Preface

Razlika med programskimi jeziki (interpreted, complied, ...)

To make use of an algorithm in a computer, we must first convert it to a program. We do this by using a programming language (a very formal language with strict rules about spelling and grammar) which the computer is able to convert unambiguously into computer instructions, or machine language.

The reason that we do not write computer instructions directly is that they are difficult for humans to read and understand. For example, these are the computer instructions (in the Intel 8086 machine language, a subset of the Intel Pentium machine language) required to add 17 and 20:

```
1011 0000 0001 0001
0000 0100 0001 0100
1010 0010 0100 1000 0000 0000 ```
```

The first line tells the computer to copy 17 into the AL register: the first four characters (1011) tell the computer to copy information into a register, the next four characters (0000) tell the computer to use register named AL, and the last eight digits (0001 0001, which is 17 in binary) specify the number to be copied.

As you can see, it is quite hard to write a program in machine language. In the 1940s, the programmers of the first computers had to do this because there were no other options! To simplify the programming process, assembly language was introduced.

Each assembly instruction corresponds to one machine language instruction, but it is more easily understood by humans, as can be seen in the equivalent addition program in the 8086 assembly language:

```
MOV AL, 17D
ADD AL, 20D
MOV [SUM], AL ```
```

Programs written in assembly language cannot be understood by the computer directly, so a translation step is needed. This is done using an assembler, whose job it is to translate from assembly language to machine language.

> Although assembly language was a great improvement over machine language, it can still be quite cryptic, and it is so low-level that the simplest task requires many instructions. High-level languages were developed to make programming even easier.

> In a high-level language, an instruction may correspond to several machine language instructions, which makes programs easier to read and write. This is the Python equivalent of the code above:

```
sum = 17 + 20
```

Compilers, interpreters and the Python programming language

> Programs written in high-level languages must also be translated into machine language before a computer can execute them. Some programming languages translate the whole program at once and store the result in another file which is then executed. Some languages translate and execute programs line-by-line. We call these languages compiled languages and interpreted languages, respectively. Python is an interpreted language.

> A compiled language comes with a compiler, which is a program which compiles source files to executable binary files. An interpreted language comes with an interpreter, which interprets source files and executes them. Interpretation can be less efficient than compilation, so interpreted languages have a reputation for being slow.

> Programs which need to use a lot of computer resources, and which therefore need to be as efficient as possible, are often written in a language like C. C is a compiled language which is in many ways lower-level than Python – for example, a C programmer needs to handle a lot of memory management explicitly; something a Python programmer seldom needs to worry about.

> This fine-grained control allows for a lot of optimisation, but can be time-consuming and error-prone. For applications which are not resource-intensive, a language like Python allows programs to be developed more easily and rapidly, and the speed difference on a modern computer is usually negligible

Python je visoko-nivojski, interpretiran programski jezik.

Visoko-nivojski pomeni, da je naš program napisan v bolj človeško berljivem jeziku, vendar pa ga je nato potrebno še pretvoriti v bolj nizko-nivojski (oziroma, računalniku lažje berljiv) jezik.

Interpretirani programski jezik pomeni, da se vsaka vrstica našega programa posebej interpretira in pretvori v ukaze za naš računalnik, tekom izvajanja programa. Nasprotno temu obstajajo **compiled** programski jeziki, kjer se celoten program pretvori v ukaze za naš računalnik, nato pa se program lahko izvede. Prednost interpretiranega programskega jezika je hitrejše pisanje programa, saj ni potrebno ponovno compile-at celotne kode za vsako majhno spremembo.

Glavni razlogi za popularnost Pythona so, da je zelo preprost in lahko berljiv. Navklub temu, pa omogoča pisanje zelo kompleksne kode. Python je znan po tem, da je vsaka stvar objekt in, da ima dinamične spremenljivke - ista spremenljivka je lahko uporabljena za shranjevanje različnih data tipov.

Množična uporaba se je začela po letu 2000, z izdajo verzije 2.0 in z izdajo verzije 3.0 v letu 2008. Verziji med seboj nista popolnoma kompatibilni in omenit je treba, da se verzijo 2.x ne bo več vzdrževalo od leta 2020 naprej.

> A rough estimate of the complexity of a language can be gleaned from the number of keywords or reserved words in the language. These are words that are reserved for special meaning by the compiler or interpreter because they designate specific built-in functionality of the language. Python 3 has 33 keywords, and Python 2 has 31. By contrast, C++ has 62, Java has 53, and Visual Basic has more than 120, though these latter examples probably vary somewhat by implementation or dialect.

> What does it mean to be **intepreted lanugage**: Many languages are compiled, meaning the source code you create needs to be translated into machine code, the language of your computer's processor, before it can be run. Programs written in an interpreted language are passed straight to an interpreter that runs them directly (line by line). This makes for a quicker development cycle because you just type in your code and run it, without the intermediate compilation step. One potential downside to interpreted languages is execution speed. Programs that are compiled into the native language of the computer processor tend to run more quickly than interpreted programs. For some applications that are particularly computationally intensive, like graphics processing or intense number crunching, this can be limiting. In practice, however, for most programs, the difference in execution speed is measured in milliseconds, or seconds at most, and not appreciably noticeable to a human user. The expediency of coding in an interpreted language is typically worth it for most applications.

> Python 2.0 was released in 2000, and the 2.x versions were the prevalent releases until December 2008. At that time, the development team made the decision to release version 3.0, which contained a few relatively small but significant changes that were not backward compatible with the 2.x versions. Python 2 and 3 are very similar, and some features of Python 3 have been backported to Python 2. But in general, they remain not quite compatible.

> Both Python 2 and 3 have continued to be maintained and developed, with periodic release updates for both. As of this writing, the most recent versions available are 2.7.15 and 3.6.5. However, an official End Of Life date of January 1, 2020 has been established for Python 2, after which time it will no longer be maintained. If you are a newcomer to Python, it is recommended that you focus on Python 3, as this tutorial will do.

Installing Python

Da se prične s programiranjem moramo imeti inštaliran Python Interpreter oziroma lahko uporabimo Online Python Interpreter.

Uporabljali bomo Python3.x verzijo.

Windows

Preverimo, če imamo že inštaliran Python:

- Odpremo CMD
- vpišemo `python --version`
  - Če piše “python is not recognized as an internal or external command…. Potem nimamo inštaliranega Python internpreterja

Inštalacija:

- python.org
- Najdemo za željeni operacijski sistem. Zdownloadamo najnovejšo različico 3.x verzije
  - Embedded zip file - to je, da ti extractaš v svojo datoteko in je to to
  - Executable - da ti inštalira in nrdi path itd..
- ADD Python to PATH!

Problemi:

- Če ma windows že inštalirano verzijo Python (primer: python2.7). Če probaš ukaz “python --version” uporabi verzijo 2.7, čeprov smo glihkr dodal verzijo 3.7. Lahko se proba z drugimi ukazi za Python3.7 (python3, python3.7, py3, py…). Lahko se zamenja “PATH VARIABLE” sam jst tega ne znam. Če ne druzga se lahko pomaknejo v mapo kjer je ta Python3.7 inštaliran in tm začenejo CMD in delajo normalno.

Linux

There is a very good chance your Linux distribution has Python installed already, but it probably won’t be the latest version, and it may be Python 2 instead of Python 3.

To find out what version(s) you have, open a terminal window and try the following commands:

python --version

python2 --version

python3 --version

One or more of these commands should respond with a version, as below:

$ python3 --version Python 3.6.5 If the version shown is Python 2.x.x or a version of Python 3 that is not the latest (3.6.5 as of this writing), then you will want to install the latest version. The procedure for doing this will depend on the Linux distribution you are running.

MacOS

While current versions of macOS (previously known as “Mac OS X”) include a version of Python 2, it is likely out of date by a few months. Also, this tutorial series uses Python 3, so let’s get you upgraded to that.

The best way we found to install Python 3 on macOS is through the Homebrew package manager. This approach is also recommended by community guides like The Hitchhiker’s Guide to Python.

Step 1: Install Homebrew (Part 1) To get started, you first want to install Homebrew:

Open a browser and navigate to http://brew.sh/ (http://brew.sh/). After the page has finished loading, select the Homebrew bootstrap code under “Install Homebrew”. Then hit Cmd+C to copy it to the clipboard. Make sure you’ve captured the text of the complete command because otherwise the installation will fail. Now you need to open a Terminal.app window, paste the Homebrew bootstrap code, and then hit Enter. This will begin the Homebrew installation. If you’re doing this on a fresh install of macOS, you may get a pop up alert asking you to install Apple’s “command line developer tools”. You’ll need those to continue with the installation, so please confirm the dialog box by clicking on “Install”. At this point, you’re likely waiting for the command line developer tools to finish installing, and that’s going to take a few minutes. Time to grab a coffee or tea!

Step 2: Install Homebrew (Part 2) You can continue installing Homebrew and then Python after the command line developer tools installation is complete:

Confirm the “The software was installed” dialog from the developer tools installer. Back in the terminal, hit Enter to continue with the Homebrew installation. Homebrew asks you to enter your password so it can finalize the installation. Enter your user account password and hit Enter to continue. Depending on your internet connection, Homebrew will take a few minutes to download its required files. Once the installation is complete, you’ll end up back at the command prompt in your terminal window. Whew! Now that the Homebrew package manager is set up, let’s continue on with installing Python 3 on your system.

Step 3: Install Python Once Homebrew has finished installing, return to your terminal and run the following command:

$brewinstallpython3Note: Whenyoucopythiscommand, besureyoudon't includethe$ character at the beginning. That’s just an indicator that this is a console command.

This will download and install the latest version of Python. After the Homebrew brew install command finishes, Python 3 should be installed on your system.

You can make sure everything went correctly by testing if Python can be accessed from the terminal:

Open the terminal by launching Terminal.app. Type pip3 and hit Enter. You should see the help text from Python’s “Pip” package manager. If you get an error message running pip3, go through the Python install steps again to make sure you have a working Python installation. Assuming everything went well and you saw the output from Pip in your command prompt window…congratulations! You just installed Python on your system, and you’re all set to continue with the next section in this tutorial.

Text-Editor

Visual Studio Code https://code.visualstudio.com/download (https://code.visualstudio.com/download)

Python extension - **TODO** https://code.visualstudio.com/docs/python/python-tutorial (https://code.visualstudio.com/docs/python/python-tutorial)

> Zdownloadaš Python extension imenovan Python izdan s strani Microsoft. Izbereš Python Interpreter -> `Ctrl+Shift+P` in vtipkaš `Python: Select Interpreter` . Nato izberemo naš željen Python Interpreter.

In [ ]:

```
```

Začel bomo s čisto preprostim programom.

Naloga: 5x zapored izpiši neko številko

Najbolje je, če si program razdelimo na čim manjše "koščke".

V našem primeru rabimo vedeti:

- ukaz, ki izpiše neko število
- ta ukaz ponoviti 5x

Za izpis nečesa se uporablja beseda

```
print()
```

In [1]:

```
print(10)
```

```
10
```

Če je naprimer naša naloga 5x izpisati številko 1.

To bi lahko napisal na sledeč način:

In [2]:

```
print(1)
print(1)
print(1)
print(1)
print(1)
```

```
1
1
1
1
1
```

Kar se tukaj zgodi je, da python interpreter vzeme prvo vrstico in izpiše številko znotraj oklepaja.

Nato se premakne v naslednjo vrstico in izvede ukaz v tej vrstici, ki ponovno izpiše številko znotraj oklepaja.

In [ ]:

```

```

Tekom programiranja se odločmo, da hočmo namesto številke 1 izpisat številko 3.

In zdej gremo v vsako vrstico in zamenjamo 1 z 3.

In [3]:

```
print(3)
print(3)
print(3)
print(3)
print(3)
```

```
3
3
3
3
3
```

Če bi bil naš program, da moramo neko številko izpisat 1000x bi potem na roke moral popravljat vsako 1ko v 3ko. Kar pa je zamudno in lahko povzroči veliko število človeških napak (ponesreč izpustimo 1 vrstico, itd..)

Dosti lažje bi bilo, če bi mi lahko na začetku računalniku povedal, naj si shrani številko katero hočemo izpisat. Potem pa jo računalnik izpiše 100x.

To lahko dosežemo s pomočjo spremeljivk.

In [ ]:

```

```

In [ ]:

```

```

Spremenljivke

Spremenljivka je kot neka beseda v katero shranimo vrednost in do te vrednosti dostopamo kasneje v kodi.

```
x = 2
```

Beri: **Vrednost 2 shrani v spremenljivko z imenom x.**

Oziroma bolj natančno: **Ovrednoti kar je na desni strani enačaja in to shrani v levo stran enačaja**

Spremenljivke nam omogočajo shranjevanje vrednosti in lepšo kontrolo nad kodo.

Napišimo naš primer z uporabo spremenljivke.

In [5]:

```
x = 3 # definiramo našo spremenljivko in vanjo shranimo našo vrednost, katero želim
print(x)
print(x)
print(x)
print(x)
print(x)
```

```
3
3
3
3
3
```

Če sedaj hočemo, da se namesto številke 3 izpiše številka 12 lahko enostavno popravimo 1 vrstico.

In [6]:

```
x = 12
print(x)
print(x)
print(x)
print(x)
print(x)
```

```
12
12
12
12
12
```

Da je koda lažje berljiva, tudi po tem, ko nekdo drug bere za tabo, obstaja nek skupek priporočil kako naj bo koda zapisana (PEP8 (https://www.python.org/dev/peps/pep-0008/?)). Not recmo piše, da nej se spremenljivke poimenuje z uporabo snake_case (vse je z malimi začetnicai, besede ločimo z podčrtajem)

Pri imenu spremenljivk je tudi treba paziti, saj so case-sensitive.

In [7]:

```
x = 1
X = 2
print(x)
print(X)
```

```
1
2
```

Prav tako spremenljivk ne moremo poimenovati s posebnimi imeni ("keywords") katere Python že uporablja (False, None,...).

In [8]:

```
False = 1
```

```
  File "<ipython-input-8-1950c547d36b>", line 1
    False = 1
          ^
SyntaxError: can't assign to keyword
```

V Pythonu so spremenljivke **dinamične**. To pomeni, da nam ni potrebno izrecno povedati računalniku kakšnega tipa je spremenljivka.

Da vidimo kakšnega tipa je spremenljivka, uporabimo besedo:

```
type()
```

In [1]:

```
x = 1
print(x)
print(type(x)) # type(x) nam pove kakšnega tipa je spremenljivka x

print("------------")

x = 1.2
print(x)
print(type(x))
```

```
1
<class 'int'>
------------
1.2
<class 'float'>
```

Poznamo več različnih tipov spremenljivk, različnih vrednosti katere lahko shranimo:

Integer (celo število) - int

V Python3 ni maximalne velikosti integerja. Številka je lahko velika kolikor želimo. Omejeni smo samo z našim pomnilnikom.

In [3]:

```
x = 5
print(x)
print(type(x))
```

```
5
<class 'int'>
```

In [4]:

```
x = 126765060022822940149670320537 6
print(x)
print(type(x))
```

```
126765060022822940149670320537 6
<class 'int'>
```

Floating-point (decimalno število) - float

Float predstavlja decimalno ševilo (število s plavajočo vecijo).

Treba je pazit saj te številke niso popolnoma natančne ampak le aproksimacije (te aproksimacije se vidjo šele pri n-ti decimalki).

> Almost all platforms represent Python float values as 64-bit “double-precision” values, according to the IEEE 754 standard. In that case, the maximum value a floating-point number can have is approximately 1.8 × 10^^308.

> The closest a nonzero number can be to zero is approximately 5.0 × 10-324. Anything closer to zero than that is effectively zero:

> Floating point numbers are represented internally as binary (base-2) fractions. Most decimal fractions cannot be represented exactly as binary fractions, so in most cases the internal representation of a floating-point number is an approximation of the actual value. In practice, the difference between the actual value and the represented value is very small and should not usually cause significant problems.

In [5]:

```
x = 5.43
print(x)
print(type(x))
```

```
5.43
<class 'float'>
```

Complex numbers (kompleksna števila) - complex

Nam predstavlja kompleksna števila. Števila, ki so sestavljena iz realnega in imaginarnega dela.

In [6]:

```
x = 2 + 3j
print(x)
print(type(x))
```

```
(2+3j)
<class 'complex'>
```

Boolean (True or False) - bool

Boolean spremenljivka lahko zavzeme samo 2 vrednosti. Ali True ali False.

In [8]:

```
x = True
print(x)
print(type(x))

print("-------")

x = False
print(x)
print(type(x))
```

```
True
<class 'bool'>
-------
False
<class 'bool'>
```

Tudi, če spremenljivka sama po sebi ni True ali False, se jo še vedno lahko pretvorivmo v tip bool. Tako lahko vidimo, da so naslednje vrednosti False:

- Boolean False
- numerična vrednost 0 (0, 0.0, 0+0j...)
- Empty string
- Keyword None
- Empty object (kot je prazen list, prazna terka...)

Vse ostalo je True.

Da pretvorivmo neko spremenljivko v boolean tip, uporabimo besedo:

```
bool(spemenljivka)
```

In [40]:

```
print(bool(False)) # bool(x) pretvor vrednost x v boolean (al true al false)
print(bool(0))
print(bool(""))
print(bool(None))
print(bool([]))
print("********")
print(bool(True))
print(bool(1))
print(bool("abc"))
print(bool([1,2]))
```

```
False
False
False
False
False
********
True
True
True
True
```

Na podoben način lahko spreminjamo spremenljivke v ostale tipe:

```
int(spremenljivka),
str(spremenljivka),
complex(spremenljivka)
```

String (stavek) - str

Stringi so zaporedja črk. Začnejo in končajo se z dvojnim (") ali enojnim (') narekovajem.

Vsebuje lahko neomejeno število črk. Edina omejitev je naš pomnilnik.

Lahko je tudi prazen stavek.

In [13]:

```
x = "Stavek" # navaden string z dvojnim narekovajem ""
print(x)
print(type(x))
```

```
Stavek
<class 'str'>
```

In [15]:

```
x = 'String' # navaden string z enojnim narekovajem ''
print(x)
print(type(x))
```

```
String
<class 'str'>
```

In [16]:

```
x = "" # prazen string
print(x)
print(type(x))
```

```
<class 'str'>
```

Če želimo v našem stringu uporabiti narekovaje naredimo to tako:

In [17]:

```
x = "String with (')"
y = 'String with (")'
print(x)
print(y)
```

```
String with (')
String with (")
```

Večina črk ima samo 1, primarni pomen. In to je dejanska črka. **A** pomeni **A**, **e** pomeni **e**, itd.

Določene črke pa imajo tudi sekundarni pomen. Če pred črko vstavimo *backslash* (\ ) s tem povemo Pythonu, naj uporabi njen sekundarni pomen.

- **n** primarni pomen je črka n. Njen sekundarni pomen (**\n**) pomeni "premik v novo vrstico".
- **t** primarni pomen je črka t. Njen sekundarni pomen (**\t**) pomeni "tabulator".
- **** primerni pomen je sporočilo Pythonu naj uporabi sekundarni pomen črke. Njen sekundarni pomen (****) pomeni črka \ (backslash)

In [19]:

```
x = "String with (\")"
print(x) # ponavadi bi python prebral drugi " kot konec stringa

print("----------")

x = "String \nString"
print(x) # ponavadi bi python prebral n kot n. Ampak z \ ga ne prebere tko kot pona
```

```
String with (")
----------
String
String
```

Obstaja tudi možnost večvrstičnega izpisa.

In [20]:

```
print('''
To je primer večvrstičnega izpisa.
Vrstica 1
Vrstica 2 ''')
```

```
To je primer večvrstičnega izpisa.
Vrstica 1
Vrstica 2
```

Input() funkcija

S pomočjo te funkcije lahko uporabnika vprašamo za nek input.

In [54]:

```
#Calculating months
age = input('Enter your age: ')  # Enter 3
print('You have lived for', age, "years.")
```

```
Enter your age: 12
You have lived for 12 years.
```

Potrebno je paziti, ker nam input vrne vrednost datatipa **string**.

In [55]:

```
age = input("Enter your age: ")
print(type(age))
print(age)
```

```
Enter your age: 12
<class 'str'>
12
```

Vaje

Vaja 01

Naloga: S funkcijo print() in type() izpišite po eno spremenljivko tipa boolean, integer, float, complex in string.

In [56]:

```
a = True
b = 2
c = 3.4
d = 1 +9j
e = "neki"

print(type(a))
print(a)
print()

print(type(b))
print(b)
print()

print(type(c))
print(c)
print()

print(type(d))
print(d)
print()

print(type(e))
print(e)
print()
```

```
<class 'bool'>
True

<class 'int'>
2

<class 'float'>
3.4

<class 'complex'>
(1+9j)

<class 'str'>
neki
```

Vaja 02

Naloga: V neko spremenljivko shranite poljubno float vrednost. Izpišite spremenljivko in njen tip.

To spremenljivko pretvorite v boolan vrednost in to vrednost shranite v novo spremenljivko. Izpišite novo spremenljivko in njen tip.

In [21]:

```
x = 1.2
print(type(x))
print(x)
print()

y = bool(y)
print(type(y))
print(y)
print()
```

```
<class 'float'>
1.2

<class 'bool'>
True
```

Vaja 03

Naloga: Uporabnika zaprosite naj vnese neko celo število.

To vrednost shranite v spremenljivko z imenom **n** in jo izpišite in izpišite njen tip.

Nato to vrednost pretvorite v float vrednost. Dobljeno float vrednost shranite v spremenljivko **n**. Nato **n** izpišite in izpišite njen tip.

In [22]:

```
n = input("Vnesite celo število: ")

print(type(n))
print(n)
print()


n = float(n)
print(type(n))
print(n)
```

```
Vnesite celo število: 3
<class 'str'>
3

<class 'float'>
3.0
```

Izpisovanje in formating

Da nekaj izpišemo uporabimo besedo

```
print()
```

In [60]:

```
print("Hello World")
```

```
Hello World
```

S prihodom Python3.6 verzije se stringe izpisuje s pomočjo f-string

```
f'Besedilo {spremenljivka1:format1}, besedilo naprej{spremenljivka2:format2
}, besedilo naprej....'
```

Dokumentacija f-string (https://docs.python.org/3.6/library/string.html#formatspec)

In [61]:

```
ime = "Anže"
starost = 10
print(f'{ime} je {starost} let star')
```

```
Anže je 10 let star
```

Še primer kako lahko oblikujemo naš format izpisa.

In [62]:

```
ime = "Anže"
starost = 10
print(f'{ime:-^10} je {starost:*>10.3f} let star, oziroma {starost*12:e} mesecev.')
# {ime:-^10} ime -> spremenljivka, "-" -> znak s katerim zapolni mesta, "^" -> naj
# {starost:*>10.3f} starost -> ime spremenljivke, "*" -> znak s katerim zapolni mes
# {starost*12:e} starost*12 -> spremenljivka ki jo želimo izpisat, "e" -> naj bo st
```

```
---Anže--- je ****10.000 let star, oziroma 1.200000e+02 mesecev.
```

Pred tem, s prihdom Python2.6, se je uporabljalo

```
str.format()
```

In [63]:

```
ime = "Anže"
starost = 10
print("Živjo {}. Star si {} let.".format(ime, starost))
```

```
Živjo Anže. Star si 10 let.
```

.format() je počasnejši od f' ' stavka

Še pred tem se je uporabljalo

```
%-formating
```

In [1]:

```
name = "Anže"
age = 10
print("Živjo %s. Star si %s let." % (name, age))
```

```
Živjo Anže. Star si 10 let.
```

Ta način je najpočasnejši. Pri veliki količini spremenljivk hitro postane nepregleden. Lahko vodi do napak, kot so nepravilno prikazovanje touples in dictionaries.

String operacije

Nad string-i lahko izvajamo tudi različne operacije.

Vse črke stringa lahko pretvorimo v male črke, oziroma velike črke.

In [2]:

```
my_str = "Živjo Anže. Star si 10 let."
print(my_str) # not modified

print(my_str.lower())
```

```
Živjo Anže. Star si 10 let.
živjo anže. star si 10 let.
```

In [3]:

```
my_str = "Živjo Anže. Star si 10 let."
print(my_str) # not modified

print(my_str.upper())
```

```
Živjo Anže. Star si 10 let.
ŽIVJO ANŽE. STAR SI 10 LET.
```

Preverimo lahko ali se naš string začne oziroma konča s poljubnim sub-string-om.

In [4]:

```
my_str = "Živjo Anže. Star si 10 let."
print(my_str) # not modified

print(my_str.startswith("Živjo"))
print(my_str.startswith("Zdravo"))
print(my_str.startswith("Živ"))
```

```
Živjo Anže. Star si 10 let.
True
False
True
```

In [5]:

```
my_str = "Živjo Anže. Star si 10 let."
print(my_str) # not modified

print(my_str.endswith("Živjo"))
print(my_str.endswith("let"))
print(my_str.endswith("let."))
```

```
Živjo Anže. Star si 10 let.
False
False
True
```

Iz začetka oziroma konca našega stringa lahko odstranimo znake.

In [6]:

```
my_string = "Živjo Anže. Star si 10 let."
print(my_str)

print(my_str.strip("."))
print(my_str.strip("Živjo"))
```

```
Živjo Anže. Star si 10 let.
Živjo Anže. Star si 10 let
 Anže. Star si 10 let.
```

Znake v stringu lahko nadomestimo s pooljubnimi znaki.

In [7]:

```
my_string = "Živjo Anže. Star si 10 let."
print(my_string)

print(my_string.replace(" ", "-"))
print(my_string.replace("Živjo", "Zdravo"))
```

```
Živjo Anže. Star si 10 let.
Živjo-Anže.-Star-si-10-let.
Zdravo Anže. Star si 10 let.
```

Različne stringe lahko med seboj združujemo (concate)

In [8]:

```
str1 = "Živjo"
str2 = "Anže"

print(str1 + str2)
print(str1 + " " + str2)
```

```
ŽivjoAnže
Živjo Anže
```

In [ ]:

```
```

Matematične operacije

- + seštevanje
- - odštevanje
- * množenje
- / deljenje
- // celoštevilsko deljenje
- ** eksponent
- % ostanek pri deljenju

In [24]:

```
x = 9
y = 4
```

In [25]:

```
x + y
```

Out[25]:

13

In [26]:

```
# še drugačen način seštevanja
# x += y
# x
```

In [27]:

```
x - y
```

Out[27]:

5

In [28]:

```
x * y
```

Out[28]:

36

In [29]:

```
x / y
```

Out[29]:

2.25

In [30]:

```
a = 6
b = 3
a / b # Pri navadnem deljenju je rezultat vedno float. Tud če je delenje brez ostan
```

Out[30]:

```
2.0
```

In [31]:

```
x // y # celoštevilsko deljenje... 9 / 4 = 2*4 + ostanek (ta dvojka se izpiše)
```

Out[31]:

```
2
```

In [32]:

```
x % y # ostanek pri deljenju
```

Out[32]:

```
1
```

In [33]:

```
x ** y # na potenco
```

Out[33]:

```
6561
```

Potek operacij

> Operational precedence

In [75]:

```
x = 20 + 4 * 10
x # kaj se bo izpisal? 60 al 240
```

Out[75]:

```
60
```

Vsaka operacija ima določeno pomembnost.

V izrazu se prvo izvedejo operacije z najvišjo pomembnostjo. Ko pridobimo te rezultate, se nato izvedejo naslednje najpomembnejše operacije in tako do konca.

V primeru operacij z enako pomembnostjo se le te izvajajo od leve-proti-desni.

Tabela (od najpomembnejše do najmanj)

Operacije	Opis

Operacije	Opis
**	exponentiation
+x, -x, ~x	unary positive, unary negation, bitwise negation
*, /, //, %	multiplication, division, floor division, modulo
+, -	addition, subtraction
<<, >>	bit shifts
&	bitwise AND
^	bitwise XOR
|	bitwise OR
==, !=, <, <=, >, >=, is, is not	comparisons, identity
not	Boolean NOT
and	Boolean AND
or	Boolean OR

Potek operacij se lahko spremeni z uporabo oklepajev ().

Izrazi v okepajih se izvedejo pred izrazi, ki niso v oklepajih.

Nič ni narobe s pretirano uporabo oklepajev tudi, če niso potrebni. Uporaba oklepajev velja za dobro prakso, saj izboljša berljivost kode.

In [76]:

```
x = 20 + (4 * 10) # prvo se izvede oklepaj in dobimo 20 + 40 = 60
y = (20 + 4) * 10 # prvo se izvede oklepaj in dobimo 24 * 10 = 240
print(x)
print(y)
```

```
60
240
```

Primerjalne operacije

- < manjši
- > večji
- <= manjše ali enako
- >= večje ali enako
- == enako
- != neenako

In [77]:

```
5 < 10
```

Out[77]:

```
True
```

In [78]:

```
10 > 5
```

Out[78]:

```
True
```

In [79]:

```
3 <= 2
```

Out[79]:

```
False
```

In [80]:

```
5 >= 5
```

Out[80]:

```
True
```

Ko primerjamo dve spremenljivki z uporabo == , primerjamo njuni vrednosti.

In [81]:

```
5 == 4
```

Out[81]:

```
False
```

In [82]:

```
# treba pazit pri primerjanju float vrednosti, ker na prvi decimalki je stvar še en
x = 1.1000 + 2.2000
y = 3.3000
print(x == y)
print(f' x: {x:.50} \n y: {y:.50}')
```

```
False
 x: 3.30000000000000026645352591003756970167160034179690
 y: 3.29999999999999982236431605997495353221893310546875
```

In [83]:

```
4 != 4
```

Out[83]:

```
False
```

Primer večih primerjav v eni vrstici:

In [86]:

```
1 < 4 > 6 < 10
# same as (1 < 4) and (4 > 6) and (6 < 10)
```

Out[86]:

```
False
```

Logične operacije

- not
- or
- and
- is > Primerja identiteto
- in > Preverja, če je vrednost znotraj primerjalne vrednosti

NOT

A	NOT
False	True
True	False

In [87]:

```
x = False
not x # obrne vrednost. Če je vrednost True jo obrne v False, če je False jo obrne
```

Out[87]:

```
True
```

OR

A	B	OR
False	False	False
False	True	True
True	False	True
True	True	True

In [88]:

```
x = True
y = False
x or y # če je ena izmed vrednosti True, bo izraz True
```

Out[88]:

```
True
```

AND

A	B	AND
False	False	False
False	True	False
True	False	Fasle
True	True	True

In [89]:

```
x = True
y = False
x and y # če je ena izmed vrednosti False, bo izraz False
```

Out[89]:

```
False
```

IS

In [40]:

```
a = [1,2,3]
b = [1,2,3]
c = a

print("a == b")
print(a == b)
print()
print("a is b")
print(a is b)

print(30*"*")

print("a == c")
print(a == c)
print()
print("a is c")
print(a is c)

print(30*"-")
print("a id: ", id(a))
print("b id: ", id(b))
print("c id: ", id(c)) # c in a imasta isto identiteto. To tud pomen, da če spremen

print()
```

```
a == b
True

a is b
False
******************************
a == c
True

a is c
True
------------------------------
a id:  2684900183488
b id:  2684900126528
c id:  2684900183488
```

IN

In [93]:

```
x = "b"
x in "abc" # primerja ali je x v stringu, listu, itd..
```

Out[93]:

```
True
```

Vaja 01

Naloga: Uporabnika vprašajte naj vnese svojo starost v letih.

Vrednost pretvorite v mesece in to izpišite.

In [94]:

```
age = int(input("Vnesi koliko let si sat: "))

months = age*12
print(f"Star si {months} mesecev.")
```

```
Vnesi koliko let si sat: 12
Star si 144 mesecev.
```

Vaja 02

Naloga: Uporabnika vprašajte za dve decimalni vrednosti.

Preverite, če je prva vrednost večja od druge.

In [99]:

```
x = float(input("first: "))
y = float(input("second: "))

print(type(x))
print(x)
print()

print(type(y))
print(y)
x >= y
```

```
first: 12.4
second: 12.6
<class 'float'>
12.4

<class 'float'>
12.6
```

Out[99]:

```
False
```

Vaje 03

Naloga: Uporabnika vprašajte za 3 celoštevilske vrednosti in jih izpišite s pomočjo print() in type().

V eni vrstici preverite ali je druga vrednost enaka prvi in ali je tretja vrednost manjša ali enaka prvi.

In [105]:

```
a = int(input("1: "))
b = int(input("2: "))
c = int(input("3: "))

print(f"Tip: {type(a)}, Vrednost: {a}")
print(f"Tip: {type(b)}, Vrednost: {b}")
print(f"Tip: {type(c)}, Vrednost: {c}")

print((b == a) and (c <= a))
```

```
1: 1
2: 1
3: 2
Tip: <class 'int'>, Vrednost: 1
Tip: <class 'int'>, Vrednost: 1
Tip: <class 'int'>, Vrednost: 2
False
```

List

List je zbirka elementov. (V drugih programskih jezikih je znan kot "array")

V Pythonu je list definiran z oglatimi oklepaji [], elementi v listu pa so ločeni z vejico ,

In [9]:

```
živali  = ["pingvin", "medved", "los", "volk"]
print(živali)
```

```
['pingvin', 'medved', 'los', 'volk']
```

Glavne karakteristike list-ov so:

- Lists are ordered
- Lists can contain any arbitrary objects.
- List elements can be accessed by index.
- Lists can be nested to arbitrary depth.
- Lists are mutable.
- Lists are dynamic.

Lists are ordered

To pomeni, da so podatki shranjenji v list v določenem zaporedju in ostanejo v tem zaporedju.

In [10]:

```
a  = ["pingvin", "medved", "los", "volk"]
b  = ["los", "medved", "pingvin", "volk"]
a == b # čeprov mata list a in v enake elemente, niso v istem zaporedju zato nista
```

Out[10]:

```
False
```

Lists Can Contain Arbitrary Objects

Za podatke v list-u ni potrebno, da so istega tipa (data type).

In [11]:

```
a = [21.42, "medved", 3, 4, "volk", False, 3.14159]
a
```

Out[11]:

```
[21.42, 'medved', 3, 4, 'volk', False, 3.14159]
```

Podatki v list-u se lahko podvajajo.

In [12]:

```
a  = ["pingvin", "medved", "los", "volk", "medved"]
a
```

Out[12]:

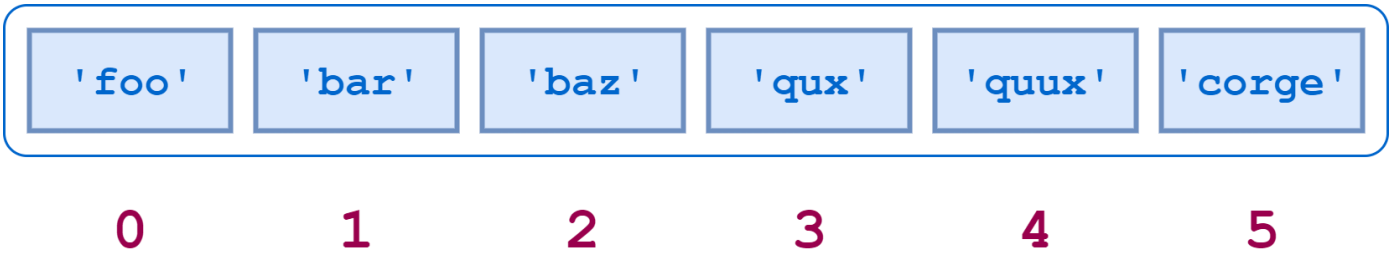
```
['pingvin', 'medved', 'los', 'volk', 'medved']
```

List Elements Can Be Accessed by Index

In [41]:

```
a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

Do elementov v list-u lahko dostopamo, če vemo njegov index (na kateri poziciji je).



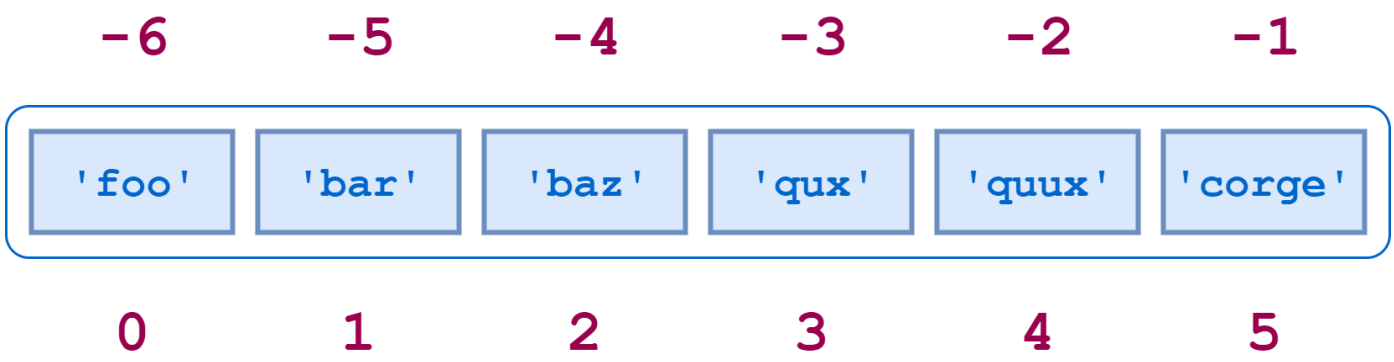
V Pythonu se indexiranje začne z 0.

In [42]:

```
print(a[0])
print(a[2])
print(a[3])
```

```
foo
baz
qux
```

Indexiramo lahko tudi z negativnimi vrednostmi:



In [43]:

```
print(a[-6])
print(a[-1])
```

```
foo
corge
```

Slicing

To nam pomaga pridobiti določene pod-liste iz že narejene list-e.

In [44]:

```
print(a[2:5])
# a[m:n] nam vrne list vrednosti, ki se nahajajo v a od vključno indexa m do izvzet
# a[2:5] nam vrne elemente v listu a od vključno 2 do ne vključno 5
```

```
['baz', 'qux', 'quux']
```

In [45]:

```
print(a[-5:-2]) # isto deluje z negativnimi indexi
```

```
['bar', 'baz', 'qux']
```

In [46]:

```
print(a[:4]) # če izvzamemo začetni index nam začne pri indexu 0
```

```
['foo', 'bar', 'baz', 'qux']
```

In [47]:

```
print(a[2:]) # če izvzamemo zadnji index se sprehodi do konca seznama
```

```
['baz', 'qux', 'quux', 'corge']
```

Specificeramo lahko tudi korak, za koliko naj se premakne.

In [48]:

```
print(a[::2]) # začne pri indexu 0, do konca, vsako drugo vrednost
```

```
['foo', 'baz', 'quux']
```

In [49]:

```
print(a[1:5:2])
print(a[6:0:-2]) # korak je lahko tudi negativen
print(a[::-1]) # sintaksa za sprehajanje po listu v obratnem vrstnem redu
```

```
['bar', 'qux']
['corge', 'qux', 'bar']
['corge', 'quux', 'qux', 'baz', 'bar', 'foo']
```

Lists can be nested to arbitrary depth

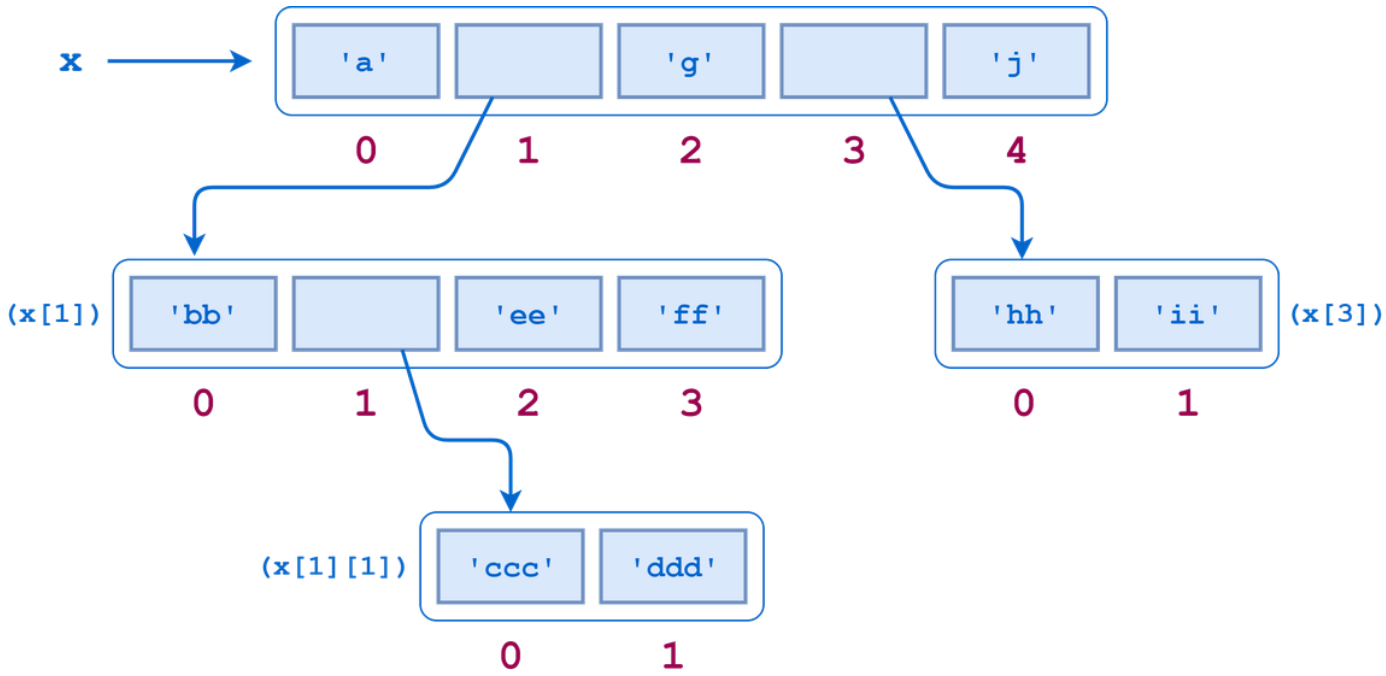
Elementi v listu so lahko poljubnega data type.

Lahko je tudi še en list. Tako lahko dodajamo dimenzije našemu list-u

In [120]:

```
x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
print(x)
```

```
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
```



In [121]:

```
print(x[2]) # element na indexu 2 je preprosti string dolžine 1 črke
```

```
g
```

In [122]:

```
print(x[1]) # 1 element je nov list z 4 elementi
```

```
['bb', ['ccc', 'ddd'], 'ee', 'ff']
```

In [123]:

```
print(x[1][0]) # da pridemo do njihovih elementov preprosto dodamo nov []
```

```
bb
```

In [124]:

```
print(x[1][1])
print(x[1][1][0])
```

```
['ccc', 'ddd']
ccc
```

In [ ]:

In [ ]: