Chapter 7: Collection data structures

In Python, we have the following built-in data structures:

- Lists
- Dictionaries
- Tuples
- Sets

We have already introduced the **list** data structure in Chapter 3 (03_iterations_lists.pdf). Lists are *heterogeneous* containers for multiple items in Python.

Dictionaries

Unlike sequences (like *lists* in Python), which are indexed by a range of numbers, dictionaries are indexed by *keys*. It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}` . Placing a comma-separated list of *key: value* pairs within the braces adds initial *key: value* pairs to the dictionary; this is also the way dictionaries are written on output.

Dictionaries are sometimes found in other languages as *associative arrays*.

Example: let's create a shopping list with the name of products as keys and quantities as values:

In [1]:

```
shopping_list = {'apple': 6, 'bread': 2, 'milk': 6, 'butter': 1}
print(shopping_list)
```

```
{'apple': 6, 'bread': 2, 'milk': 6, 'butter': 1}
```

Example: let's create a dictionary with the name of countries as keys and their capital cities as values:

In [2]:

```
capitals = {
    'Aland Islands': 'Mariehamn',
    'Albania': 'Tirana',
    'Andorra': 'Andorra la Vella',
    'Armenia': 'Yerevan',
    'Austria': 'Vienna',
    ...
    'Switzerland': 'Bern',
    'Turkey': 'Ankara',
    'Ukraine': 'Kyiv',
    'United Kingdom': 'London',
    'Northern Cyprus': 'North Nicosia'
}
print(capitals)
```

```
{'Aland Islands': 'Mariehamn', 'Albania': 'Tirana', 'Andorra': 'Ando
rra la Vella', 'Armenia': 'Yerevan', 'Austria': 'Vienna', 'Azerbaija
n': 'Baku', 'Belarus': 'Minsk', 'Belgium': 'Brussels', 'Bosnia and H
erzegovina': 'Sarajevo', 'Bulgaria': 'Sofia', 'Croatia': 'Zagreb',
'Cyprus': 'Nicosia', 'Czech Republic': 'Prague', 'Denmark': 'Copenha
gen', 'Estonia': 'Tallinn', 'Faroe Islands': 'Torshavn', 'Finland':
'Helsinki', 'France': 'Paris', 'Georgia': 'Tbilisi', 'Germany': 'Ber
lin', 'Gibraltar': 'Gibraltar', 'Greece': 'Athens', 'Guernsey': 'Sai
nt Peter Port', 'Vatican City': 'Vatican City', 'Hungary': 'Budapes
t', 'Iceland': 'Reykjavik', 'Ireland': 'Dublin', 'Isle of Man': 'Dou
glas', 'Italy': 'Rome', 'Jersey': 'Saint Helier', 'Kosovo': 'Pristin
a', 'Latvia': 'Riga', 'Liechtenstein': 'Vaduz', 'Lithuania': 'Vilniu
s', 'Luxembourg': 'Luxembourg', 'Macedonia': 'Skopje', 'Malta': 'Val
letta', 'Moldova': 'Chisinau', 'Monaco': 'Monaco', 'Montenegro': 'Po
dgorica', 'Netherlands': 'Amsterdam', 'Norway': 'Oslo', 'Poland': 'W
arsaw', 'Portugal': 'Lisbon', 'Romania': 'Bucharest', 'Russia': 'Mos
cow', 'San Marino': 'San Marino', 'Serbia': 'Belgrade', 'Slovakia':
'Bratislava', 'Slovenia': 'Ljubljana', 'Spain': 'Madrid', 'Svalbar
d': 'Longyearbyen', 'Sweden': 'Stockholm', 'Switzerland': 'Bern', 'T
urkey': 'Ankara', 'Ukraine': 'Kyiv', 'United Kingdom': 'London', 'No
rthern Cyprus': 'North Nicosia'}
```

Elements of a dictionary can be accessed through their key:

In [3]:

```
print(capitals['Hungary'])
```

```
Budapest
```

Dictionaries are mutable, so the values can be modified:

In [4]:

```
capitals['Hungary'] = 'Esztergom' # old capital city of Hungary between the 11-13th century
print(capitals['Hungary'])

capitals['Hungary'] = 'Budapest'
print(capitals['Hungary'])
```

```
Esztergom
Budapest
```

Dictionaries can also be extended with new elements (*key: value* pairs):

In [5]:

```
capitals['USA'] = 'Washington'
print(capitals['USA'])
```

```
Washington
```

Removing or deleting existing elements is also possible:

In [6]:

```
del capitals['USA'] # deleting, because not in Europe
print(capitals['USA'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-6-7e32e4eb21a0> in <module>
      1 del capitals['USA'] # deleting, because not in Europe
----> 2 print(capitals['USA'])

KeyError: 'USA'
```

Tuples

Tuples are a sequence of heterogeneous elements, similarly like *lists*. Its initial elements are defined as a comma separated list, surrounded by parentheses.

Note: lists are surrounded by brackets!

In [8]:

```
neighbours = ('Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia')
print(neighbours)
```

```
('Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia')
```

The items of a list can be accessed by the numerical indexes. (The first item is indexed with *zero*.)

In [9]:

```
print(neighbours[0])
print(neighbours[2:5])
print(len(neighbours))
```

```
Austria
('Ukraine', 'Romania', 'Serbia')
7
```

The elements of tuple can also be fetched by *tuple unpacking*, which means that the items of a tuple are extracted into distinct variables:

In [10]:

```
a, b, c, d, e, f, g = neighbours
print(a, b, c, d, e, f, g)
```

```
Austria Slovakia Ukraine Romania Serbia Croatia Slovenia
```

Through *tuple packing*, we can create a **new** tuple with its elements defined:

In [11]:

```
neighbours2 = (a, b, c, d, e, f, g)
print(neighbours == neighbours2)
```

```
True
```

While lists are mutable, tuples are immutable, meaning that the elements cannot be modified:

In [12]:

```
neighbours[0] = 'Renamed country'
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-12-ee27e9767c8f> in <module>
----> 1 neighbours[0] = 'Renamed country'

TypeError: 'tuple' object does not support item assignment
```

New elements can neither be added to a tuple. Removing existing elements is also not possible.

In [13]:

```
neighbours.append('New country')
```

```
---------------------------------------------------------------------
-------
AttributeError                            Traceback (most recent cal
l last)
<ipython-input-13-1f4e9477ed13> in <module>
----> 1 neighbours.append('New country')

AttributeError: 'tuple' object has no attribute 'append'
```

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements. Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

We could see that lists, tuples and even strings have many common properties, such as indexing and slicing operations. They are **sequence data types**.

Dictionary as a list of tuples

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs. The dictionary in this case is build from a list tuples, each tuple containing precisely two elements: a key and a value, in this order.

In [16]:

```
capitals = dict([
    ('Aland Islands', 'Mariehamn'),
    ('Albania', 'Tirana'),
    ('Andorra', 'Andorra la Vella'),
    ('Armenia', 'Yerevan'),
    ('Austria', 'Vienna'),
    ...
    ('Switzerland', 'Bern'),
    ('Turkey', 'Ankara'),
    ('Ukraine', 'Kyiv'),
    ('United Kingdom', 'London'),
    ('Northern Cyprus', 'North Nicosia')
])
print(capitals)
```

```
{'Aland Islands': 'Mariehamn', 'Albania': 'Tirana', 'Andorra': 'Ando
rra la Vella', 'Armenia': 'Yerevan', 'Austria': 'Vienna', 'Azerbaija
n': 'Baku', 'Belarus': 'Minsk', 'Belgium': 'Brussels', 'Bosnia and H
erzegovina': 'Sarajevo', 'Bulgaria': 'Sofia', 'Croatia': 'Zagreb',
'Cyprus': 'Nicosia', 'Czech Republic': 'Prague', 'Denmark': 'Copenha
gen', 'Estonia': 'Tallinn', 'Faroe Islands': 'Torshavn', 'Finland':
'Helsinki', 'France': 'Paris', 'Georgia': 'Tbilisi', 'Germany': 'Ber
lin', 'Gibraltar': 'Gibraltar', 'Greece': 'Athens', 'Guernsey': 'Sai
nt Peter Port', 'Vatican City': 'Vatican City', 'Hungary': 'Budapes
t', 'Iceland': 'Reykjavik', 'Ireland': 'Dublin', 'Isle of Man': 'Dou
glas', 'Italy': 'Rome', 'Jersey': 'Saint Helier', 'Kosovo': 'Pristin
a', 'Latvia': 'Riga', 'Liechtenstein': 'Vaduz', 'Lithuania': 'Vilniu
s', 'Luxembourg': 'Luxembourg', 'Macedonia': 'Skopje', 'Malta': 'Val
letta', 'Moldova': 'Chisinau', 'Monaco': 'Monaco', 'Montenegro': 'Po
dgorica', 'Netherlands': 'Amsterdam', 'Norway': 'Oslo', 'Poland': 'W
arsaw', 'Portugal': 'Lisbon', 'Romania': 'Bucharest', 'Russia': 'Mos
cow', 'San Marino': 'San Marino', 'Serbia': 'Belgrade', 'Slovakia':
'Bratislava', 'Slovenia': 'Ljubljana', 'Spain': 'Madrid', 'Svalbar
d': 'Longyearbyen', 'Sweden': 'Stockholm', 'Switzerland': 'Bern', 'T
urkey': 'Ankara', 'Ukraine': 'Kyiv', 'United Kingdom': 'London', 'No
rthern Cyprus': 'North Nicosia'}
```

Tuple unpacking

Since we know that each tuple in a dictionary contains a key and a value, *tuple unpacking* can be very useful to extract them into separate variables, e.g.:

In [17]:

```
pair = ('Hungary', 'Budapest')
key, value = pair
print("Key is {0}, value is {1}".format(key, value))
```

```
Key is Hungary, value is Budapest
```

Iterating through a dictionary

Accessing the list of *key-value* tuples can be done with the `items()` function of the dictionary:

In [18]:

```
print("List of key-value pairs:")
print(capitals.items())
```

```
List of key-value pairs:
dict_items([('Aland Islands', 'Mariehamn'), ('Albania', 'Tirana'),
('Andorra', 'Andorra la Vella'), ('Armenia', 'Yerevan'), ('Austria',
'Vienna'), ('Azerbaijan', 'Baku'), ('Belarus', 'Minsk'), ('Belgium',
'Brussels'), ('Bosnia and Herzegovina', 'Sarajevo'), ('Bulgaria', 'S
ofia'), ('Croatia', 'Zagreb'), ('Cyprus', 'Nicosia'), ('Czech Republ
ic', 'Prague'), ('Denmark', 'Copenhagen'), ('Estonia', 'Tallinn'),
('Faroe Islands', 'Torshavn'), ('Finland', 'Helsinki'), ('France',
'Paris'), ('Georgia', 'Tbilisi'), ('Germany', 'Berlin'), ('Gibralta
r', 'Gibraltar'), ('Greece', 'Athens'), ('Guernsey', 'Saint Peter Po
rt'), ('Vatican City', 'Vatican City'), ('Hungary', 'Budapest'), ('I
celand', 'Reykjavik'), ('Ireland', 'Dublin'), ('Isle of Man', 'Dougl
as'), ('Italy', 'Rome'), ('Jersey', 'Saint Helier'), ('Kosovo', 'Pri
stina'), ('Latvia', 'Riga'), ('Liechtenstein', 'Vaduz'), ('Lithuani
a', 'Vilnius'), ('Luxembourg', 'Luxembourg'), ('Macedonia', 'Skopj
e'), ('Malta', 'Valletta'), ('Moldova', 'Chisinau'), ('Monaco', 'Mon
aco'), ('Montenegro', 'Podgorica'), ('Netherlands', 'Amsterdam'),
('Norway', 'Oslo'), ('Poland', 'Warsaw'), ('Portugal', 'Lisbon'),
('Romania', 'Bucharest'), ('Russia', 'Moscow'), ('San Marino', 'San
Marino'), ('Serbia', 'Belgrade'), ('Slovakia', 'Bratislava'), ('Slov
enia', 'Ljubljana'), ('Spain', 'Madrid'), ('Svalbard', 'Longyearbye
n'), ('Sweden', 'Stockholm'), ('Switzerland', 'Bern'), ('Turkey', 'A
nkara'), ('Ukraine', 'Kyiv'), ('United Kingdom', 'London'), ('Northe
rn Cyprus', 'North Nicosia')])
```

Accessing ONLY the list of *keys* or *values* is also possible with the `keys()` and `values()` functions of the dictionary:

In [19]:

```
print("List of keys:")
print(capitals.keys())
```

```
List of keys:
dict_keys(['Aland Islands', 'Albania', 'Andorra', 'Armenia', 'Austri
a', 'Azerbaijan', 'Belarus', 'Belgium', 'Bosnia and Herzegovina', 'B
ulgaria', 'Croatia', 'Cyprus', 'Czech Republic', 'Denmark', 'Estoni
a', 'Faroe Islands', 'Finland', 'France', 'Georgia', 'Germany', 'Gib
raltar', 'Greece', 'Guernsey', 'Vatican City', 'Hungary', 'Iceland',
'Ireland', 'Isle of Man', 'Italy', 'Jersey', 'Kosovo', 'Latvia', 'Li
echtenstein', 'Lithuania', 'Luxembourg', 'Macedonia', 'Malta', 'Mold
ova', 'Monaco', 'Montenegro', 'Netherlands', 'Norway', 'Poland', 'Po
rtugal', 'Romania', 'Russia', 'San Marino', 'Serbia', 'Slovakia', 'S
lovenia', 'Spain', 'Svalbard', 'Sweden', 'Switzerland', 'Turkey', 'U
kraine', 'United Kingdom', 'Northern Cyprus'])
```

In [20]:

```
print("List of values:")
print(capitals.values())
```

```
List of values:
dict_values(['Mariehamn', 'Tirana', 'Andorra la Vella', 'Yerevan',
'Vienna', 'Baku', 'Minsk', 'Brussels', 'Sarajevo', 'Sofia', 'Zagre
b', 'Nicosia', 'Prague', 'Copenhagen', 'Tallinn', 'Torshavn', 'Helsi
nki', 'Paris', 'Tbilisi', 'Berlin', 'Gibraltar', 'Athens', 'Saint Pe
ter Port', 'Vatican City', 'Budapest', 'Reykjavik', 'Dublin', 'Dougl
as', 'Rome', 'Saint Helier', 'Pristina', 'Riga', 'Vaduz', 'Vilnius',
'Luxembourg', 'Skopje', 'Valletta', 'Chisinau', 'Monaco', 'Podgoric
a', 'Amsterdam', 'Oslo', 'Warsaw', 'Lisbon', 'Bucharest', 'Moscow',
'San Marino', 'Belgrade', 'Bratislava', 'Ljubljana', 'Madrid', 'Long
yearbyen', 'Stockholm', 'Bern', 'Ankara', 'Kyiv', 'London', 'North N
icosia'])
```

We can also use a *for* loop to iterate through the items of a dictionary:

In [21]:

```
for item in capitals.items():
    print(item)
```

```
('Aland Islands', 'Mariehamn')
('Albania', 'Tirana')
('Andorra', 'Andorra la Vella')
('Armenia', 'Yerevan')
('Austria', 'Vienna')
...
('Switzerland', 'Bern')
('Turkey', 'Ankara')
('Ukraine', 'Kyiv')
('United Kingdom', 'London')
('Northern Cyprus', 'North Nicosia')
```

Here we iterate through the *key: value* tuples of a dictionary with the `item` variable.
The *key* is the element with the index 0, the *value* is the element with the index 1 in `item`:

In [22]:

```
for item in capitals.items():
    key = item[0]
    value = item[1]
    print("{0}: {1}".format(key, value))
```

```
Aland Islands: Mariehamn
Albania: Tirana
Andorra: Andorra la Vella
Armenia: Yerevan
Austria: Vienna
...
Switzerland: Bern
Turkey: Ankara
Ukraine: Kyiv
United Kingdom: London
Northern Cyprus: North Nicosia
```

By creating a list of tuples from the dictionary, we can fetch a single tuple by its numerical index and then extract the *key* and the *value* into separate variables with *tuple unpacking*:

In [23]:

```
item_10 = list(capitals.items())[10]
print(item_10)
key, value = item_10
print("Key is {0}, value is {1}".format(key, value))
```

```
('Croatia', 'Zagreb')
Key is Croatia, value is Zagreb
```

We can also use a *for* loop to iterate through the *key: value* pairs in a dictionary with *tuple unpacking*:

In [24]:

```
for item in capitals.items():
    key, value = item
    print("{0}: {1}".format(key, value))
```

```
Aland Islands: Mariehamn
Albania: Tirana
Andorra: Andorra la Vella
Armenia: Yerevan
Austria: Vienna
...
Switzerland: Bern
Turkey: Ankara
Ukraine: Kyiv
United Kingdom: London
Northern Cyprus: North Nicosia
```

We do not even need a temporary `item` variable, the unpacking can be done directly in the `for` statement:

In [25]:

```
for key, value in capitals.items():
    print("{0}: {1}".format(key, value))
```

```
Aland Islands: Mariehamn
Albania: Tirana
Andorra: Andorra la Vella
Armenia: Yerevan
Austria: Vienna
...
Switzerland: Bern
Turkey: Ankara
Ukraine: Kyiv
United Kingdom: London
Northern Cyprus: North Nicosia
```

Now compare the 2 versions of iterating through the items of a dictionary and observe how *tuple unpacking* makes accessing the key and the value easier:

```
for item in capitals.items():
    key = item[0]
    value = item[1]
    print("{0}: {1}".format(key, value))

for key, value in capitals.items():
    print("{0}: {1}".format(key, value))
```

Note: keys in a dictionary can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified.

Note: the same tuple unpacking were utilized when using the `enumerate()` function introduced in Chapter 5 (05_basic_algorithms.pdf#Maximum-search):

```
data = [ ... ]
for idx, value in enumerate(data):
    ...
```

Summary exercise on dictionaries

The dictionaries `population_2008` and `population_2018` store the population of some European countries in the according years (2008 and 2010):

In [26]:

```
population_2008 = { 'Belgium': 10666866, 'Bulgaria': 7518002, 'Czechia': 1034342
2, 'Denmark': 5475791, 'Germany': 82217837, 'Estonia': 1338440, 'Ireland': 44577
65, 'Greece': 11060937, 'Spain': 45668939, 'France': 64007193, 'Croatia': 431196
7, 'Italy': 58652875, 'Cyprus': 776333, 'Latvia': 2191810, 'Lithuania': 3212605,
 'Luxembourg': 483799, 'Hungary': 10045401, 'Malta': 407832, 'Netherlands': 1640
5399, 'Austria': 8307989, 'Poland': 38115641, 'Portugal': 10553339, 'Romania': 2
0635460, 'Slovenia': 2010269, 'Slovakia': 5376064, 'Finland': 5300484, 'Sweden':
 9182927, 'United Kingdom': 61571647, 'Iceland': 315459, 'Liechtenstein': 35356,
 'Norway': 4737171, 'Switzerland': 7593494, 'Montenegro': 615543, 'North Macedon
ia': 2045177, 'Albania': 2958266, 'Serbia': 7365507, 'Turkey': 70586256, 'Andorr
a': 83137, 'Belarus': 9689770, 'Bosnia and Herzegovina': 3843846, 'Kosovo': 2153
139, 'Moldova': 3572703, 'San Marino': 32054, 'Ukraine': 46192309, 'Armenia': 32
30086, 'Azerbaijan': 8629900, 'Georgia': 4382070 }
population_2018 = { 'Belgium': 11398589, 'Bulgaria': 7050034, 'Czechia': 1061005
5, 'Denmark': 5781190, 'Germany': 82792351, 'Estonia': 1319133, 'Ireland': 48303
92, 'Greece': 10741165, 'Spain': 46658447, 'France': 66926166, 'Croatia': 410549
3, 'Italy': 60483973, 'Cyprus': 864236, 'Latvia': 1934379, 'Lithuania': 2808901,
 'Luxembourg': 602005, 'Hungary': 9778371, 'Malta': 475701, 'Netherlands': 17181
084, 'Austria': 8822267, 'Poland': 37976687, 'Portugal': 10291027, 'Romania': 19
530631, 'Slovenia': 2066880, 'Slovakia': 5443120, 'Finland': 5513130, 'Sweden':
10120242, 'United Kingdom': 66273576, 'Iceland': 348450, 'Liechtenstein': 38114,
 'Norway': 5295619, 'Switzerland': 8484130, 'Montenegro': 622359, 'North Macedon
ia': 2075301, 'Albania': 2870324, 'Serbia': 7001444, 'Turkey': 80810525, 'Andorr
a': 74794, 'Belarus': 9491823, 'Bosnia and Herzegovina': 3502550, 'Kosovo': 1798
506, 'Moldova': 3547539, 'San Marino': 34453, 'Ukraine': 42386403, 'Armenia': 29
72732, 'Azerbaijan': 9898085, 'Georgia': 3729633 }

print("Population of European countries in 2008:")
print(population_2008)

print()
print("Population of European countries in 2018:")
print(population_2018)
```

```
Population of European countries in 2008:
{'Belgium': 10666866, 'Bulgaria': 7518002, 'Czechia': 10343422, 'Den
mark': 5475791, 'Germany': 82217837, 'Estonia': 1338440, 'Ireland':
4457765, 'Greece': 11060937, 'Spain': 45668939, 'France': 64007193,
'Croatia': 4311967, 'Italy': 58652875, 'Cyprus': 776333, 'Latvia': 2
191810, 'Lithuania': 3212605, 'Luxembourg': 483799, 'Hungary': 10045
401, 'Malta': 407832, 'Netherlands': 16405399, 'Austria': 8307989,
'Poland': 38115641, 'Portugal': 10553339, 'Romania': 20635460, 'Slov
enia': 2010269, 'Slovakia': 5376064, 'Finland': 5300484, 'Sweden': 9
182927, 'United Kingdom': 61571647, 'Iceland': 315459, 'Liechtenstei
n': 35356, 'Norway': 4737171, 'Switzerland': 7593494, 'Montenegro':
615543, 'North Macedonia': 2045177, 'Albania': 2958266, 'Serbia': 73
65507, 'Turkey': 70586256, 'Andorra': 83137, 'Belarus': 9689770, 'Bo
snia and Herzegovina': 3843846, 'Kosovo': 2153139, 'Moldova': 357270
3, 'San Marino': 32054, 'Ukraine': 46192309, 'Armenia': 3230086, 'Az
erbaijan': 8629900, 'Georgia': 4382070}

Population of European countries in 2018:
{'Belgium': 11398589, 'Bulgaria': 7050034, 'Czechia': 10610055, 'Den
mark': 5781190, 'Germany': 82792351, 'Estonia': 1319133, 'Ireland':
4830392, 'Greece': 10741165, 'Spain': 46658447, 'France': 66926166,
'Croatia': 4105493, 'Italy': 60483973, 'Cyprus': 864236, 'Latvia': 1
934379, 'Lithuania': 2808901, 'Luxembourg': 602005, 'Hungary': 97783
71, 'Malta': 475701, 'Netherlands': 17181084, 'Austria': 8822267, 'P
oland': 37976687, 'Portugal': 10291027, 'Romania': 19530631, 'Sloven
ia': 2066880, 'Slovakia': 5443120, 'Finland': 5513130, 'Sweden': 101
20242, 'United Kingdom': 66273576, 'Iceland': 348450, 'Liechtenstei
n': 38114, 'Norway': 5295619, 'Switzerland': 8484130, 'Montenegro':
622359, 'North Macedonia': 2075301, 'Albania': 2870324, 'Serbia': 70
01444, 'Turkey': 80810525, 'Andorra': 74794, 'Belarus': 9491823, 'Bo
snia and Herzegovina': 3502550, 'Kosovo': 1798506, 'Moldova': 354753
9, 'San Marino': 34453, 'Ukraine': 42386403, 'Armenia': 2972732, 'Az
erbaijan': 9898085, 'Georgia': 3729633}
```

Data source: EuroStat (https://ec.europa.eu/eurostat/)

Exercise 1

Task: What was the population of Hungary in 2008 and in 2018?

In [27]:

```
print("Population of Hungary in 2008: {0}".format(population_2008["Hungary"]))
print("Population of Hungary in 2018: {0}".format(population_2018["Hungary"]))
```

```
Population of Hungary in 2008: 10045401
Population of Hungary in 2018: 9778371
```

Exercise 2

Task: What was the population change between 2008 and 2018 in Hungary? What is the average change per year?

In [28]:

```
diff = population_2018["Hungary"] - population_2008["Hungary"]
print("Population difference for Hungary: {0}".format(diff))

diff_avg = diff // 10
print("Average population change for Hungary per year: {0}".format(diff_avg))
```

```
Population difference for Hungary: -267030
Average population change for Hungary per year: -26703
```

Exercise 3

Task: Display for all countries the population change between 2008 and 2018!

In [29]:

```
for key in population_2008.keys():
    diff = population_2018[key] - population_2008[key]
    print("{0}: {1}".format(key, diff))
```

```
Belgium: 731723
Bulgaria: -467968
Czechia: 266633
Denmark: 305399
Germany: 574514
...
San Marino: 2399
Ukraine: -3805906
Armenia: -257354
Azerbaijan: 1268185
Georgia: -652437
```

Exercise 4

Task: Which country had the largest population growth in the given timespan? Which one had the largest population decline?

In [30]:

```
max_country = "Hungary"
max_diff = population_2018[max_country] - population_2008[max_country]
min_country = "Hungary"
min_diff = population_2018[min_country] - population_2008[min_country]

for key in population_2008.keys():
    diff = population_2018[key] - population_2008[key]
    if diff > max_diff:
        max_diff = diff
        max_country = key
    if diff < min_diff:
        min_diff = diff
        min_country = key

print("Largest growth: {0} ({1})".format(max_country, max_diff))
print("Largest decline: {0} ({1})".format(min_country, min_diff))
```

```
Largest growth: Turkey (10224269)
Largest decline: Ukraine (-3805906)
```

Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic usage include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets.
Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary and not a set.

In [31]:

```
neighbours = {'Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia'}
print(neighbours)
```

```
{'Slovakia', 'Romania', 'Austria', 'Ukraine', 'Croatia', 'Serbia',
'Slovenia'}
```

Since sets are unordered theoretically, Python does not support indexing for sets, meaning we cannot access an item with a numerical index:

In [32]:

```
print(neighbours[0])
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-32-a5f7753d713b> in <module>
----> 1 print(neighbours[0])

TypeError: 'set' object is not subscriptable
```

However, we can perform membership testing, evaluating whether an item is in the list or not:

In [33]:

```
print('Serbia' in neighbours)
print('Germany' in neighbours)
```

```
True
False
```

New element can be added with the `add()` method to a set, existing element can be removed with the `remove()` method from a set. Sets also guarantee to contain no duplicate entries:

In [34]:

```
neighbours.add('Ukraine') # already in the set
print(neighbours)
```

```
{'Slovakia', 'Romania', 'Austria', 'Ukraine', 'Croatia', 'Serbia',
'Slovenia'}
```

Demonstration of basic set operations:

In [35]:

```
german_speakers = {'Germany', 'Austria', 'Switzerland'}
```

In [36]:

```
print("Union: {0}".format(neighbours | german_speakers))
print("Intersection: {0}".format(neighbours & german_speakers))
print("Difference: {0}".format(neighbours - german_speakers))
print("Symmetric difference: {0}".format(neighbours ^ german_speakers))
```

```
Union: {'Slovakia', 'Romania', 'Switzerland', 'Austria', 'Ukraine',
'Croatia', 'Serbia', 'Slovenia', 'Germany'}
Intersection: {'Austria'}
Difference: {'Slovakia', 'Romania', 'Ukraine', 'Croatia', 'Serbia',
'Slovenia'}
Symmetric difference: {'Slovakia', 'Romania', 'Switzerland', 'Ukrain
e', 'Croatia', 'Serbia', 'Slovenia', 'Germany'}
```

Exercise 5

Task: Verify whether the dictionaries `population_2008` and `population_2018` contain exactly the same countries as keys.

In [37]:

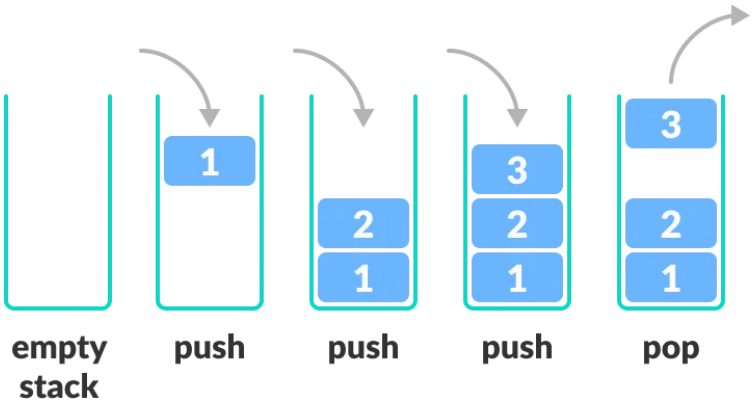
```
keyset_2008 = set(population_2008.keys())
keyset_2018 = set(population_2018.keys())
nomatch = keyset_2008 ^ keyset_2018

if len(nomatch) == 0:
    print("The 2 dictionaries contains the same countries.")
else:
    print("There are some countries only present in one of the dictionaries: {0}
".format(nomatch))
```

```
The 2 dictionaries contains the same countries.
```

Stacks

A stack is an abstract data structure that follows the "*last-in-first-out*" or *LIFO* model: elements are added to the top of the stack and only the top element of a stack can be removed.
Some well-known real world examples for usage of stacks:

- undoing the operations in a text editor;
- going back to the previous web page in a browser.



The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (*last-in, first-out*). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index.

In [38]:

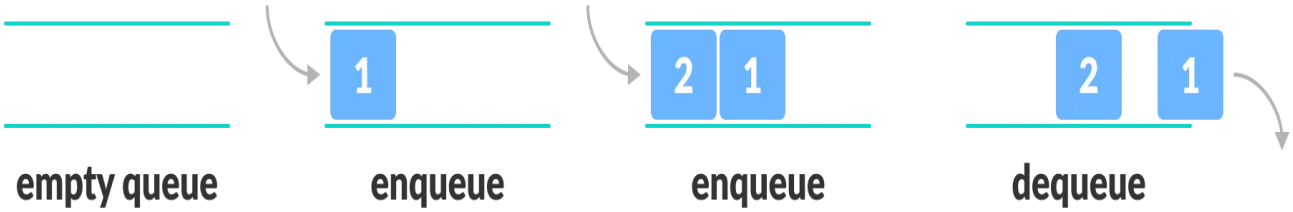
```
stack = [1, 2, 3, 4, 5]
stack.append(6)
stack.append(7)
print(stack)
print(stack.pop())
print(stack.pop())
print(stack)
stack.append(8)
stack.append(9)
print(stack)

print("Process all the elements of the stack:")
while len(stack) > 0:
    print(stack.pop())
```

```
[1, 2, 3, 4, 5, 6, 7]
7
6
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 8, 9]
Process all the elements of the stack:
9
8
5
4
3
2
1
```

Queues

A queue is an abstract data structure that follows "*first-in-first-out*" or *FIFO model*: new elements are added to the back of queue and only the front element of the queue can be removed.
Some well-known real world examples for usage of queues:

- waiting in a line (in a polite way);
- a printer machine's queue of files to be printed.



It is also possible to use a list as a queue, where the first element added is the first element retrieved (*first-in, first-out*); however, **lists are not efficient for this purpose**. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends.

In [39]:

```
from collections import deque

queue = deque([1, 2, 3, 4, 5])
queue.append(6)
queue.append(7)
print(queue)
print(queue.popleft())
print(queue.popleft())
print(queue)
queue.append(8)
queue.append(9)
print(queue)

print("Process all the elements of the stack:")
while len(queue) > 0:
    print(queue.popleft())
```

```
deque([1, 2, 3, 4, 5, 6, 7])
1
2
deque([3, 4, 5, 6, 7])
deque([3, 4, 5, 6, 7, 8, 9])
Process all the elements of the stack:
3
4
5
6
7
8
9
```

Note: `deque` is short for *double ended queue*, because we can manage both ends of the data structure (add or remove elements).