno je tudi programiranje v Pythonu – Processing.py: http://py.processing.org/

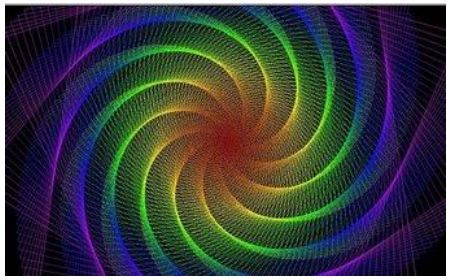
- … ali pa Pygame – računalniške igre, umetnost, multimedijski projekti:
  - http://pygame.org/hifi.html

genumet1.py, genumet2.py, genumet3.py

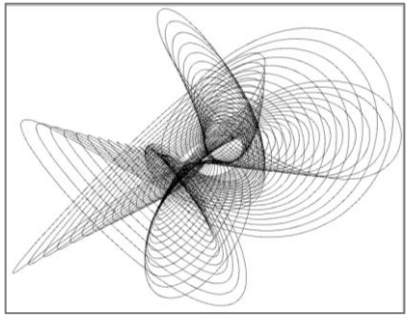
Pythonov modul Turtle

- Omogoča enostavno izdelavo kompleksnih oblik in slik
  - https://michael0x2a.com/blog/turtle-examples
  - Python IDLE > Help > Turtle Demo

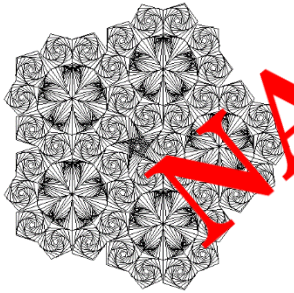




http://www.instructables.com/id/Easy-Designs-Turtle-Graphics-Python/



http://python3.codes/extensible-harmonograph/



Python 3.5.2 Shell > Help > Turtle Demo

http://codegolf.stackexchange.com/questions/36374/redraw-an-image-with-just-one-closed-curve

NA VAJAH!!!

Izbor metod modula Turtle

Method	Parameters	Description
Turtle	None	Creates and returns a new turtle object
forward	amount	Moves the turtle forward by the specified amount
backward	amount	Moves the turle backward by the specified amount
right	angle	Turns the turtle clockwise
left	angle	Turns the turtle counter clockwise
penup	None	Picks up the turtle's pen
pendown	None	Puts down the turtle's pen
up	None	Picks up the turtle's pen
down	None	Puts down the turtle's pen
color	color name	Changes the color of the turtle's pen
fillcolor	color name	Changes the color of the turtle will use to fill a polygon
heading	None	Returns the current heading
position	None	Returns the current position
goto	x,y	Move the turtle to position x,y
begin_fill	None	Remember the starting point for a filled polygon
end_fill	None	Close the polygon and fill with the current fill color
dot	None	Leave a dot at the current position
stamp	None	Leaves an impression of a turtle shape at the current location
shape	shapename	Should be 'arrow', 'classic', 'turtle', or 'circle'

http://interactivepython.org/runestone/static/IntroPythonTurtles/Summary/summary.html

- Spisek vseh metod:
  - https://docs.python.org/2/library/turtle.html

9: OBDELAVA IN VIZUALIZACIJA PODATKOV

Podatkovna znanost in obdelava podatkov

- 'Data Science' knjižnice
  - NumPy – znanstveni izračuni, delo z matrikami (Arrays)
  - SciPy – temelji na NumPy; linearna algebra, optimizacija, statistika
  - Pandas – 1- in 2-dimenzionalne podatkovne strukture (Data Frames)

- Strojno učenje
  - scikit-learn – klasifikacija, regresija, razvrščanje v skupine, redukcija dimenzionalnosti, modeliranje
  - TensorFlow
  - Keras
  - PyTorch

  } globoko učenje

- Obdelava slike
  - scikit-image
  - Pillow – osnovne funkcije
  - OpenCV – napredno slikovno procesiranje, računalniški vid

VIZUALIZACIJA PODATKOV

- Matplotlib
- Bokeh
- Seaborn
- Plotly
- ggplot2

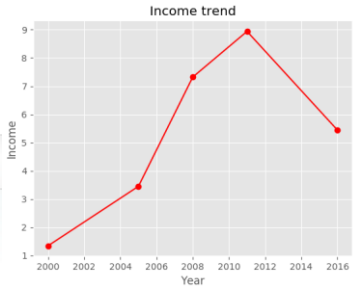
The Python ecosystem for Data Science: A guided tour - Christian Staudt
https://www.youtube.com/watch?v=EBgUiuFXE3E

Paketi / Knjižnice

- Temeljno orodje je knjižnica **Matplotlib**. Nanjo se naslanja več drugih knjižnic za vizualizacijo, med drugim **Pandas**, **Seaborn** in **ggplot**, nekatere druge pa omogočajo enostavno pretvorbo vizualizacij iz Matplotlib, npr. **Bokeh** in **Plotly**, ki vizualizacijam dodata visoko raven interaktivnosti, ki jo omogoča tudi **Pygal**. Vsako od omenjenih orodij ima svoje značilnosti. Tako je v Pandas poudarek na upravljanju s podatki, zato ga uporabljamo v kombinaciji s skoraj vsemi ostalimi knjižnicami. Druga pogosto uporabljena knjižnica je **Numpy** s funkcijami za numerično računstvo, npr. za generiranje podatkov, in z matematičnimi funkcijami. Knjižnica Seaborn se uporablja za vizualizacije statističnih obdelav. Dodatna orodja združljiva z Matplotlib omogočajo izdelavo 3D grafov (modul **Mplot3d**), prikaz podatkov na zemljevidih (modul **Basemap** in knjižnica **Cartopy**) in vključevanje interaktivnih gradnikov (v Pythonu modul **widgets** knjižnice Matplotlib, v Ipythonu knjižnica **Ipywidgets**). Visoko interaktivno orodje za prikaz geografskih podatkov je **Folium**. Knjižnico Bokeh lahko v strežniškem načinu dela uporabljamo za upravljanje in vizualizacijo velikih količin podatkov (big data). To orodje vsebuje možnost vključitve geografskih podatkov. Posebni knjižnici za numerično obdelavo slik sta **Scipy** in orodjarna **Scikit-image**, za nekatere osnovne operacije nad slikami pa uporabimo Numpy. Vizualizacije prikažemo na zaslon, lahko pa jih shranimo v slikovne datoteke (navadno v png format) ali celo v html format (v Bokeh).

MATPLOTLIB

```python
import matplotlib.pyplot as plt
plt.style.use('ggplot')
x = [2000, 2005, 2008, 2011, 2016]
y = [1.34, 3.45, 7.32, 8.94, 5.44]
plt.plot(x, y, 'r-o')
plt.title('Income trend')
plt.xlabel('Year')
plt.ylabel('Income')
plt.show()
```



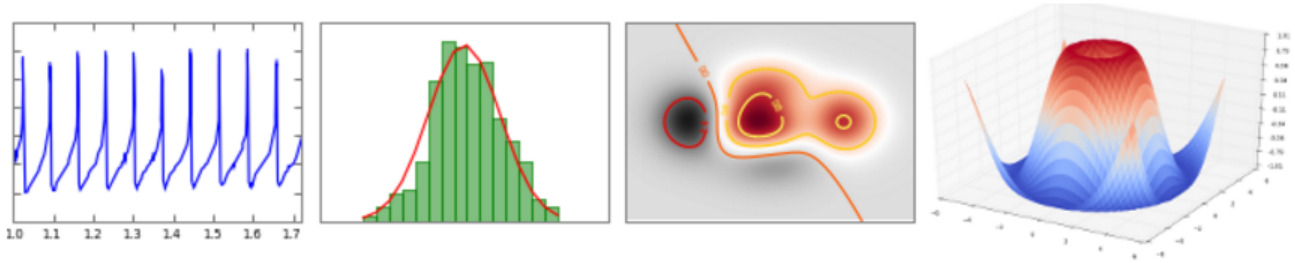
matplotlib

home | examples | gallery | pyplot | docs »

http://matplotlib.org/

Introduction

matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB®* or Mathematica®†), web application servers, and six graphical user interface toolkits.



matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code. For a sampling, see the screenshots, thumbnail gallery, and examples directory

For simple plotting the `pyplot` interface provides a MATLAB-like interface, particularly when combined with `IPython`. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

NUMPY, PANDAS

NumPy

Scipy.org

http://www.numpy.org/

NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

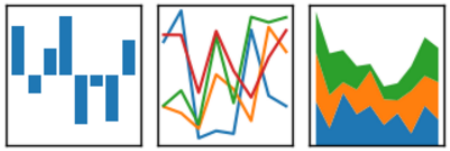
- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

NumPy is licensed under the BSD license, enabling reuse with few restrictions.

pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$



overview // get pandas // documentation // community

Python Data Analysis Library

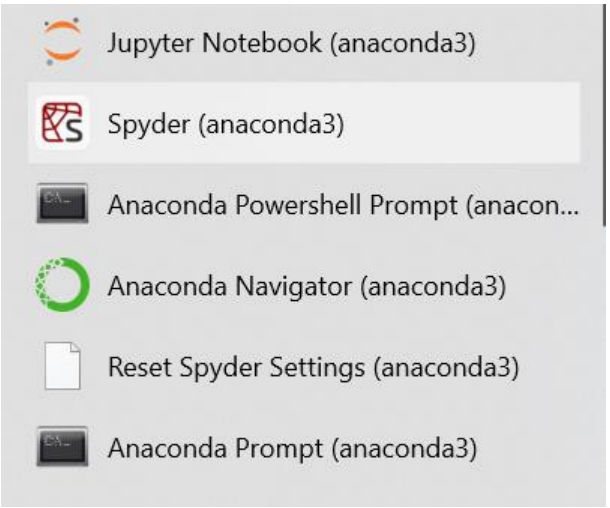
pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

http://pandas.pydata.org/

ANACONDA

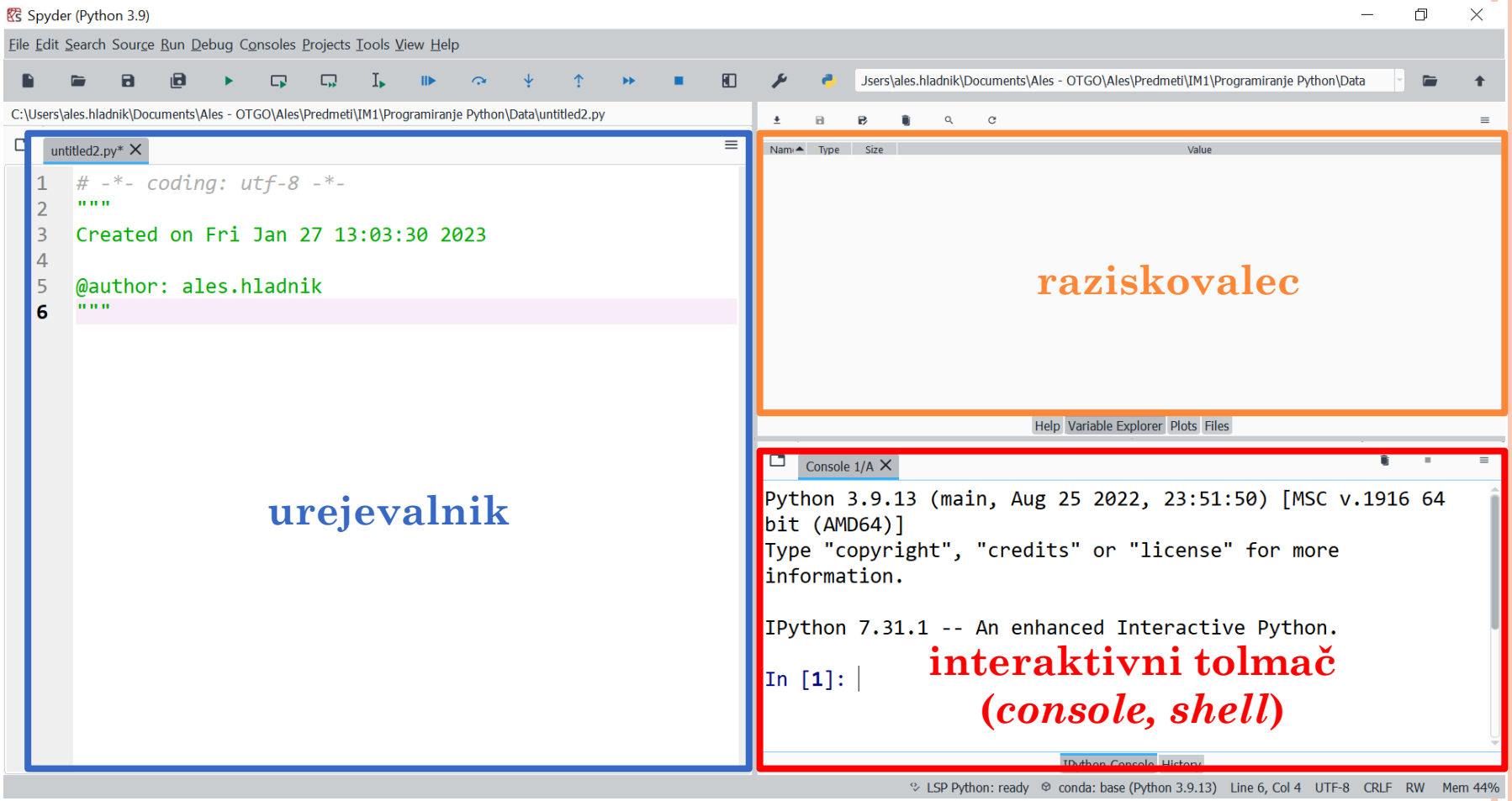
- Anaconda je ena od Pythonovih distribucij in vsebuje več kot 100 najrazličnejših paketov (= knjižnic in modulov) za obdelavo podatkov v Pythonu – vključno z večino paketov za vizualizacijo
  - Namestitev: https://www.anaconda.com/download

- Ob namestitvi so nam na voljo tudi različni vmesniki za delo s Pythonom …



- … mi bomo v nadaljevanju uporabljali orodje **Spyder** (Scientific PYthon Development EnviRonment)

SPYDER


Spyder (Python 3.9)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Jsers\ales.hladnik\Documents\Ales - OTGO\Ales\Predmeti\IM1\Programiranje Python\Data
C:\Users\ales.hladnik\Documents\Ales - OTGO\Ales\Predmeti\IM1\Programiranje Python\Data\untitled2.py
untitled2.py*
-*- coding: utf-8 -*-
"""
Created on Fri Jan 27 13:03:30 2023
@author: ales.hladnik
"""
urejevalnik
Name Type Size Value
raziskovalec
Help Variable Explorer Plots Files
Console 1/A
Python 3.9.13 (main, Aug 25 2022, 23:51:50) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.
IPython 7.31.1 -- An enhanced Interactive Python.
In [1]:
interaktivni tolmač
(console, shell)
IPython Console History
LSP Python: ready
conda: base (Python 3.9.13)
Line 6, Col 4
UTF-8
CRLF
RW
Mem 44%

Izdelava enostavnega diagrama

- Uvozimo/naložimo paketa *pandas in matplotlib.pyplot* in ju poimenujemo kot *pd* in *plt* :

```
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
```

- Preberemo podatke iz Excelove datoteke 'podatki1.xlsx' na disku in jih naložimo (uvozimo) v podatkovno strukturo tipa *DataFrame* v spremenljivko *tabela*:

```
>>> tabela = pd.read_excel('podatki1.xlsx') *
>>> tabela
```

```
   name         x         y        y2
0     A  0.704715  3.523573  0.867978
1     B  0.356913  1.784567  0.782291
2     C  0.043230  0.216149  0.024473
3     D  0.019762  0.098812  0.988338
4     E  0.761426  3.807131  0.551990
5     F  0.307774  1.538870  0.852090
6     G  0.633166  3.165828  0.551740
7     H  0.178966  0.894830  0.420789
8     I  0.173878  0.869392  0.582029
9     J  0.381444  1.907221  0.696369
```

	A	B	C	D
1	name	x	y	y2
2	A	0,704715	3,523573	0,867978
3	B	0,356913	1,784567	0,782291
4	C	0,04323	0,216149	0,024473
5	D	0,019762	0,098812	0,988338
6	E	0,761426	3,807131	0,55199
7	F	0,307774	1,53887	0,85209
8	G	0,633166	3,165828	0,55174
9	H	0,178966	0,89483	0,420789
10	I	0,173878	0,869392	0,582029
11	J	0,381444	1,907221	0,696369
12				

* Funkcije za uvoz surovih (npr. CSV) podatkov: pd.read_csv, numpy.loadtxt, numpy.genfromtext

Izdelava enostavnega diagrama (nad.)

- Sedaj lahko podatke filtriramo, obdelujemo, analiziramo, npr.:

```
>>> tabela_filt = tabela[tabela['x'] > 0.3]
>>> tabela_filt
```

	name	x	y	y2
0	A	0.704715	3.523573	0.867978
1	B	0.356913	1.784567	0.782291
4	E	0.761426	3.807131	0.551990
5	F	0.307774	1.538870	0.852090
6	G	0.633166	3.165828	0.551740
9	J	0.381444	1.907221	0.696369

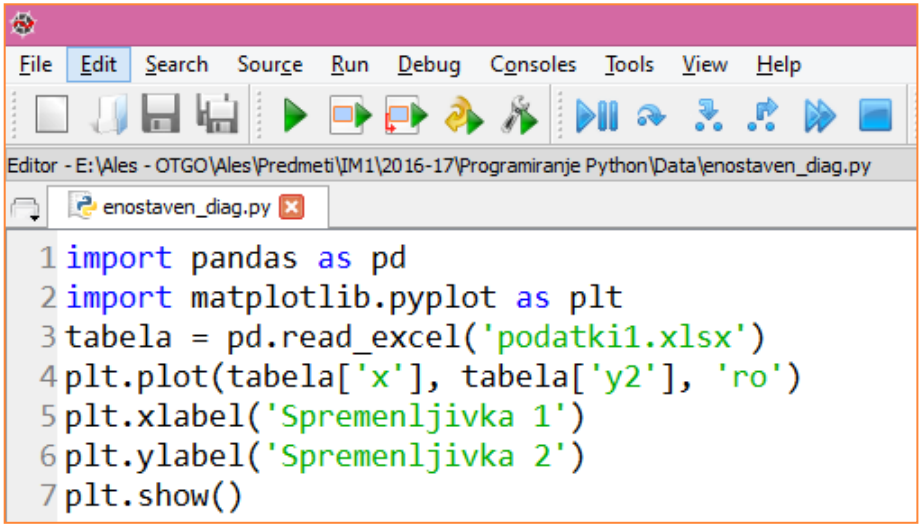
Tabela vsebuje le tiste vrstice, pri katerih je izpolnjen pogoj x > 0.3

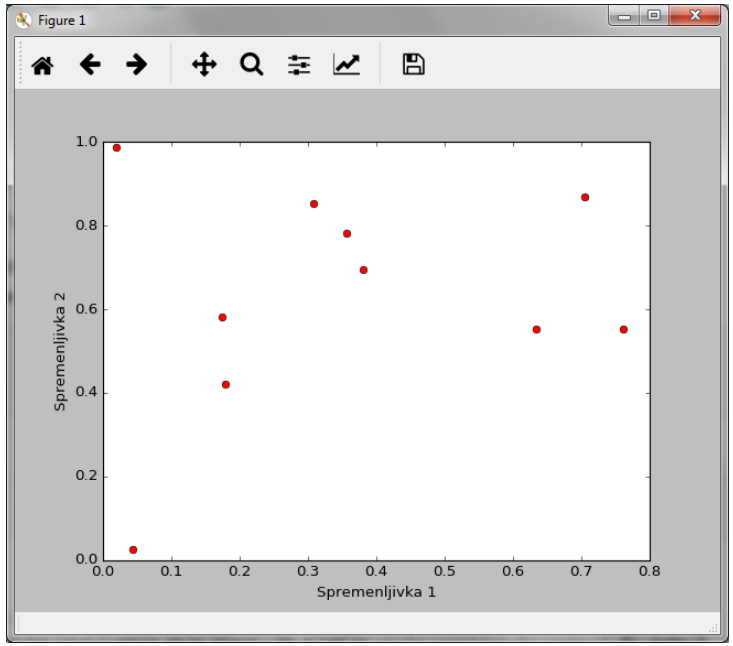
```
>>> y_povp = tabela_filt['y'].mean()
>>> y_povp
2.621198418600945
```

Izpis povprečja vseh šestih vrednosti stolpca y

Izdelava enostavnega diagrama (nad.)

- Ker gre za daljši niz Pythonovih ukazov, jih je smiselno napisati v urejevalniku, shraniti v datoteko in pognati (*Run* > *Run*; bližnjica F5) v interaktivnem tolmaču:





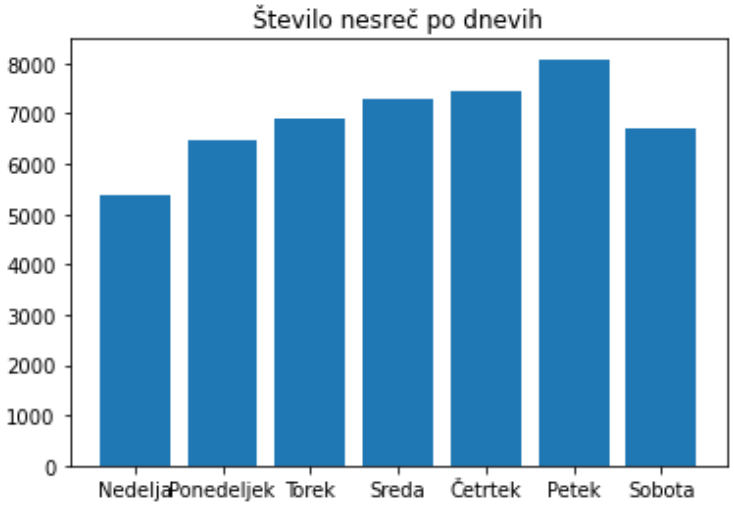
- Razpršeni diagram 'Spremenljivka 1' vs. 'Spremenljivka 2' se prikaže v oknu Raziskovalca (zavihek 'Plots')

Analiza in vizualizacija večje tabele

- Kompleksnejši primer vizualizacije ("Big Data"):
  - https://realpython.com/blog/python/working-with-large-excel-files-in-pandas/
  - Analizirali bomo manjšo podatkovno tabelo 'Road Safety Data-Accidents 2022-Provisional Mid Year Unvalidated Data', pridobljene s spletne strani https://www.data.gov.uk/dataset/cb7ae6f0-4be6-4935-9277-47e5ce24a11f/road-safety-data:

    https://data.dft.gov.uk/road-accidents-safety-data/dft-road-casualty-statistics-accident-provisional-mid-year-unvalidated-2022.csv

```python
# Analiza in vizualizacija "Big Data" (s. 92)
# vir: https://realpython.com/blog/python/working-with-large-excel-files-in-pandas/
# ---------------------------------------------------------------------------

# Uvozimo potrebne knjižnice
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
podatki = pd.read_excel('dft-road-casualty-statistics-accident-provisional-mid-year-

# Koliko je vseh vrstic v tabeli?
print('Vseh vrstic:', len(podatki))

# Kako se glasijo imena stolpcev?
print(list(podatki))

# Koliko nesreč se je zgodilo ob sobotah?
nesrece_sobota = podatki[podatki.day_of_week == 7]
print('Nesrece, ki so se zgodile na soboto:', (len(nesrece_sobota)))

# Koliko nesreč se je zgodilo ob deževnih sobotah?
nesrece_sobota_dez = podatki[(podatki.day_of_week == 7) &
    (podatki.weather_conditions == 2)]
print('Nesrece, ki so se zgodile na deževno soboto:', len(nesrece_sobota_dez))

# Koliko nesreč se je zgodilo ob deževnih sobotah in so bili v njih
# udeleženi vsaj 3 avtomobili?
nesrece_sobota_dez_3avti = podatki[
    (podatki.day_of_week == 7) & (podatki.weather_conditions == 2) &
    (podatki.number_of_vehicles >= 3)]
print("Nesrece, ki so se zgodile na deževno soboto, udeleženih > 2 avtov:",
    len(nesrece_sobota_dez_3avti))
```



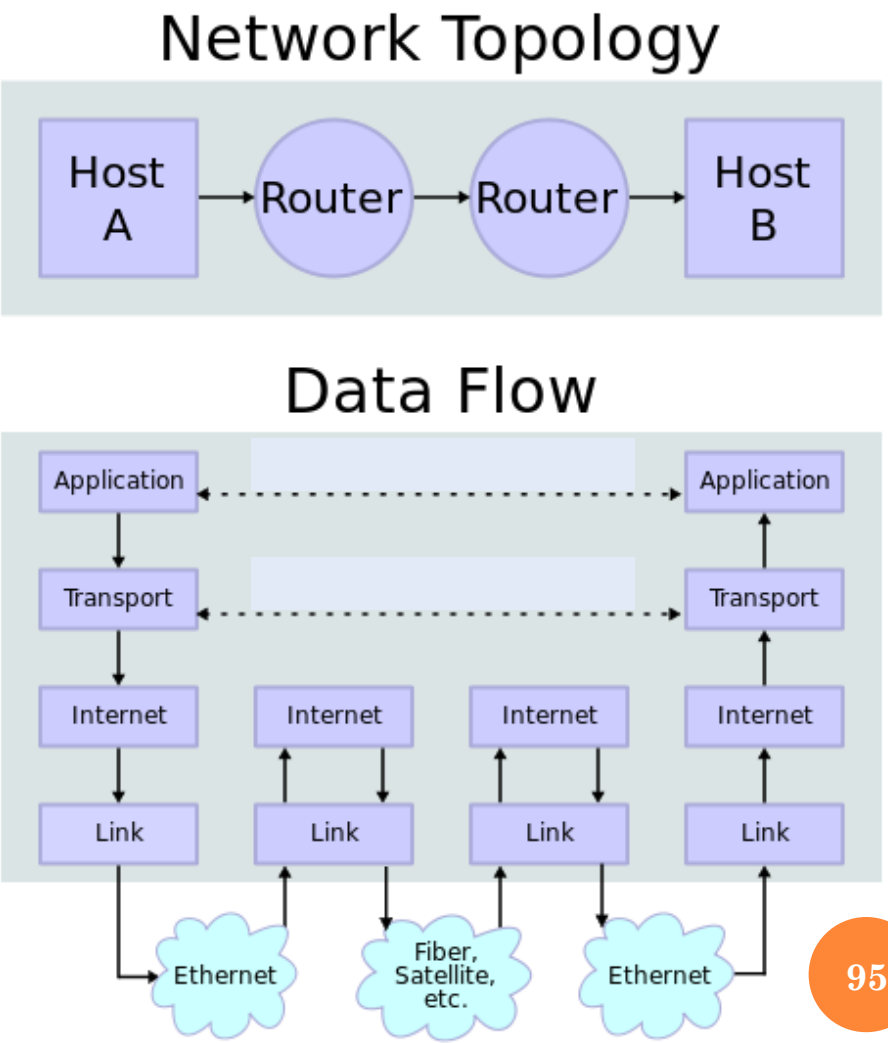
10: Luščenje spletnih podatkov (*HTML parsing*, *Web scraping*)

Spletno luščenje podatkov

- Podatki na spletnih straneh so dostikrat na voljo v uporabniku prijazni obliki (PDF, XLS(X), CSV…)
- Včasih pa so informacije, ki nas zanimajo, razpršene po večjem številu spletnih strani, vsebujejo nepomembne/moteče podatke, se pogosto spreminjajo, ipd.
- V teh primerih lahko uporabimo različna orodja za "Web scraping"
- Tudi s Pythonom lahko učinkovito iščemo po spletnih straneh in ekstrahiramo relevantne podatke
- Najprej ponovimo, kako delujeta internet in splet!

Internetni sklad protokolov

- Štirinivojska arhitektura (TCP/IP model)



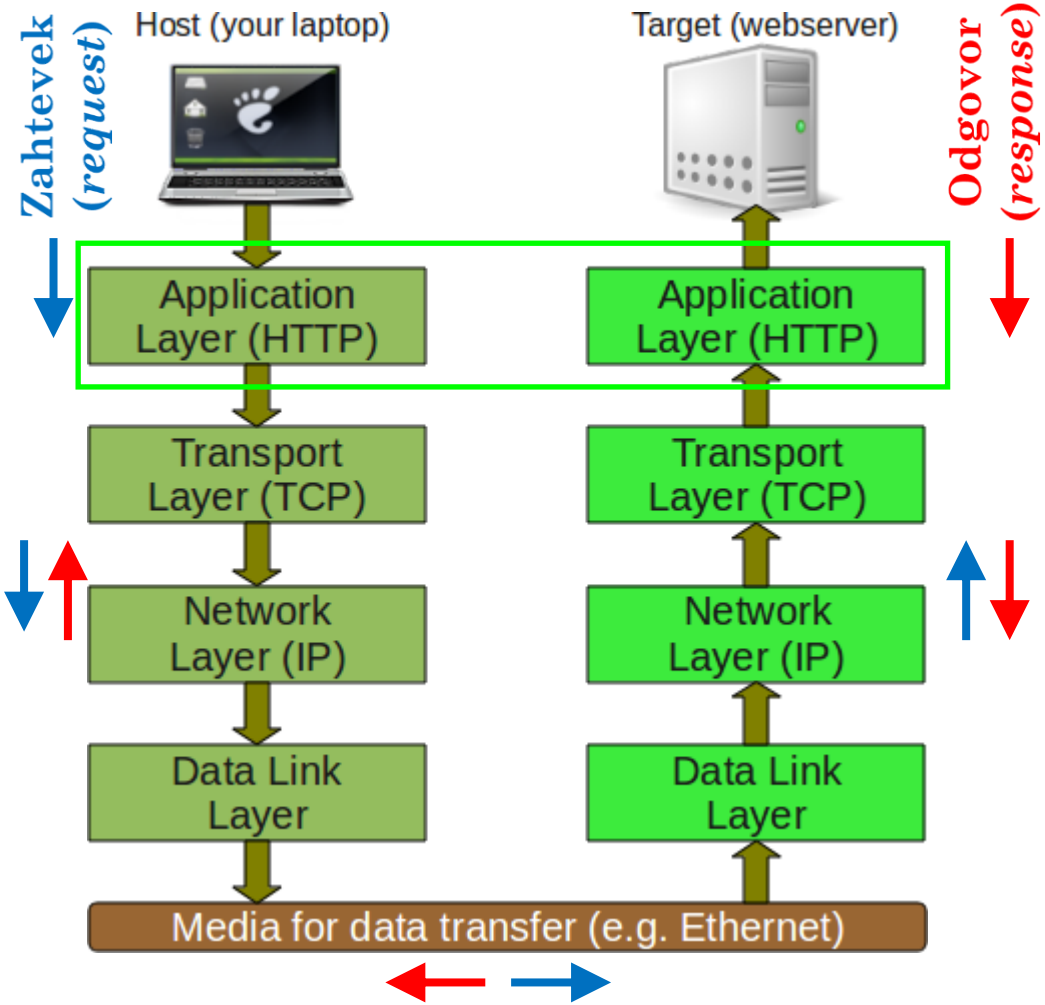
Aplikacijski nivo: HTTP, FTP, SMTP …

Transportni nivo: TCP, UDP …

Omrežni nivo: IP

Povezovalni nivo: Ethernet, Wi-Fi …

SPLETNI PROTOKOL: HTTP



http://www.thegeekstuff.com/2011/11/tcp-ip-fundamentals

- HTTP: osnovni aplikacijski protokol razvit za dostopanje do spletnih vsebin
- Osnovni princip:
  1. Poveži se s strežnikom
  2. Pošlji zahtevek (*request*)
  3. Pridobi dokument(e)
  4. Zapri povezavo s strežnikom

Kaj dela spletni brskalnik?

- Vsakič, ko uporabnik klikne na hiperpovezavo, da bi prišel na novo spletno stran, brskalnik vzpostavi povezavo s spletnim strežnikom in izda "GET" zahtevo (= *request*) s ciljem pridobiti vsebino strani z določenega spletnega naslova (URL-ja)
- Strežnik vrne (= *response*) HTML dokument brskalniku, ki ga oblikuje in prikaže uporabniku

- Namesto brskalnika pa lahko za pridobivanje in prikaz podatkov z oddaljenega strežnika uporabimo tudi druge načine
  - Telnet (nas ne zanima ☺)
  - Python

splet_podatki1.py

PYTHON KOT SPLETNI BRSKALNIK

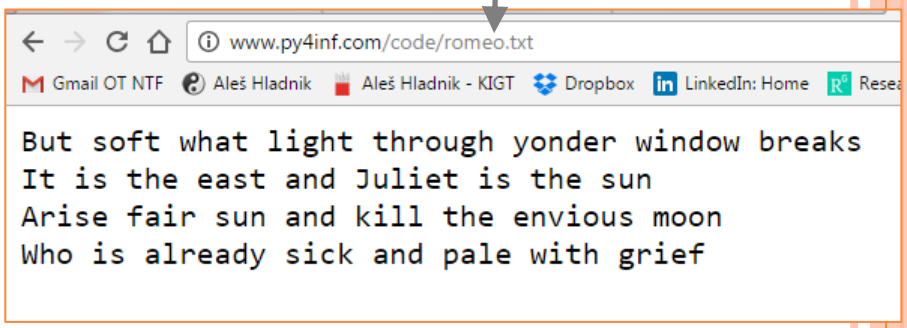
- Prvi način: 10 vrstic kode

```
1 import socket
2 mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3 mysock.connect(('www.py4inf.com', 80))
4 mysock.send(b'GET http://www.py4inf.com/code/romeo.txt HTTP/1.0\n\n')
5 while True:
6     data = mysock.recv(512)
7     if (len(data) < 1) :
8         break
9     print(data)
10 mysock.close()
```

Tekstovna, ne HTML datoteka !

- Drugi način: 3 vrstice kode (!!!)

```
1 import requests
2 r = requests.get('http://www.py4inf.com/code/romeo.txt')
3 print(r.text)
```



- Kdo sploh še potrebuje spletni brskalnik ☺

Orodja in opozorilo

- Requests (za pridobivanje spletnega dokumenta)
  - https://requests.readthedocs.io/en/master/#
- BeautifulSoup4 (za ekstrakcijo oz. razčlenjevanje (*parsing*) posameznih delov dokumenta)
  - https://www.crummy.com/software/BeautifulSoup/
- RegEx (napredno…)
  - https://docs.python.org/3.6/library/re.html

- Pri spletnem luščenju moramo biti previdni!
  - Legalni problemi (tožbe, sodni pregoni)
  - Če je na voljo API, ga uporabljajmo!
  - Slabo napisana HTML (CSS, JavaScript, XML…) koda ovira uspešno pridobivanje spletnih podatkov

Primer št. 1

splet_podatki2.py

- Web Scraping 101 – Introduction
  - https://www.youtube.com/watch?v=6UkI-VF6h0k&list=PLozj871qTroOuALRdOGF5cUBol5ov7fNN

Primer št. 2

splet_podatki3.py

- Scrape Websites with Python + Beautiful Soup 4 + Requests -- Coding with Python
  - https://www.youtube.com/watch?v=3xQTJi2tqgk&list=PLozj871qTroOuALRdOGF5cUBol5ov7fNN&index=2

Seminarska naloga: Napotki za izdelavo

- Skupaj s še dvema sošolcema si izberite projekt*, pri katerem boste uporabili vsaj tri Pythonove knjižnice (npr. Numpy, Pandas in MatplotLib)
- Namestite (*install*) potrebne pakete/knjižnice/module – uporabite distribucijo Anaconda in/ali opcijo pip install – da jih boste lahko uvozili (*import*) in uporabili v svoji skripti
- Napišite program (kodo), v katerem boste implementirali posamezne korake projekta
- V seminarski nalogi in predstavitvi jih prikažite – pojasnite dele kode in predstavite rezultate v obliki diagramov, tabel, vizualizacij…

** Lahko si pomagate s kakim že izdelanim, ki je na voljo na spletu – v tem primeru v seminarski nalogi obvezno navedite njegov URL naslov! – a ga čim bolj priredite v skladu z vašimi preferencami in znanjem programiranja. Naj bo projekt v čim večji meri vaše avtorsko delo!*

Zbirke podatkov in primeri uporabe

- Prosto dostopne podatkovne zbirke
  - Kaggle: https://www.kaggle.com/datasets
  - Več možnosti: https://careerfoundry.com/en/blog/data-analytics/where-to-find-free-datasets/

- Problem trgovskega potnika: https://towardsdatascience.com/solving-travelling-salesperson-problems-with-python-5de7e883d847

- Detekcija barv s pomočjo knjižnic Pandas & OpenCV: https://data-flair.training/blogs/project-in-python-colour-detection/

- Python za oblikovalce (samo za MacOS !): https://pythonfordesigners.com/

LITERATURA

- G. Anželj in sod.: Računalništvo in informatika 1 – E-učbenik za informatiko v gimnaziji, 2017. Splet: http://lusy.fri.uni-lj.si/ucbenik/
- Dokumentacija Python ver. 3. Splet: https://docs.python.org/3/

Tutoriali, tečaji in priročniki o Pythonu

- SoloLearn. Splet: https://www.sololearn.com/Play/Python
- J. Demšar: Python za programerje. Splet: http://goo.gl/PK1rUZ
- Beginner's Guide to Python. Splet: https://wiki.python.org/moin/BeginnersGuide