Lesson_4 Dictionaries and sets

March 18, 2020

1 Dictionaries and sets

1.0.1 Agenda

- What's a Python Dictionary
- Hashes and hashable values
- How dictionaries can be created
- Interacting with dict data
- Shallow copy vs. Deep copy
- Dictionary view objects
- Alternative containers: defaultdict, OrderedDict and Counter

1.0.2 Resources

- Python reference - Dictionaries
- Mapping Types
- PEP 274 – Dict Comprehensions
- Collections

1.0.3 What's a Python dictionary

A Python dictionary is an **unordered** and **mutable** Python **container** that stores **mappings** of unique keys to values

Generally known as an associative array, consists of a collection of key-value pairs (each key-value pair maps a key to an associated value)

```
dictionary = {
   <key>: <value>,
   <key>: <value>,
   ...
}
```

A **mapping** object maps hashable values to arbitrary objects. Python has only one standard mapping type - the dictionary.

[1]:
```
import pprint


pretty_printer = pprint.PrettyPrinter(indent=4)

population = {
    'Berlin': 3748148,
    'Hamburg': 1822445,
    'Munich': 1471508,
    'Cologne': 1085664,
    'Frankfurt': 753056
}
pretty_printer.pprint(population)

capitals = {
    'Austria': 'Vienna',
    'Germany': 'Berlin',
    'Netherlands': 'Amsterdam'
}
pretty_printer.pprint(capitals)


cities = {
    'Austria': [
        'Vienna', 'Graz', 'Linz', 'Salzburg'
    ],
    'Germany': [
        'Berlin', 'Hamburg', 'Munich', 'Cologne'
    ],
    'Netherlands': [
        'Amsterdam', 'Rotterdam', 'The Hague', 'Utrecht'
    ]
}
pretty_printer.pprint(cities)
```

```
{   'Berlin': 3748148,
    'Cologne': 1085664,
    'Frankfurt': 753056,
    'Hamburg': 1822445,
    'Munich': 1471508}
{'Austria': 'Vienna', 'Germany': 'Berlin', 'Netherlands': 'Amsterdam'}
{   'Austria': ['Vienna', 'Graz', 'Linz', 'Salzburg'],
    'Germany': ['Berlin', 'Hamburg', 'Munich', 'Cologne'],
    'Netherlands': ['Amsterdam', 'Rotterdam', 'The Hague', 'Utrecht']}
```

[2]:
```
faulty_dict = {
    (1, 2): 1,
    (1, [2, 2]): 2
}
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-2-e65a1c768cf4> in <module>
  1 faulty_dict = {
  2     (1, 2): 1,
----> 3     (1, [2, 2]): 2
  4 }


TypeError: unhashable type: 'list'
```

1.0.4 Hashes and hashable values

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.

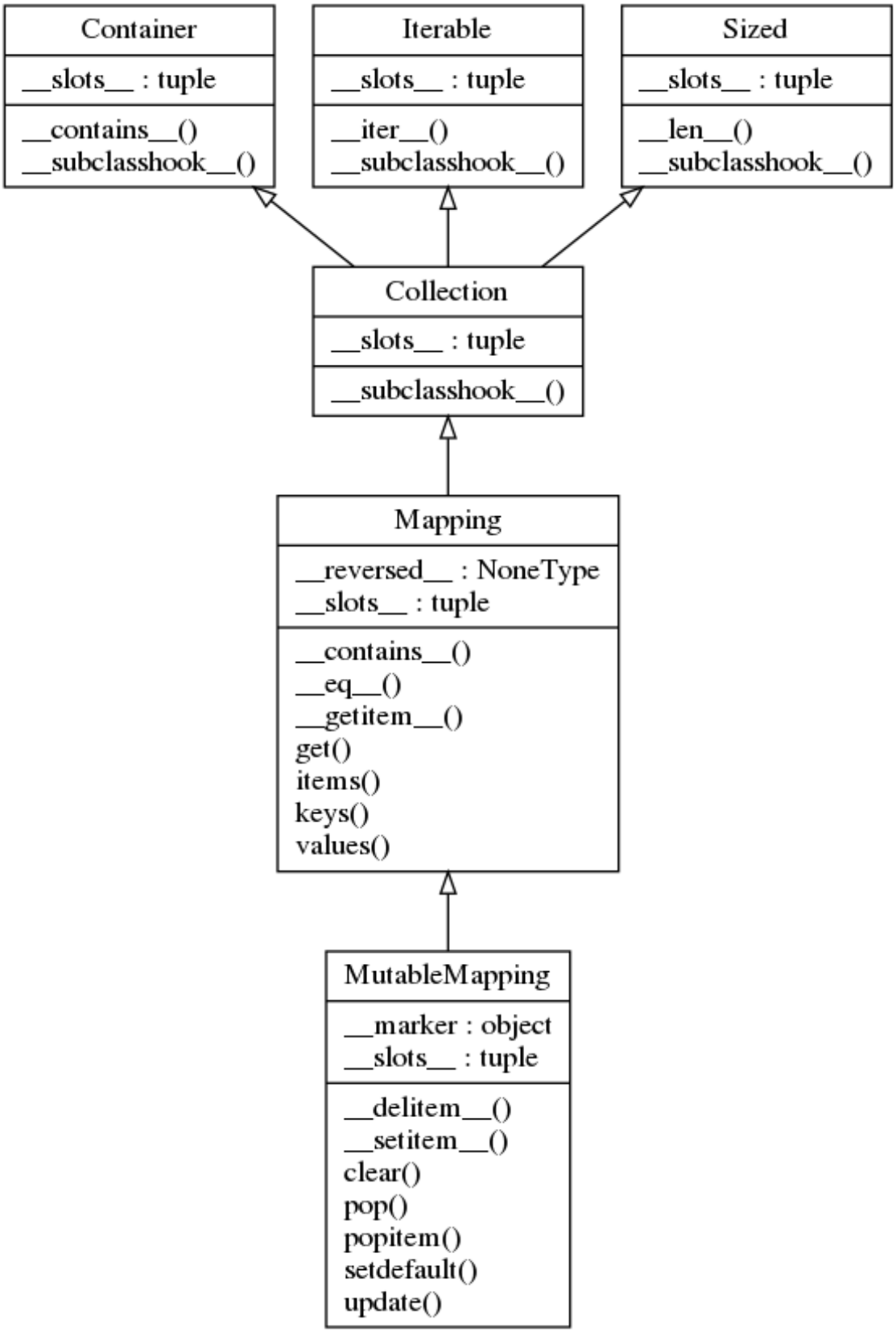
All of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) > are not. Objects which are instances of user-defined classes are hashable by default. They all compare > unequal (except with themselves), and their hash value is derived from their id()."

[6]:
```
simple_integer = 12
print(hash(simple_integer))

simple_float = 1.2
print(hash(simple_float))

simple_string = 'random string'
print(hash(simple_string))

simple_tuple = (1, 'two', 3)
```

```
print(hash(simple_tuple))

tuple_of_tuples = ((1, 2), (3, 4), (4, 5))
print(hash(tuple_of_tuples))

tuple_of_lists = ([1, 2], (3, 4), [5, 6])
print(hash(tuple_of_lists))
```

```
12
461168601842738689
-4794976470439490725
9163880764000795146
5631452659006521503


      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-6-89feb6cc31e8> in <module>
        15
        16 tuple_of_lists = ([1, 2], (3, 4), [5, 6])
   ---> 17 print(hash(tuple_of_lists))


       TypeError: unhashable type: 'list'
```

1.0.5 How dictionaries can be created



[7]:
```
from collections.abc import MutableMapping, Mapping, Iterable, Collection,␣
↪Container, Sized

my_dict = {}

type(my_dict)
```

[7]:
```
dict
```

[8]:
```
isinstance(my_dict, MutableMapping)
```

[8]:
```
True
```

[9]:
```
isinstance(my_dict, Mapping)
```

[9]:
```
True
```

[10]:
```
isinstance(my_dict, Iterable)
```

[10]:
```
True
```

[11]:
```
isinstance(my_dict, Collection)
```

[11]:
```
True
```

[12]:
```
isinstance(my_dict, Container)
```

[12]:
```
True
```

[13]:
```
isinstance(my_dict, Sized)
```

[13]:
```
True
```

Mapping Types — dict

```
class dict(**kwarg)
class dict(mapping, **kwarg)
class dict(iterable, **kwarg)
```

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created.

If a positional argument is given and it is a mapping object, a dictionary is created with the same key-> value pairs as the mapping object.

Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes

a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

[14]:
```
# Return a new dictionary initialized from an optional positional
# argument and a possibly empty set of keyword arguments.
dict_1 = dict(one=1, two=2, three=3)

print(dict_1)
```

```
{'one': 1, 'two': 2, 'three': 3}
```

[15]:
```
print(dict())
```

```
{}
```

[16]:
```
# If a positional argument is given and it is a mapping object,
# a dictionary is created with the same key-> value pairs as the mapping object.
dict_2 = dict({'one': 1, 'two': 2, 'three': 3})

print(dict_2)

dict_3 = {'one': 1, 'two': 2, 'three': 3}

print(dict_3)
print(type(dict_3))
```

```
{'one': 1, 'two': 2, 'three': 3}
{'one': 1, 'two': 2, 'three': 3}
<class 'dict'>
```

[17]:
```
from collections.abc import Iterable

first_iterable = zip(['one', 'two', 'three'], [1, 2, 3])
print(isinstance(first_iterable, Iterable))
```

```
True
```

[18]:
```
dict_4 = dict(first_iterable)
print(type(dict_4))

print(dict_4)
```

```
<class 'dict'>
{'one': 1, 'two': 2, 'three': 3}
```

[19]:
```
from collections.abc import Iterable

second_iterable = zip(['one', 'two', 'three'], [1, 2, 3])
print(isinstance(second_iterable, Iterable))
```

```
True
```

Dictionary comprehension A **dictcomp** builds a dict instance by producing key:value pair from any iterable

[20]:
```
population = [
    ('Berlin', 3748148),
    ('Hamburg', 1822445),
    ('Munich', 1471508),
    ('Cologne', 1085664),
    ('Frankfurt', 753056)
]
print(type(population))

population_as_dict = {
    city: citizens for city, citizens in population
}
print(type(population_as_dict))
print(population_as_dict)
```

```
<class 'list'>
<class 'dict'>
{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664,
'Frankfurt': 753056}
```

[78]:
```
cities = [
    # city, population, surface in km2
    ('Berlin', 3748148, 891.1 ),
    ('Hamburg', 1822445, 755.22),
    ('Munich', 1471508, 310.4),
    ('Cologne', 1085664, 405.2),
    ('Frankfurt', 753056, 248.3)
]

cities_by_name = {
    city: [population, surface]
    for city, population, surface in cities
}
"""
```

```
{
 'Berlin': [3748148, 891.1],
 'Hamburg': [1822445, 755.22],
 'Munich': [1471508, 310.4],
 'Cologne': [1085664, 405.2],
 'Frankfurt': [753056, 248.3]
}
"""

cities_suface = {
    city: surface
    for city, _, surface in cities
}
"""
{
 'Berlin': 891.1,
 'Hamburg': 755.22,
 'Munich': 310.4,
 'Cologne': 405.2,
 'Frankfurt': 248.3
}
"""

cities_populations = {
    city: population
    for city, population, _ in cities
}
"""
{
 'Berlin': 3748148,
 'Hamburg': 1822445,
 'Munich': 1471508,
 'Cologne': 1085664,
 'Frankfurt': 753056
}
"""

big_cities = {
    city: surface
    for city, _, surface in cities
    if surface > 400
}
"""
{
 'Berlin': 891.1,
 'Hamburg': 755.22,
 'Cologne': 405.2
```

```
}
"""

big_cities_as_list = [
    surface
    for _, _, surface in cities
    if surface > 400
]
"""
[891.1, 755.22, 405.2]
"""

city_names = [
    key
    for key in cities_by_name.keys()
]
"""
['Berlin', 'Hamburg', 'Munich', 'Cologne', 'Frankfurt']
"""

surface = {
    city: str(surface) + ' km2' # never concatenate strings like this, we will␣
 ↪see how to properly do it ;)
    for city, _, surface in cities
}
"""
{
 'Berlin': '891.1 km2',
 'Hamburg': '755.22 km2',
 'Munich': '310.4 km2',
 'Cologne': '405.2 km2',
 'Frankfurt': '248.3 km2'
}
"""
```

[78]:
```
"\n{\n 'Berlin': '891.1 km2',\n 'Hamburg': '755.22 km2',\n 'Munich': '310.4
km2',\n 'Cologne': '405.2 km2',\n 'Frankfurt': '248.3 km2'\n}\n"
```

1.0.6 Interacting with dict data

[21]:
```
# Access values in a dictionary
population_as_dict = {
    'Berlin': 3748148,
    'Hamburg': 1822445,
    'Munich': 1471508,
    'Cologne': 1085664,
```

```
    'Frankfurt': 753056
}

berlin = population_as_dict['Berlin']
print(type(berlin))
print(berlin)
```

```
<class 'int'>
3748148
```

[22]:
```
population_as_dict['Paris']
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)

<ipython-input-22-fda95eb74691> in <module>
----> 1 population_as_dict['Paris']


KeyError: 'Paris'
```

[23]:
```
population_as_dict[1]
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)

<ipython-input-23-5274e10ccc7b> in <module>
----> 1 population_as_dict[1]


KeyError: 1
```

[60]:
```
help(dict)
```

```
Help on class dict in module builtins:

class dict(object)
 |  dict() -> new empty dictionary
```

```
|  dict(mapping) -> new dictionary initialized from a mapping object's
|      (key, value) pairs
|  dict(iterable) -> new dictionary initialized as if via:
|      d = {}
|      for k, v in iterable:
|          d[k] = v
|  dict(**kwargs) -> new dictionary initialized with the name=value pairs
|      in the keyword argument list.  For example:  dict(one=1, two=2)
|
|  Methods defined here:
|
|  __contains__(self, key, /)
|      True if the dictionary has the specified key, else False.
|
|  __delitem__(self, key, /)
|      Delete self[key].
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(…)
|      x.__getitem__(y) <==> x[y]
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __ne__(self, value, /)
```

```
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(…)
 |      D.__sizeof__() -> size of D in memory, in bytes
 |
 |  clear(…)
 |      D.clear() -> None.  Remove all items from D.
 |
 |  copy(…)
 |      D.copy() -> a shallow copy of D
 |
 |  get(self, key, default=None, /)
 |      Return the value for key if key is in the dictionary, else default.
 |
 |  items(…)
 |      D.items() -> a set-like object providing a view on D's items
 |
 |  keys(…)
 |      D.keys() -> a set-like object providing a view on D's keys
 |
 |  pop(…)
 |      D.pop(k[,d]) -> v, remove specified key and return the corresponding
value.
 |      If key is not found, d is returned if given, otherwise KeyError is
raised
 |
 |  popitem(…)
 |      D.popitem() -> (k, v), remove and return some (key, value) pair as a
 |      2-tuple; but raise KeyError if D is empty.
 |
 |  setdefault(self, key, default=None, /)
 |      Insert key with a value of default if key is not in the dictionary.
 |
 |      Return the value for key if key is in the dictionary, else default.
 |
 |  update(…)
 |      D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
 |      If E is present and has a .keys() method, then does:  for k in E: D[k] =
E[k]
 |      If E is present and lacks a .keys() method, then does:  for k, v in E:
D[k] = v
 |      In either case, this is followed by: for k in F:  D[k] = F[k]
```

```
|
|  values(…)
|      D.values() -> an object providing a view on D's values
|
|  ----------------------------------------------------------------------
|  Class methods defined here:
|
|  fromkeys(iterable, value=None, /) from builtins.type
|      Create a new dictionary with keys from iterable and values set to value.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __hash__ = None
```

[61]:
```
population_as_dict.get('Cologne')
```

[61]: 1085664

[24]:
```
print(population_as_dict.get('Paris'))
```

```
None
```

[25]:
```
print(population_as_dict.get('Paris', 'Not found'))
```

```
Not found
```

[26]:
```
population_as_dict['Paris'] = 2193031

print(population_as_dict)
```

```
{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664,
'Frankfurt': 753056, 'Paris': 2193031}
```

[27]:
```
population_as_dict['Paris']
```

[27]: 2193031

[28]:
```
population_as_dict['Paris'] = 3000000
population_as_dict['Paris']
```

[28]:
```
3000000
```

[29]:
```
population_as_dict.update({'Vienna': 1888776, 'Amsterdam': 758198})
print(population_as_dict)
```

```
{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664,
'Frankfurt': 753056, 'Paris': 3000000, 'Vienna': 1888776, 'Amsterdam': 758198}
```

[30]:
```
population_as_dict.update(Salzburg=153377)
print(population_as_dict)
```

```
{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664,
'Frankfurt': 753056, 'Paris': 3000000, 'Vienna': 1888776, 'Amsterdam': 758198,
'Salzburg': 153377}
```

[31]:
```
austrian_population = [
    ('Graz', 222326),
    ('Linz', 204846),
    ('Innsbruck', 132493)
]

population_as_dict.update(austrian_population)
print(population_as_dict)
```

```
{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664,
'Frankfurt': 753056, 'Paris': 3000000, 'Vienna': 1888776, 'Amsterdam': 758198,
'Salzburg': 153377, 'Graz': 222326, 'Linz': 204846, 'Innsbruck': 132493}
```

!! The *update* method updates the dictionary with the key/value pairs from other, overwriting existing keys.

[32]:
```
print('Paris' in population_as_dict)

del population_as_dict['Paris']

print('Paris' in population_as_dict)
```

```
True
False
```

[33]:
```
# key does not exists
del population_as_dict['Cluj-Napoca']
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)
```

```
<ipython-input-33-bd10e920c7ba> in <module>
  1 # key does not exists
----> 2 del population_as_dict['Cluj-Napoca']


KeyError: 'Cluj-Napoca'
```

[34]:
```
print(population_as_dict)
```

```
{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664,
'Frankfurt': 753056, 'Vienna': 1888776, 'Amsterdam': 758198, 'Salzburg': 153377,
'Graz': 222326, 'Linz': 204846, 'Innsbruck': 132493}
```

[35]:
```
vienna = population_as_dict.pop('Vienna')
print(vienna)
```

```
1888776
```

[50]:
```
# key does not exists
cluj = population_as_dict.pop('Cluj-Napoca')
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)

<ipython-input-50-c35ef8fe733c> in <module>
  1 # key does not exists
----> 2 cluj = population_as_dict.pop('Cluj-Napoca')


KeyError: 'Cluj-Napoca'
```

[59]:
```
# key does not exists, but with default value
cluj = population_as_dict.pop('Cluj-Napoca', 'City not found')
print(cluj)
```

```
City not found
```

[51]:
```
for city in population_as_dict:
    print(city)
    print(population_as_dict[city])
```

```
Berlin
3748148
Hamburg
1822445
Munich
1471508
Cologne
1085664
Frankfurt
753056
Amsterdam
758198
Salzburg
153377
Graz
222326
Linz
204846
Innsbruck
132493
```

1.0.7 Shallow copy vs. Deep copy

[52]:
```
capitals = {
    'Austria': 'Vienna',
    'Germany': 'Berlin',
    'Netherlands': 'Amsterdam'
}
print(capitals)

capitals_copy = capitals.copy()
print(capitals_copy)

print(capitals == capitals_copy)

capitals_copy['Austria'] = 'London'
print(capitals)
print(capitals_copy)

print(capitals == capitals_copy)
```

```
{'Austria': 'Vienna', 'Germany': 'Berlin', 'Netherlands': 'Amsterdam'}
{'Austria': 'Vienna', 'Germany': 'Berlin', 'Netherlands': 'Amsterdam'}
True
{'Austria': 'Vienna', 'Germany': 'Berlin', 'Netherlands': 'Amsterdam'}
{'Austria': 'London', 'Germany': 'Berlin', 'Netherlands': 'Amsterdam'}
False
```

[53]:
```
major_cities = {
    'Austria': [
        'Vienna', 'Graz', 'Linz', 'Salzburg'
    ],
    'Germany': [
        'Berlin', 'Hamburg', 'Munich', 'Cologne'
    ],
    'Netherlands': [
        'Amsterdam', 'Rotterdam', 'The Hague', 'Utrecht'
    ]
}
major_cities_copy = major_cities.copy()

print(major_cities == major_cities_copy)

major_cities['Austria'][0] = 'Iasi'
print(major_cities)

print(major_cities_copy)
print(major_cities == major_cities_copy)
```

```
True
{'Austria': ['Iasi', 'Graz', 'Linz', 'Salzburg'], 'Germany': ['Berlin',
'Hamburg', 'Munich', 'Cologne'], 'Netherlands': ['Amsterdam', 'Rotterdam', 'The
Hague', 'Utrecht']}
{'Austria': ['Iasi', 'Graz', 'Linz', 'Salzburg'], 'Germany': ['Berlin',
'Hamburg', 'Munich', 'Cologne'], 'Netherlands': ['Amsterdam', 'Rotterdam', 'The
Hague', 'Utrecht']}
True
```

[54]:
```
import copy

major_cities = {
    'Austria': [
        'Vienna', 'Graz', 'Linz', 'Salzburg'
    ],
    'Germany': [
        'Berlin', 'Hamburg', 'Munich', 'Cologne'
    ],
    'Netherlands': [
        'Amsterdam', 'Rotterdam', 'The Hague', 'Utrecht'
    ]
}
major_cities_copy = copy.deepcopy(major_cities)

print(major_cities == major_cities_copy)
```

```
major_cities['Austria'][0] = 'Iasi'
print(major_cities)

print(major_cities_copy)
print(major_cities == major_cities_copy)
```

```
True
{'Austria': ['Iasi', 'Graz', 'Linz', 'Salzburg'], 'Germany': ['Berlin',
'Hamburg', 'Munich', 'Cologne'], 'Netherlands': ['Amsterdam', 'Rotterdam', 'The
Hague', 'Utrecht']}
{'Austria': ['Vienna', 'Graz', 'Linz', 'Salzburg'], 'Germany': ['Berlin',
'Hamburg', 'Munich', 'Cologne'], 'Netherlands': ['Amsterdam', 'Rotterdam', 'The
Hague', 'Utrecht']}
False
```

1.1 Dictionary view objects

Python official reference > >The objects returned by dict.keys(), dict.values() and dict.items() are view objects. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. > >Dictionary views can be iterated over to yield their respective data, and support membership tests >

1.1.1 keys

[55]:
```
print(population_as_dict)

cities = population_as_dict.keys()
print(type(cities))
```

```
{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664,
'Frankfurt': 753056, 'Amsterdam': 758198, 'Salzburg': 153377, 'Graz': 222326,
'Linz': 204846, 'Innsbruck': 132493}
<class 'dict_keys'>
```

[56]:
```
from collections.abc import Iterable

print(isinstance(cities, Iterable))

print(list(cities))
print(tuple(cities))

print('------')

for city in cities:
    print(city)
```

```
True
['Berlin', 'Hamburg', 'Munich', 'Cologne', 'Frankfurt', 'Amsterdam', 'Salzburg',
'Graz', 'Linz', 'Innsbruck']
('Berlin', 'Hamburg', 'Munich', 'Cologne', 'Frankfurt', 'Amsterdam', 'Salzburg',
'Graz', 'Linz', 'Innsbruck')
------
Berlin
Hamburg
Munich
Cologne
Frankfurt
Amsterdam
Salzburg
Graz
Linz
Innsbruck
```

[57]:
```
from collections.abc import Collection

print(isinstance(cities, Collection))

len(cities)
```

```
True
```

[57]: 10

[62]:
```
from collections.abc import Mapping

print(isinstance(cities, Mapping))

dict(cities)
```

```
False
```

```
      ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-62-fd23c552723e> in <module>
         3 print(isinstance(cities, Mapping))
         4
   ----> 5 dict(cities)
```

```
        ValueError: dictionary update sequence element #0 has length 6; 2 is␣
↪required
```

1.1.2 values

[63]:
```
print(population_as_dict)

citizens = population_as_dict.values()
print(type(citizens))

print('------')
print(list(citizens))
```

```
{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664,
'Frankfurt': 753056, 'Amsterdam': 758198, 'Salzburg': 153377, 'Graz': 222326,
'Linz': 204846, 'Innsbruck': 132493}
<class 'dict_values'>
------
[3748148, 1822445, 1471508, 1085664, 753056, 758198, 153377, 222326, 204846,
132493]
```

[64]:
```
from collections.abc import Iterable, Collection, Mapping

print(isinstance(citizens, Iterable))
print(isinstance(citizens, Collection))
print(isinstance(citizens, Mapping))
```

```
True
True
False
```

1.1.3 items

print(population_as_dict)

dict_items = population_as_dict.items() print(type(dict_items))

[65]:
```
from collections.abc import Iterable, Collection, Mapping

print(isinstance(dict_items, Iterable))
print(isinstance(dict_items, Collection))
print(isinstance(dict_items, Mapping))
```

```
        ␣
↪---------------------------------------------------------------------------
```

```
        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-65-babdf46ffcaa> in <module>
          1 from collections.abc import Iterable, Collection, Mapping
          2
    ----> 3 print(isinstance(dict_items, Iterable))
          4 print(isinstance(dict_items, Collection))
          5 print(isinstance(dict_items, Mapping))


        NameError: name 'dict_items' is not defined
```

[66]:
```
print(dict_items)
```

```
      ␣
↪-------------------------------------------------------------------------
        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-66-f24f234ee236> in <module>
    ----> 1 print(dict_items)


        NameError: name 'dict_items' is not defined
```

[ ]:
```
for city, citizens in population_as_dict.items():
    print('{} - {}'.format(city, citizens))
```

1.2 Alternative containers: defaultdict, OrderedDict and Counter

```
__builtin__.dict.__subclasses__()

[collections.OrderedDict,
 collections.defaultdict,
 collections.Counter,
...
]
```

1.2.1 defaultdict

Returns a new dictionary-like object. defaultdict is a subclass of the built-in dict class. [...] The first argument provides the initial value for the default_factory attribute; it defaults to None. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

If the default_factory attribute is None, this raises a KeyError exception with the key as argument.

If default_factory is not None, it is called without arguments to provide a default value for the given key, this value is inserted in the dictionary for the key, and returned.

[ ]:
```
from collections.abc import Iterable, Collection, Mapping
from collections import defaultdict

austrian_population = [
    ('Graz', 222326),
    ('Linz', 204846),
    ('Innsbruck', 132493)
]

d = defaultdict(None)
d.update(austrian_population)

print(type(d))
print(d)

print(isinstance(d, Iterable))
print(isinstance(d, Collection))
print(isinstance(d, Mapping))

d['Paris']
```

[67]:
```
from collections import defaultdict

austrian_population = [
    ('Graz', 222326),
    ('Linz', 204846),
    ('Innsbruck', 132493)
]

d = defaultdict(list)
d.update(austrian_population)

d['Paris']
```

[67]:
```
[]
```

[68]:
```
from collections import defaultdict

austrian_population = [
    ('Graz', 222326),
    ('Linz', 204846),
    ('Innsbruck', 132493)
]

def my_callable():
    return 'Not found'

d = defaultdict(my_callable)
d.update(austrian_population)

d['Paris']
```

[68]:
```
'Not found'
```

1.2.2 OrderedDict

Return an instance of a dict subclass, supporting the usual dict methods. An OrderedDict is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

[69]:
```
from collections import OrderedDict

odict1 = OrderedDict()
odict1['a'] = 'A'
odict1['b'] = 'B'
odict1['c'] = 'C'
odict1['d'] = 'D'
odict1['e'] = 'E'

odict2 = OrderedDict()
odict2['e'] = 'E'
odict2['d'] = 'D'
odict2['c'] = 'C'
odict2['b'] = 'B'
odict2['a'] = 'A'

print(odict1 == odict2)

standard_dict1 = dict(odict1)
standard_dict2 = dict(odict2)
```

```
print(standard_dict1 == standard_dict2)
```

```
False
True
```

[70]:
```
print(odict1)

odict1.move_to_end('c', last=True)
print(odict1)

print('----')

odict1.move_to_end('c', last=False)
print(odict1)
```

```
OrderedDict([('a', 'A'), ('b', 'B'), ('c', 'C'), ('d', 'D'), ('e', 'E')])
OrderedDict([('a', 'A'), ('b', 'B'), ('d', 'D'), ('e', 'E'), ('c', 'C')])
----
OrderedDict([('c', 'C'), ('a', 'A'), ('b', 'B'), ('d', 'D'), ('e', 'E')])
```

1.2.3 Counter

A mapping that holds an integer count for each key. Updating an existing key adds to its count

Keeps track of how many times equivalent values are added.

Along with its functions are used collectively for processing huge amounts of data. It creates a hash-map for the data container invoked with. This is more efficient than processing elements by hand.

[71]:
```
from collections import Counter


dummy_txt = """Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Quisque malesuada nibh et justo blandit, a consectetur sapien ornare.
Fusce a nunc sem. Fusce interdum quam quis magna dapibus, ut dapibus massa␣
 ↪porta.
Nunc facilisis eget purus sit amet placerat. Sed nec sapien vel neque pulvinar␣
 ↪pretium et sed ante.
Quisque mauris nibh, maximus at massa quis, mattis ornare turpis. Nunc accumsan␣
 ↪rutrum nibh,
sed sagittis leo dignissim eget. Phasellus vehicula ligula at vulputate␣
 ↪molestie.
Aenean non leo sed libero hendrerit mattis.
Sed aliquet tincidunt scelerisque. Class aptent taciti sociosqu ad litora␣
 ↪torquent per conubia nostra,
```

```
per inceptos himenaeos. Nam feugiat sem velit, id convallis odio scelerisque id.
 ↪
Ut vitae elementum sem, quis fermentum enim. Sed pharetra enim mauris, eu␣
 ↪vehicula lorem aliquet ut.
Cras pretium id ipsum ac suscipit. Cras blandit erat in nisl venenatis semper.
Nam eget felis mollis, aliquet sem ut, cursus quam. Duis sed lectus ligula.␣
 ↪Praesent condimentum,
nunc vitae molestie mattis, velit magna lobortis erat, in condimentum enim diam␣
 ↪eu ante.
Donec non pellentesque erat. Nunc a hendrerit quam, tempor porta libero.
Mauris commodo ante et diam porttitor, quis euismod tellus scelerisque.
Donec turpis nulla, commodo sed leo in, auctor interdum nulla.
"""

txt_counter = Counter(dummy_txt)
txt_counter
```

[71]:
```
Counter({'L': 1,
         'o': 49,
         'r': 55,
         'e': 127,
         'm': 58,
         ' ': 199,
         'i': 100,
         'p': 27,
         's': 87,
         'u': 81,
         'd': 33,
         'l': 53,
         't': 90,
         'a': 88,
         ',': 19,
         'c': 39,
         'n': 69,
         'g': 11,
         '.': 24,
         '\n': 16,
         'Q': 2,
         'q': 19,
         'b': 11,
         'h': 10,
         'j': 1,
         'F': 2,
         'N': 5,
         'f': 4,
         'S': 3,
         'v': 11,
```

```
'x': 1,
'P': 2,
'A': 1,
'C': 3,
'U': 1,
'D': 3,
'M': 1})
```

[72]:
```
txt_counter.most_common(5)
```

[72]:
```
[(' ', 199), ('e', 127), ('i', 100), ('t', 90), ('a', 88)]
```

1.3 Dictionaries homework

Using data from Eurostat, create a list of tuples representing the **"Self-perceived health by country and sex, age group >16, for people living in cities"** for 2017-2018. Have at least 30 values in your dataset.

The dataset will have the following struncture: > [> (country, year, sex, health_index) >]

Example: > [('France', 2017, M, 12), ...]

Using only comprehensions, create the following dicts: - two dicts that group all data by country for each year > health_index_2017 = {'France': [sex, health_index]} > health_index_2017 = {'France': [sex, health_index]} - one dict that groups all data by year for Germany > germany = {2017: [sex, health_index]} - one dict that grups all data by country and year, by using year in the key together with the country name > health_index = {'France_2017': [year, sex, health_index]} - starting from the previous **health_index** dict, display only the data where the health_index > 5 - starting from the previous **health_index** dict, display only the data where the health_index > 5 and sex is 'F' - starting from the previous **health_index** dict, create a for loop to print the health_index

2 Sets

A set is an unordered collection of items where each element is unique (no duplicates) and must be immutable.

The set itself, however, is mutable

2.0.1 Creating and updating a set

[82]:
```
set1 = {1, 2, 3, 4, 3, 2, 5, 6, 3}
print(set1)
```

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

[83]:
```
set2 = {1.0, "Hello", (1, 2, 3)}
print(set2)
```

```
{'Hello', 1.0, (1, 2, 3)}
```

[84]:
```
empty_set = {}
print(type(empty_set))
```

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

[85]:
```
empty_set2 = set()
print(type(empty_set2))
```

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

[86]:
```
list_1 = [1, 2, 3, 2, 5, 2]
set_from_list = set(list_1)
print(set_from_list)
```

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

[87]:
```
set_from_list.add(2)
print(set_from_list)

set_from_list.add(9)
print(set_from_list)
```

```
{1, 2, 3, 5}
{1, 2, 3, 5, 9}
```

[88]:
```
set_from_list.update([1, 2, 8, 7, 3])
print(set_from_list)
```

```
{1, 2, 3, 5, 7, 8, 9}
```

[90]:
```
set_from_list.remove(3)
print(set_from_list)

set_from_list.remove(33)
```

```
{1, 5, 7, 8, 9}
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)
```

```
        <ipython-input-90-36bf0f442bbe> in <module>
          2 print(set_from_list)
          3
    ----> 4 set_from_list.remove(33)


        KeyError: 33
```

[89]:
```
set_from_list.discard(2)
print(set_from_list)

set_from_list.discard(33)
print(set_from_list)
```

```
{1, 3, 5, 7, 8, 9}
{1, 3, 5, 7, 8, 9}
```

[92]:
```
# since sets are not ordered, pop will remove a random element
set_from_list.pop()
print(set_from_list)
```

```
{5, 7, 8, 9}
```

[93]:
```
for letter in set("apple"):
    print(letter)
```

```
p
l
e
a
```

2.1 Sets homework

Create two sets with 10 numbers each (some of the numbers should be present in both sets). With these two sets, exemplify the following basic sets operations: union, intersection, difference and symetric difference.

Resources: https://docs.python.org/3.6/library/stdtypes.html#set-types-set-frozenset