List

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

Uporablja se, da več različnih vrednosti ali spremenljivk shranimo znotraj ene spremenljivke. Tako lahko preko ene spremenljivke dostopamo do več različnih stringov, števil, itd...

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 enaka
```

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]:

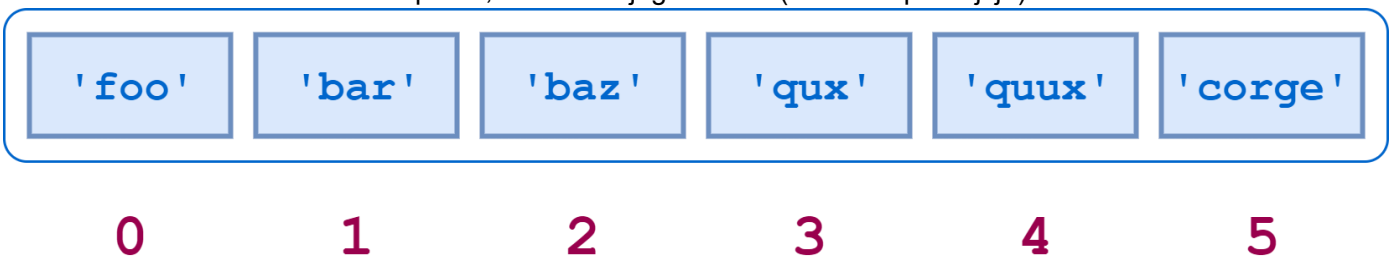
```
['pingvin', 'medved', 'los', 'volk', 'medved']
```

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).



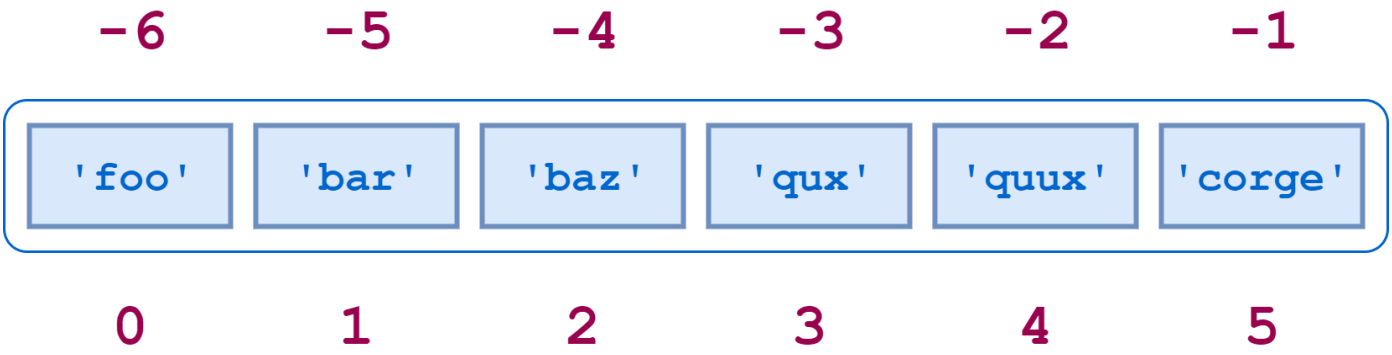
V Pythonu se indexiranje začne z 0.

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 izvzeto indexa
# 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']
```

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

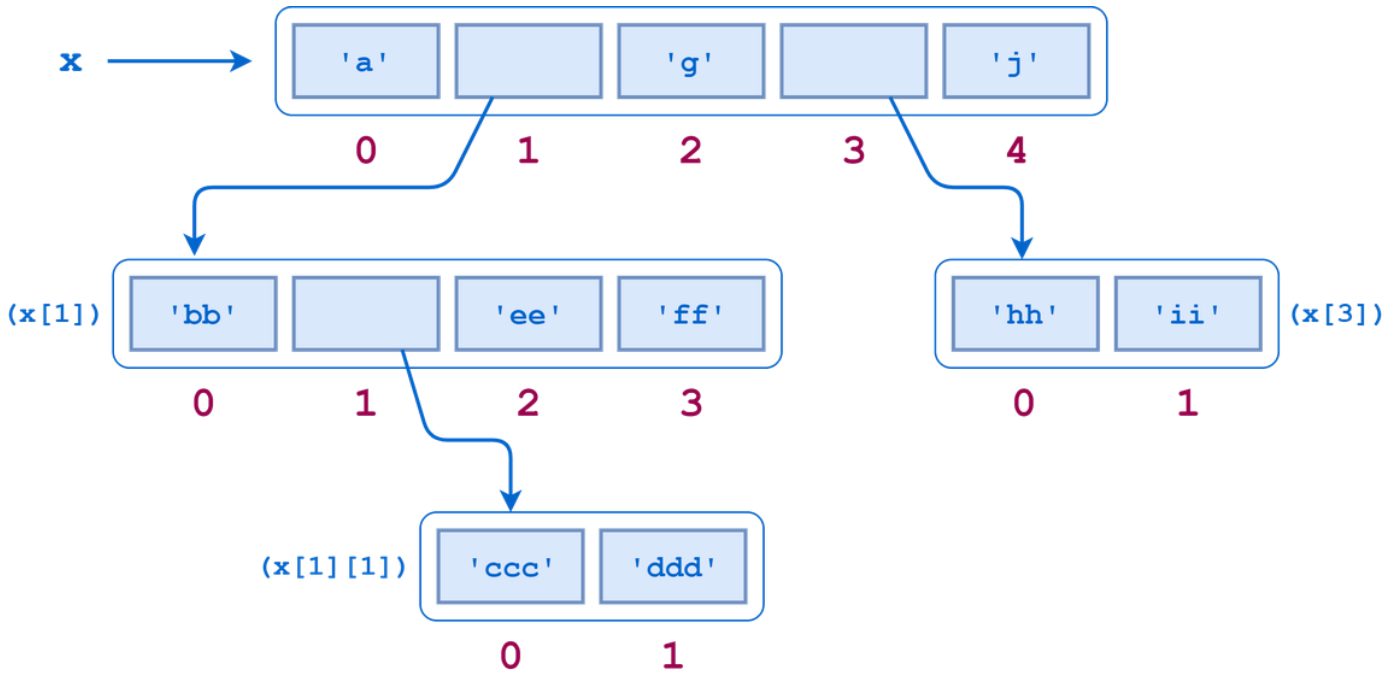
Elementi v listu so lahko poljubnega data type.

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 [1]:

```
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 [2]:

```
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 [ ]:

```

```

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: Touples are ordered
t  = ("pingvin", "medved", "los", "volk")
t2  = ("pingvin", "volk", "medved", "los")

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

```
Touples ARE ordered
```

In [137]:

```
# Primer: Touples 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: Touples 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: Touples 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 touple namesto list?

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

TECHNICAL

Treba pazit kadar inicializiramo touple 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 touple
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 evalua
the expression (2) as simply the integer 2 and creates an int object. To tell Python that y
to define a singleton tuple, include a trailing comma (,) just before the closing parenthes
'''

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

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

()
<class 'tuple'>

2
<class 'int'>

(2,)
<class 'tuple'>
```

In [ ]:

```

```

Vaja

Naloga: Iz sledečega tuple pridobite vrednost **cc**

```
our_tuple = ("a", ["bb", "cc"], "d", [("eee"), ["ffff"], "ggg"])
```

In [3]:

```
# Rešitev
our_tuple = ("a", ["bb", "cc"], "d", [("eee"), ["ffff"], "ggg"])

print(our_tuple[1][1])
```

```
cc
```

Vaja

> **Naloga:** Pri sledečem tuple vzemite zadnjih 5 vrednosti.

```
our_tuple = (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)

Rešitev:
(16, 17, 18, 19, 20)
```

In [4]:

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

```
(16, 17, 18, 19, 20)
```

In [ ]:

```

```

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, touple, ...)
- 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': 'Micka'}
```

Posodabljanje vrednosti.

In [150]:

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

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

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, touple.

Touple 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,"touple"):'e',
    }
print(d)
print(d[2.3])
print(d[None])
print(d[(1, "touple")])
# PAZI: Če daš True namest None bo narobe deloval. Pomojm tretira 1 kt True pa se mu zmeša
# Sej keywords dajat sm je nesmiselno
```

```
{1: 'a', 2.3: 'b', 'string': 'c', None: 'd', (1, 'touple'): '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 ponovit):

In [104]:

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

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

Built-in Dictionary Methods

```
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 touple.

> 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 zamenjanjte 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 [5]:

```
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 [6]:

```
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 1x izpi
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} # touple 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 mutab
le
      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 uzeme 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}
```

še več teh metod https://realpython.com/python-sets/ (https://realpython.com/python-sets/)

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 operations on f
rozensets
----> 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 [ ]:

```
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
```

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 [ ]:

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

set1 = set(str1)
set2 = set(str2)


set1.intersection(set2)
```

In [ ]:

```

```

In [ ]:

```
```