

Programiranje

Naloga: 5x zapored izpiši neko številko

Za izpis nečesa se uporablja beseda

```
print()
```

In [1]:

```
print(10)
```

10

Če je na primer 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

Tekom programiranja se odločimo, 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 računalnik povedal, naj si shrani številko katero hočemo izpisat. Potem pa jo računalnik izpiše 100x.

To lahko dosežemo s pomočjo spremenljivk.

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 želimo izpisati
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 \(https://www.python.org/dev/peps/pep-0008/?\)](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 []:

In []:

```
x = 1
y = 2
rezultat = x + y
print("Sešteli smo ", x, "in ", y)
print("Rezultat je: ", rezultat)
```

In []:

In []:

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 maksimalne 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 = 1267650600228229401496703205376 # Tole je 2^100 , v Javi je omejitev men se zdi
print(x)
print(type(x))

1267650600228229401496703205376
<class 'int'>
```

Floating-point (decimalno število) - float

Float predstavlja decimalno število (število s plavajočo vecijo).

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

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(spremenljivka)
```

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 dvojnimi (") 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 dvojnimi 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".
- **** primarni 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 boolean 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

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_\(https://docs.python.org/3.6/library/string.html#formatspec\)](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 poljubnimi 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 (concat)

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 # 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
**	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 oklepajih 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 ena
x = 1.1000 + 2.2000
y = 3.3000
print(x == y)
print(f' x: {x:.50} \n y: {y:.50}')
```

False

x: 3.30000000000000002664535259100375697016716003417969

y: 3.29999999999999998223643160599749535322189331054688

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	False
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]:

```
# Kle ne morš met primera: x=1, y=1 x is y    -> ker pride True, sam to je zarad pos

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 [92]:

```
# poseben primer so številke od -5 do 256
a = -7
b = -7

for _ in range(-7, 260):
    print(f'Vrednost a: {a}, identiteta: {id(a)}')
    print(f'Vrednost b: {b}, identiteta: {id(b)}')

    if a == b:
        print("a and b have the same value.")

    if a is b:
        print("a and b are the same.")

    print(30*"")
    a += 1
    b += 1
```

```
Vrednost a: -7, identiteta: 2027528172656
Vrednost b: -7, identiteta: 2027528172144
a and b have the same value.
*****
Vrednost a: -6, identiteta: 2027528171696
Vrednost b: -6, identiteta: 2027528172656
a and b have the same value.
*****
Vrednost a: -5, identiteta: 140707435483776
Vrednost b: -5, identiteta: 140707435483776
a and b have the same value.
a and b are the same.
*****
Vrednost a: -4, identiteta: 140707435483808
Vrednost b: -4, identiteta: 140707435483808
a and b have the same value.
a and b are the same.
*****
Vrednost a: -3, identiteta: 140707435483840
Vrednost b: -3, identiteta: 140707435483840
```

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 ali enaka 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 celoštevilске 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 shranjeni v list v določenem zaporedju in ostanejo v tem zaporedju.

In [10]:

```
a = ["pingvin", "medved", "los", "volk"]  
b = ["los", "medved", "pingvin", "volk"]  
a == b # čeprav 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]:

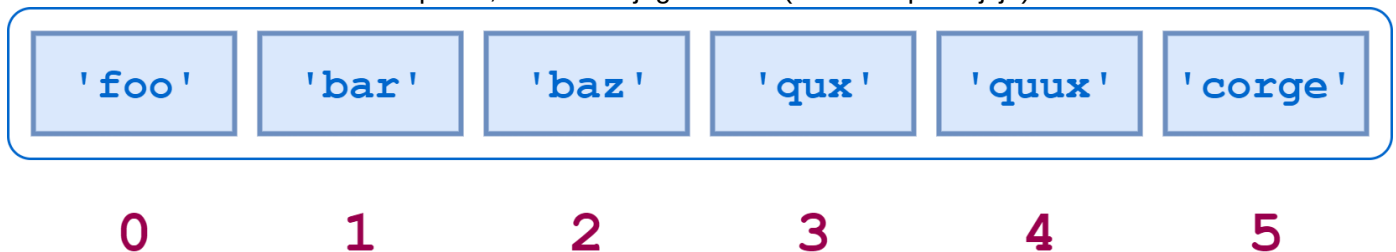
['pingvin', 'medved', 'los', 'volk', 'medved']

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:

-6 -5 -4 -3 -2 -1



0 1 2 3 4 5

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 izvezt  
# 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 izvezamemo zadnji index se sprehodi do konca seznama  
['baz', 'qux', 'quux', 'corge']
```

Specificiramo 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']
```

Use the * operator to represent the “rest” of a list

Often times, especially when dealing with the arguments to functions, it's useful to extract a few elements at the beginning (or end) of a list while keeping the “rest” for use later. Python 2 has no easy way to accomplish this aside from using slices as shown below. Python 3 allows you to use the * operator on the left hand side of an assignment to represent the rest of a sequence.

In [56]:

```
some_list = ['a', 'b', 'c', 'd', 'e']  
(first, second, *rest) = some_list  
print(rest)  
(first, *middle, last) = some_list  
print(middle)  
(*head, second_last, last) = some_list  
print(head)  
['c', 'd', 'e']  
['b', 'c', 'd']  
['a', 'b', 'c']
```

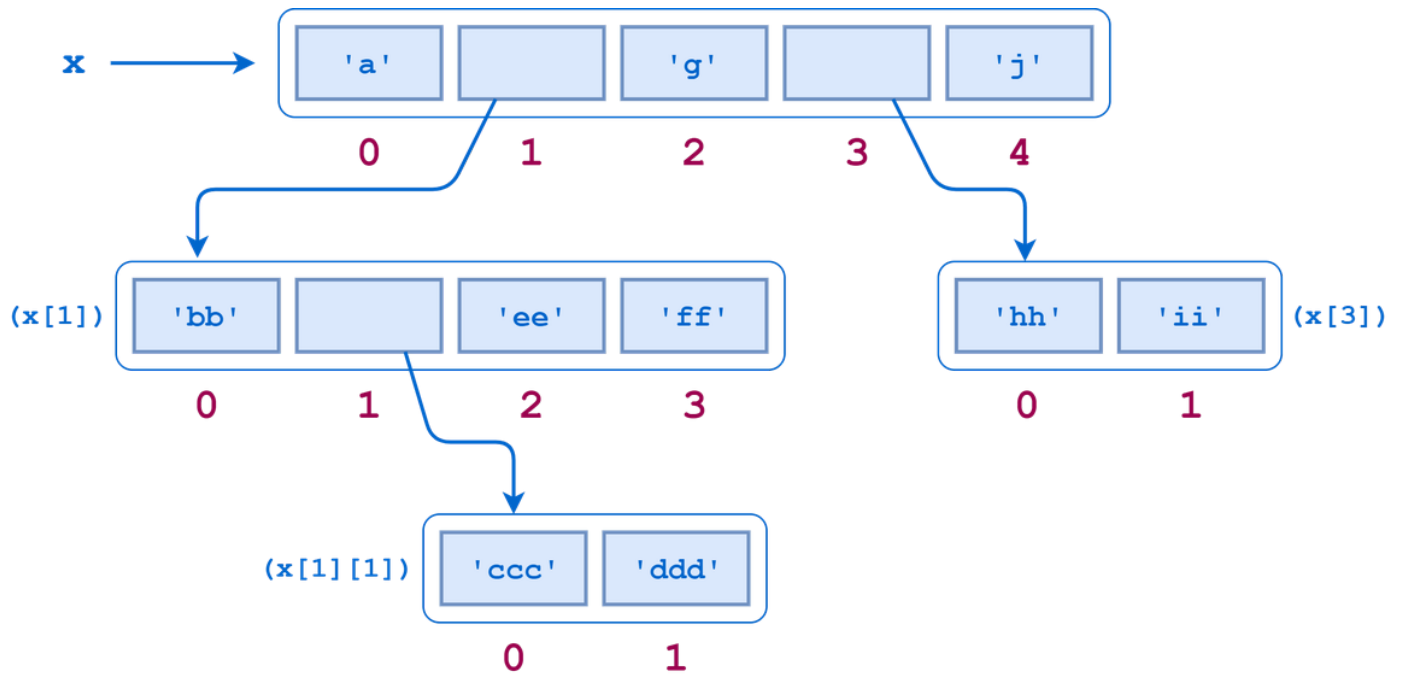
Lists can be nested to arbitrary depth

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
```

Lists Are Mutable

To pomeni, da jih lahko spreminjamo. Lahko dodajamo elemente, jih brišemo, premikamo vrstni red, itd..

Most of the data types you have encountered so far have been atomic types. Integer or float objects, for example, are primitive units that can't be further broken down. These types are immutable, meaning that they can't be changed once they have been assigned. It doesn't make much sense to think of changing the value of an integer. If you want a different integer, you just assign a different one.

By contrast, the string type is a composite type. Strings are reducible to smaller parts—the component characters. It might make sense to think of changing the characters in a string. But you can't. In Python, strings are also immutable.

Spreminjanje vrednosti elementa.

In [125]:

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

a[2] = "koza"
print(a)
```

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

Brisanje elementa.

In [126]:

```
a = ["pingvin", "medved", "los", "volk"]
del a[3]
print(a)
```

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

Spreminjanje večih elementov naenkrat.

Velikost dodanih elementov ni potrebno, da je ista kot velikost zamenjanih elementov. Python bo povečal oziroma zmanjšal list po potrebi.

In [127]:

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

a = ["pingvin", "medved", "los", "volk"]
a[1:3] = [1.1, 2.2, 3.3, 4.4, 5.5]
print(a)

a = ["pingvin", "medved", "los", "volk"]
a[1:4] = ['krava']
print(a)

a = ["pingvin", "medved", "los", "volk"]
a[1:3] = [] # slicane elemente zamenjamo z praznim listom -> jih izbrišemo
print(a)
```

```
['pingvin', 'medved', 'los', 'volk']
['pingvin', 1.1, 2.2, 3.3, 4.4, 5.5, 'volk']
['pingvin', 'krava']
['pingvin', 'volk']
```

Dodajanje elementov.

Lahko dodajamo vrednosti s pomočjo `.append()` funkcije

In [128]:

```
a = ["pingvin", "medved", "los", "volk"]
a.append(123)
print(a)
```

```
['pingvin', 'medved', 'los', 'volk', 123]
```

`.append()` doda celotno vrednost na konec lista.

In [129]:

```
a = ["pingvin", "medved", "los", "volk"]
a.append([1, 2, 3])
print(a)
```

```
['pingvin', 'medved', 'los', 'volk', [1, 2, 3]]
```

Če želimo dodati vsako vrednost posebej lahko uporabimo `.extend()`

In [130]:

```
a = ["pingvin", "medved", "los", "volk"]
a.extend([1, 2, 3])
print(a)
```

```
['pingvin', 'medved', 'los', 'volk', 1, 2, 3]
```

Dodajanje elementa na specifično mesto

```
a.insert(<index>, <obj>)
```

Element na mestu `index` zamenjamo z object.

In [131]:

```
a = ["pingvin", "medved", "los", "volk"]
a.insert(3, 3.14159)
print(a)
```

```
['pingvin', 'medved', 'los', 3.14159, 'volk']
```

```
a.remove(<obj>)
```

Odstranimo object iz liste.

In [132]:

```
a = ["pingvin", "medved", "los", "volk"]
a.remove("los")
print(a)
```

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

```
a.pop(index=-1)
```

Odstranimo element z indexa. Metoda nam vrne izbrisani element. Default pop je zadnji element.

In [133]:

```
a = ["pingvin", "medved", "los", "volk"]
default_pop = a.pop()
naslednji_pop = a.pop(1)

print(a)
print(default_pop)
print(naslednji_pop)
```

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

Lists Are Dynamic

Dynamic pove, da ni treba na začetku definirat, da bo to list.

In [134]:

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

a = 1
print(a)
print(type(a))
```

```
['pingvin', 'medved', 'los', 'volk']
<class 'list'>
1
<class 'int'>
```

In []:

In []:

Vaja 01

Naloga: Iz sledečega list-a pridobite vrednost ****ffff****

```
our_list = ["a", ["bb", "cc"], "d", [{"eee"}, {"ffff"}, "ggg"]]
```

In [192]:

```
our_list = ["a", ["bb", "cc"], "d", [{"eee"}, {"ffff"}, "ggg"]]  
  
print(our_list)  
print(our_list[3])  
print(our_list[3][1])  
print(our_list[3][1][0])
```

```
['a', ['bb', 'cc'], 'd', [{'eee'}, {'ffff'}, 'ggg']]  
[{'eee'}, {'ffff'}, 'ggg']  
{'ffff'}  
ffff
```

Vaja 02

Naloga: Pri sledečem list-u začnite z vrednostjo 4 in vzemite vsako 3 vrednost.

```
our_list = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

Rešitev:

```
[4, 7, 10, 13, 16, 19]
```

In [13]:

```
our_list = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]  
  
our_sublist = (our_list[3::3])  
print(our_sublist)
```

```
[4, 7, 10, 13, 16, 19]
```

In []:

In []:

Tuples

Imajo enake lastnosti kot list, vendar so "immutable" - Njihovih vrednosti ne moremo spreminjati.

Definira se jih z navadnimi oklepaji ()

In [135]:

```
t = ("pingvin", "medved", "los", "volk")
print(t)

('pingvin', 'medved', 'los', 'volk')
```

In [136]:

```
# Primer: Tuples are ordered
t = ("pingvin", "medved", "los", "volk")
t2 = ("pingvin", "volk", "medved", "los")

if t == t2:
    print("Tuples are NOT ordered")
else:
    print("Tuples ARE ordered")
```

Tuples ARE ordered

In [137]:

```
# Primer: Tuples can contain any arbitrary object
t = ("pingvin", "medved", "los", "volk", 1.23, True)
print(t)

('pingvin', 'medved', 'los', 'volk', 1.23, True)
```

In [138]:

```
# Primer: Tuples are indexed
t = ("pingvin", "medved", "los", "volk")
print(t)

print(t[2]) # primer, da so elementi indexirani

print(t[1:3]) # slicing primer

('pingvin', 'medved', 'los', 'volk')
los
('medved', 'los')
```

In [139]:

```
# Primer: Tuples can be nested
t = ("pingvin", "medved", "los", "volk", ("lisica", "krava"))
print(t)

('pingvin', 'medved', 'los', 'volk', ('lisica', 'krava'))
```


In [140]:

```
# Primer: Touples are IMMUTABLE
t = ("pingvin", "medved", "los", "volk")
t[1] = "Bork!"
```

```
-----
-----
TypeError                                Traceback (most recent call
last)
<ipython-input-140-1f7dd710d595> in <module>
      1 # Primer: Touples are IMMUTABLE
      2 t = ("pingvin", "medved", "los", "volk")
----> 3 t[1] = "Bork!"
```

TypeError: 'tuple' object does not support item assignment

In [141]:

```
# Primer: da je dinamična
t = ("pingvin", "medved", "los", "volk")
print(t)
print(type(t))

t = 2
print(t)
print(type(t))
```

```
('pingvin', 'medved', 'los', 'volk')
<class 'tuple'>
2
<class 'int'>
```

Zakaj bi uporabljali tuple namesto list?

- Program je hitrejši, če manipulira z tuple kot pa z list
- Če ne želimo spreminjati elementov

TECHNICAL

Treba pazit kadar inicializiramo tuple samo z eno vrednostjo.

In [142]:

```

t = (1,2,3,4) # nebi smel bit problem
print(t)
print(type(t))
print()

t = () # nebi smel bit problem. Prazen tuple
print(t)
print(type(t))
print()

t = (2) # kle nastane problem
print(t)
print(type(t))
print()
'''

```

Since parentheses are also used to define operator precedence in expressions, Python the expression (2) as simply the integer 2 and creates an int object. To tell Python to define a singleton tuple, include a trailing comma (,) just before the closing parenthesis.

```

t = (2,)
print(t)
print(type(t))

```

```

(1, 2, 3, 4)
<class 'tuple'>

```

```

()
<class 'tuple'>

```

```

2
<class 'int'>

```

```

(2,)
<class 'tuple'>

```

Dictionaries

Njihove lastnosti so sledeče:

- Are insertion ordered (vrstni red elementov je odvisen od vrstega reda dodajanja) (to velja od python 3.6+)
- Element accession (do elementov se dostopa preko ključev, ne preko indexov)
- Can be nested (kot element ima lahko še en dictionary, list, tuple,)
- Are mutable (vrednosti elementov se lahko spreminjajo)
- Are dynamic (sej to velja za vse pr pythonu)

Dictionary je sestavljen iz parov ključa in vrednosti. Vsak Ključ ima svojo vrednost.

In [96]:

```
d = {  
    'macek' : 'Silvestre',  
    'pes'   : 'Fido',  
    'papagaj': 'Kakadu'  
}  
print(d)  
print(type(d))
```

```
{'macek': 'Silvestre', 'pes': 'Fido', 'papagaj': 'Kakadu'}  
<class 'dict'>
```

In [97]:

```
# Primer: Can contain any arbitrary objects  
d = {  
    'macek' : 1,  
    'pes'   : 'Fido',  
    'papagaj': False  
}  
print(d)
```

```
{'macek': 1, 'pes': 'Fido', 'papagaj': False}
```

Accessing dictionary value

Vrednosti najdemo preko ključev.

In [145]:

```
d = {  
    'macek' : 'Silvestre',  
    'pes'   : 'Fido',  
    'papagaj': 'Kakadu'  
}  
print(d['papagaj'])
```

Kakadu

Če vpišemo ključ, ki ne obstaja python vrne napako.

In [146]:

```
d = {
    'macek' : 'Silvestre',
    'pes'    : 'Fido',
    'papagaj': 'Kakadu'
}
d['koza'] # should give KeyError
```

```
-----
-----
KeyError                                Traceback (most recent call
last)
<ipython-input-146-91bc34ee01c4> in <module>
      4 'papagaj': 'Kakadu'
      5 }
----> 6 d['koza'] # should give KeyError
```

KeyError: 'koza'

Dodajanje novih vrednosti

In [149]:

```
d = {
    'macek' : 'Silvestre',
    'pes'    : 'Fido',
    'papagaj': 'Kakadu'
}
d['koza'] = "Micka"
print(d)
```

```
{'macek': 'Silvestre', 'pes': 'Fido', 'papagaj': 'Kakadu', 'koza': 'Mi
cka'}
```

Posodabljanje vrednosti.

In [150]:

```
d['koza'] = 'Helga'
print(d)
```

```
{'macek': 'Silvestre', 'pes': 'Fido', 'papagaj': 'Kakadu', 'koza': 'He
lga'}
```

Brisanje elementa.

In [151]:

```
del d['koza']
print(d)
```

```
{'macek': 'Silvestre', 'pes': 'Fido', 'papagaj': 'Kakadu'}
```

Restrictions on dictionary keys

Kot Ključ lahko uporabimo poljubne vrednosti, dokler so "immutable". Sm spadajo integer, float, string, boolean, tuple.

Tuple je lahko ključ le, če so elementi znotraj njega tudi "immutable" (strings, integers, floats,...).

In [100]:

```
d = {1: 'a',
      2.3: 'b',
      "string": 'c',
      None: 'd',
      (1,"tuple"): 'e',
      }
print(d)
print(d[2.3])
print(d[None])
print(d[(1, "tuple")])
# PAZI: Če daš True namest None bo narobe deloval. Pomojm tretira 1 kt True pa se m
# Sej keywords dajat sm je nesmiselno
```

```
{1: 'a', 2.3: 'b', 'string': 'c', None: 'd', (1, 'tuple'): 'e'}
b
d
e
```

In [103]:

```
# Primer: Vrže error, ker hočemo kot ključ uporabiti list, ki pa je mutable
d = {[1,1]: 'a', [1,2]: 'b'}
d = {(1,2,[1,2]): "f",}
```

```
-----
-----
TypeError                                Traceback (most recent call
last)
<ipython-input-103-f98311767e81> in <module>
      1 # Primer: Vrže error, ker hočemo kot ključ uporabiti list, ki
      pa je mutable
----> 2 d = {[1,1]: 'a', [1,2]: 'b'}
      3 d = {(1,2,[1,2]): "f",}
```

TypeError: unhashable type: 'list'

Technical Note:

Why does the error message say "unhashable" rather than "mutable"? Python uses hash values internally to implement dictionary keys, so an object must be hashable to be used as a key.

<https://docs.python.org/3/glossary.html#term-hashable> (<https://docs.python.org/3/glossary.html#term-hashable>).

An object is hashable if it has a hash value which never changes during its lifetime (it needs a **hash()** method), and can be compared to other objects (it needs an **eq()** method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their id().

AMPAK

Ključ more bit edinstven (se ne sme ponoviti):

In [104]:

```
d = {  
'macek' : 'Silvestre',  
'pes' : 'Fido',  
'papagaj' : 'Kakadu',  
'macek' : 'Amadeus'  
}  
print(d)
```

```
{'macek': 'Amadeus', 'pes': 'Fido', 'papagaj': 'Kakadu'}
```

Built-in Dictionary Methods

Še nekaj ostalih metod.

A rabm to vse kazat? Pomojm se velk lah preskoč, če ni časa. Če ne pa na hitr 1 primer pokaže.

```
d.clear()
```

d.clear() empties dictionary d of all key-value pairs:

In [155]:

```
d = {'a': 10, 'b': 20, 'c': 30}  
print(d)  
  
d.clear()  
print(d)
```

```
{'a': 10, 'b': 20, 'c': 30}  
{}
```

```
d.get(<key>[, <default>])
```

get() metoda nam nudi preprost način kako dobimo vrednost ključa brez, da preverimo, če ključ sploh obstaja.

Če ključ ne obstaja dobimo None

In [156]:

```
d = {'a': 10, 'b': 20, 'c': 30}
print(d.get('b'))
print(d.get('z'))
```

20

None

Če ključ ni najden in smo specificirali dodaten argument nam vrne le tega namesto None.

In [157]:

```
d = {'a': 10, 'b': 20, 'c': 30}
print(d.get('z', -5))
```

-5

d.items()

Vrne nam list sestavljen iz touple, ki so sestavljeni iz ključ-vrednost parov. Prvi element toupla je ključ, drugi je vrednost.

In [158]:

```
d = {'a': 10, 'b': 20, 'c': 30}
print(list(d.items()))
print(list(d.items())[1])
print(list(d.items())[1][1])
```

[('a', 10), ('b', 20), ('c', 30)]

('b', 20)

20

d.keys()

Vrne nam list ključev.

In [159]:

```
d = {'a': 10, 'b': 20, 'c': 30}
print(list(d.keys()))
```

['a', 'b', 'c']

d.values()

Vrne nam list vrednosti.

In [160]:

```
d = {'a': 10, 'b': 10, 'c': 10}
print(list(d.values()))

[10, 10, 10]
```

```
d.pop(<key>[, <default>])
```

Če ključ obstaja v dictionary ga odstrani skupaj z njegovo vrednostjo.

In [161]:

```
d = {'a': 10, 'b': 20, 'c': 30}
print(d.pop('b'))
print(d)

20
{'a': 10, 'c': 30}
```

Če ne najde ključa nam vrne napako.

In [162]:

```
d = {'a': 10, 'b': 20, 'c': 30}
print(d.pop('z'))
```

```
-----
-----
KeyError                                Traceback (most recent call
last)
<ipython-input-162-f36d736a326b> in <module>
      1 d = {'a': 10, 'b': 20, 'c': 30}
----> 2 print(d.pop('z'))

KeyError: 'z'
```

Če ključ ni najden, smo pa dodatno specificirali default argument, potem nam vrne vrednost default argumenta in ne dvigne nobene napake.

In [163]:

```
d = {'a': 10, 'b': 20, 'c': 30}
print(d.pop('z', "Ni našlo ključa"))
```

Ni našlo ključa

```
d.popitem()
```

Odstrani random, arbitrarni ključ-vrednost par in nam ga vrne kot tuple.

popitem() is useful to destructively iterate over a dictionary, as often used in set algorithms

In [164]:

```
d = {'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50, 'f': 60, 'g': 70}
print(d.popitem())
print(d)
```

```
('g', 70)
{'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50, 'f': 60}
```

Če je dictionary prazen dobimo KeyError error.

In [165]:

```
d = {}
d.popitem()
```

```
-----
-----
KeyError                                Traceback (most recent call
  last)
<ipython-input-165-3d5a99fd0340> in <module>
      1 d = {}
----> 2 d.popitem()
```

KeyError: 'popitem(): dictionary is empty'

In [166]:

```
# Primer: Da je dictionary dinamičen
d = {'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50, 'f': 60, 'g': 70}
print(d)
print(type(d))
```

```
d = 1.2
print(d)
print(type(d))
```

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50, 'f': 60, 'g': 70}
<class 'dict'>
1.2
<class 'float'>
```

In []:

In []:

```
dict_ = {
    "a": 1,
    "b": 2,
    "c": 3,
    "d": 4
}
```

Vaja 03

Naloga: Sledečemu dictionary zamenjajte vrednost pod ključem **b** v vrednost 12 in odstranite vrednost pod ključem **d**.

```
our_dict = {  
    "a": 10,  
    "b": 9,  
    "c": 8,  
    "d": 7,  
    "e": 3  
}
```

In [17]:

```
our_dict = {  
    "a": 10,  
    "b": 9,  
    "c": 8,  
    "d": 7,  
    "e": 3  
}  
  
our_dict["b"] = 12  
del our_dict["d"]  
  
print(our_dict)
```

```
{'a': 10, 'b': 12, 'c': 8, 'e': 3}
```

Vaja 04

Naloga: Iz sledečega dictionary pridobite vrednost **fff**.

```
d = {  
    "a": "a",  
    "b": "b",  
    "c": {  
        1: 11,  
        2: 22,  
        3: 33,  
        4: {  
            5: "ccc",  
            6: "ddd",  
            7: "fff"  
        }  
    }  
}
```

In [16]:

```
d = {
    "a": "a",
    "b": "b",
    "c": {
        1: 11,
        2: 22,
        3: 33,
        4: {
            5: "ccc",
            6: "ddd",
            "7": "fff"
        }
    }
}

print(d["c"][4]["7"])
```

fff

Sets

(Množice)

Ta data-tip je prav tako kolekcija elementov, ampak nad set-i se da izvajati posebne operacije.

Lastnosti

- Sets are unordered
- Set elements are unique. Duplicate elements are not allowed
- A set itself may be modified, but the elements must be of type immutable.
- Sets do not support indexing, slicing, or other sequence-like behavior. (Če hočš pridt do specifičnega elementa lahko uporabiš for _ in set:)

In [113]:

```
s = {"medved", "zajec", "volk", "slon", "zajec"} # zajec se ne ponovi ampak je samo
print(s)
print(type(s))
```

```
{'volk', 'slon', 'zajec', 'medved'}
<class 'set'>
```

Set je lahko prazen ampak Python bo {} prebral kot prazen dictionary.

Edini način, da ustvarimo prazen set je z set().

In [168]:

```
s = {}
print(type(s))

s = set()
print(type(s))
```

```
<class 'dict'>
<class 'set'>
```

Elementi so lahko poljubni ampak morajo biti "immutable."

In [114]:

```
s = {42, 'eee', (1, 2, 3), 3.14159} # tuple je lahko element, ker je immutable
print(s)
```

```
{3.14159, 42, 'eee', (1, 2, 3)}
```

In [115]:

```
s = {11, [1, 2, 3], 'eeee'} # list ne more bit element, ker je mutable
print(x)
```

```
-----
-----
TypeError                                Traceback (most recent call
last)
```

```
<ipython-input-115-d32f290234c5> in <module>
----> 1 s = {11, [1, 2, 3], 'eeee'} # list ne more bit element, ker je
mutable
      2 print(x)
```

```
TypeError: unhashable type: 'list'
```

In [170]:

```
# Primer: Can be nested
s = {42, 'eee', (1, 2, (4, 5, 6)), 3.14159}
print(s)
```

```
{42, 3.14159, (1, 2, (4, 5, 6)), 'eee'}
```

Operating on a Set

Nad set-i je možno izvajanje posebnih operacij.

Večina jih je lahko zapisana na dva načina: z metodo ali z operatorjem.

Union

Vsebuje elemente iz obeh set-ov.

In [171]:

```
x1 = {1, 2, 3, 4}
x2 = {4, 5, 6, 7}
print(x1 | x2) # operator
print(x1.union(x2)) # metoda
# Element 4 je samo enkrat, ker se elementi v set-ih ne ponavljajo
```

```
{1, 2, 3, 4, 5, 6, 7}
{1, 2, 3, 4, 5, 6, 7}
```

Razlika med operatorjem in metodo je, da operator zahteva, da sta obe spremenljivki set, medtem ko metoda uzame kot argument poljuben "iterable".

In [172]:

```
print(x1.union(list(x2)))
print(x1 | list(x2)) # this one should throw error
```

```
{1, 2, 3, 4, 5, 6, 7}
```

```
-----
-----
TypeError                                Traceback (most recent call
last)
<ipython-input-172-ea3a52617f77> in <module>
      1 print(x1.union(list(x2)))
----> 2 print(x1 | list(x2)) # this one should throw error
```

TypeError: unsupported operand type(s) for |: 'set' and 'list'

Ali z operatorjem ali z metodo lahko specificiramo večje število set-ov.

In [173]:

```
x1 = {1, 2, 3, 4}
x2 = {4, 5}
x3 = {5, 6}
x4 = {9, 10}

print(x1.union(x2, x3, x4))

print(x1 | x2 | x3 | x4)
```

```
{1, 2, 3, 4, 5, 6, 9, 10}
{1, 2, 3, 4, 5, 6, 9, 10}
```

Intersection

Vrne set z elementi skupni obema set-oma.

In [174]:

```
x1 = {1, 2, 3, 4}
x2 = {4, 5, 6, 7}

print(x1.intersection(x2))
print(x1 & x2)
```

```
{4}
{4}
```

Difference

Vrne set z elementi, ki so v x1 in ne v x2.

In [175]:

```
x1 = {1, 2, 3, 4}
x2 = {4, 5, 6, 7}

print(x1.difference(x2))
print(x1 - x2)
```

```
{1, 2, 3}
{1, 2, 3}
```

Symetric Difference

Vrne set z elementi, ki so ali v prvem ali v drugem set-u, vendar ne v obeh.

In [176]:

```
x1 = {1, 2, 3, 4}
x2 = {4, 5, 6, 7}

print(x1.symmetric_difference(x2))
print(x1 ^ x2)
```

```
{1, 2, 3, 5, 6, 7}
{1, 2, 3, 5, 6, 7}
```

Modifying Sets

Set-e lahko tudi spreminjamo.

`x.add(<elem>)` adds `<elem>`, which must be a single immutable `object`, to `x`:

In [177]:

```
x = {1, 2, 3, 4}
x.add("string")
print(x)
```

```
{1, 2, 3, 4, 'string'}
```

`x.remove(<elem>)` removes `<elem>` from `x`. Python raises an exception if `<elem>` is not in `x`:

In [178]:

```
x = {1, 2, 3, 4}
print(x.remove(3))
print(x)
```

```
None
{1, 2, 4}
```

In [179]:

```
x = {1, 2, 3, 4}
x.remove(6) # gives error
```

```
-----
-----
KeyError                                Traceback (most recent call
last)
<ipython-input-179-4f450e54381b> in <module>
      1 x = {1, 2, 3, 4}
----> 2 x.remove(6) # gives error
```

KeyError: 6

`x.discard(<elem>)` also removes `<elem>` from `x`. However, if `<elem>` is not in `x`, this method quietly does nothing instead of raising an exception:

In [180]:

```
x = {1, 2, 3, 4}
print(x.discard(2))
print(x)
```

```
None
{1, 3, 4}
```

In [181]:

```
x = {1, 2, 3, 4}
x.discard(6) # doesnt give error
print(x)
```

```
{1, 2, 3, 4}
```

python

`x.pop()` removes and returns an arbitrarily chosen element from `x`. If `x` is empty, `x.pop()` raises an exception:

In [182]:

```

x = {1, 2, 3, 4}
print("First pop: ", x.pop())
print(x)

print("Second pop: ", x.pop())
print(x)

print("Third pop: ", x.pop())
print(x)

print("Fourth pop: ", x.pop())
print(x)

print("Fourth pop: ", x.pop()) #this one should give error
print(x)

```

```

First pop:  1
{2, 3, 4}
Second pop: 2
{3, 4}
Third pop:  3
{4}
Fourth pop: 4
set()

```

```

-----
-----
KeyError                                Traceback (most recent call
last)
<ipython-input-182-594f9b06ae7d> in <module>
     12 print(x)
     13
--> 14 print("Fourth pop: ", x.pop()) #this one should give error
     15 print(x)

```

KeyError: 'pop from an empty set'

`x.clear()` removes **all** elements **from** x

In [183]:

```

x = {1, 2, 3, 4}
print(x)
x.clear()
print(x)

```

```

{1, 2, 3, 4}
set()

```

Frozen sets

Python ima tudi frozenset, ki se obnaša isto kot set, le da je frozenset immutable.

In [184]:

```
x = frozenset([1, 2, 3, 4])
print(x | {9, 8, 7}) # you can perform non-modifying operations on frozensets
x.add(100) # should give error, because frozenset is immutable
```

```
frozenset({1, 2, 3, 4, 7, 8, 9})
```

```
-----
-----
AttributeError                                Traceback (most recent call
last)
<ipython-input-184-769f87059d2e> in <module>
      1 x = frozenset([1, 2, 3, 4])
      2 print(x | {9, 8, 7}) # you can perform non-modifying operation
s on frozensets
----> 3 x.add(100) # should give error, because frozenset is immutable

AttributeError: 'frozenset' object has no attribute 'add'
```

Vaja 05

Naloga: Iz sledečega lista odstranite vrednost, ki se nahaja na indexu 4. Vrednost dodajte v dictionary pod ključ **d**.

Nato iz dictionary pridobite vse vrednosti. Te vrednosti shranite v nov set in novonastali set primerjajte ali je enak podanemu set-u.

```
our_list = [1,2,3,4,5,6,7]
our_dict = {
    "a": 2,
    "b": 5,
    "c": 8,
    "d": 12,
    "e": 357,
    "f": 12
}
our_set = {357, 12, 12, 8, 5, 2, 2}
```

In [207]:

```
our_list = [1,2,3,4,5,6,7]
our_dict = {
    "a": 2,
    "b": 5,
    "c": 8,
    "d": 12,
    "e": 357,
    "f": 12
}
our_set = {357, 12, 12, 8, 5, 2, 2}
```

```
x = our_list.pop(4)
print(x)
```

```
our_dict["d"] = x
print(our_dict)
print(our_dict.values())
```

```
x_set = set(our_dict.values())
x_set == our_set
```

5

```
{'a': 2, 'b': 5, 'c': 8, 'd': 5, 'e': 357, 'f': 12}
dict_values([2, 5, 8, 5, 357, 12])
```

Out[207]:

True

In []:

Vaja 06

Naloga: Izpišite vse skupne črke sledečih dveh stringov.

```
str1 = "Danes je lep dan"
str2 = "Jutri bo deževalo"
```

OUTPUT:

```
{' ', 'a', 'd', 'e', 'l'}
```

In [3]:

```
str1 = "Danes je lep dan"  
str2 = "Jutri bo deževalo"  
  
set1 = set(str1)  
set2 = set(str2)  
  
set1.intersection(set2)
```

Out[3]:

```
{' ', 'a', 'd', 'e', 'l'}
```

In []:

In []:

In []:

In []:

In []: