Lesson Overview

Long time, no see! Welcome back to Python Academy.

So far you know:

- how to use conditions
- how to operate with basics data types.

It´s time for learn something new. In lesson 3, we will cover:

- **dictionaries**,
- **sets**.

Dictionary is a new data type and is pretty much like phonebook, where you add various values (phone number) to users (contact name). When you select specific user (contact name), dictionary will give you information with added value (phone number). Easy peasy.

And sets? Sets are also a new data type and is similar to sets you may remember from math at primary school. It is simply a collection of unique objets, but more on that in the lesson ;)

REVIEW EXERCISES

Find bugs

100% z Lekce 3

```
2
3  if isupper(some_str):
4      print('All letters in your input are uppercase')
5  else if islower(some_str):
6      print('All letters in your input are lowercase')
7  else
8      print('There are both uppercase and lowercase letters')
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Create a script `split_nums.py` that will:

- ask the user for a number
- split the given number in halves (e.g. 123456 -> split to 123 and 456, 12345 -> 12 and 345)
- convert both halves into an integer
- if both halves are an even integer, print: `'Success'` - e.g. 12 and 34
- if only the first part is even, print: `'First'` - e.g. 12 and 345
- if the second part is even, print: `'Second'` - e.g. 123 and 456
- if neither of the numbers is even print: `'Neither'` - 123 and 455
- if nothing has been entered (the user just hit Enter), print: `'No input provided'`

Example of running the script:

```
/Users/PythonBeginner/Lesson1$ python split_nums.py
Please, give me a number: 35
Neither
```

```
/Users/PythonBeginner/Lesson1$ python split_nums.py
Please, give me a number:
No input provided
```

Online Python Editor

You can create the script in you computer or here in our editor.

```
1
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Password check 1

Your task is to create a script called `check_start.py` that will:

- ask the user for the secret password
- if the password given, starts with any of the lowercase 'a','e','f','q','z', print to the terminal `'Welcome!'`

```
/Users/PythonBeginner/Lesson1$ python check_start.py
Please enter the password: abcd
Welcome!
```

```
/Users/PythonBeginner/Lesson1$ python check_start.py
Please enter the password: black hand
The input does not match
```

Online Python Editor

```
1
```

spustit kód

Click to see our solution

ONSITE PROJECT

Create a dictionary

In the few following tasks, we will be working with the script `films.py` .

```
name = 'Shawshank Redemption'
rating = 87
year = 1994
director = 'Frank Darabont'
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Add a new category

Inside your script `films.py`, write the code, that will add 2 new categories to the dictionary `film`.

- The first new category should called `'starring'` and we should associate a with it a list containing strings `'Tim Robbins'`, `'Morgan Freeman'`.
- the other category should be called `'budget'` and the value associated with it should be 200

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

In the previous step, we have made mistake by adding a new category `'budget'` and therefore we should remove it. Please remove the whole key - value pair under the `'budget'` key.

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Nest a dict within another dict

Do the following:

1. Create a new empty dictionary called `films`.
2. Add the dictionary you have stored inside the variable `film` to the newly created `films` dictionary under the key 'DRAMA'.
3. Print the content of the `films` dictionary:

Example of program execution:

```
~/PythonBeginner/Lesson2 $ python films.py
{'DRAMA': {'year': 1994, 'starring': ['Tim Robbins', 'Morgan Freeman'],
'rating': 87, 'director': 'Frank Darabont', 'name': 'Shawshank
Redemption'}}
```

Note that the order in which the categories are printed does not have to be necessarily the same as the one depicted above.

Click to see our solution

Dictionary querying

Now we have a very (very) small film database. We can now offer a user to ask for information regarding the films we store in there.

The user should be able to request information according to film genre. Try to write the code that will make the program as demonstrated below.

Example of running the program:

```
~/PythonBeginner/Lesson2 $ python films.py
We can currently offer:
['DRAMA']
What genre are you interested in? DRAMA
HERE IT GOES:
{'year': 1994, 'starring': ['Tim Robbins', 'Morgan Freeman'], 'rating':
87, 'director': 'Frank Darabont', 'name': 'Shawshank Redemption'}
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Our last task is to write a command, that will empty the whole dictionary `films` and print the result to the terminal after printing the message `DATABASE HAS BEEN ERASED:` .

Example of running the program:

```
~/PythonBeginner/Lesson2 $ python films.py
We can currently offer:
['DRAMA']
What genre are you interested in? DRAMA
HERE IT GOES:
{'year': 1994, 'starring': ['Tim Robbins', 'Morgan Freeman'], 'rating':
87, 'director': 'Frank Darabont', 'name': 'Shawshank Redemption'}
DATABASE HAS BEEN ERASED:
{}
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Dictionary Code Solutions

So this is it for playing with dictionary. Check out the whole solution, before we move to sets.

Click to see our solution

Now, let's play with `Sets` for a bit. In the following exercises, we will be working with the following 2 strings:

- str1 = 'New York'
- str2 = 'Yorkshire'

Write a short script (we can call it `common.py`) that will print all the letters that these two strings have in common

Example of running the script:

```
~/PythonBeginner/Lesson2 $ python common.py
['Y', 'e', 'k', 'r', 'o']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Unique items

In the following exercise, we will be working with the following 2 strings:

- str1 = 'New York'
- str2 = 'Yorkshire'

Write a short script (we can call it `unique.py`) that will print:

Example of running the script:

```
~/PythonBeginner/Lesson2 $ python unique.py
Unique to str1: [' ', 'w', 'N']
Unique to str2: ['h', 's', 'i']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Symmetric difference

In the following exercise, we will be working with the following 2 strings:

- str1 = 'New York'
- str2 = 'Yorkshire'

Write a short script (we can call it `unique_together.py`) that will print letters that are in str1 or str2 but not in both

Example of running the script:

```
~/PythonBeginner/Lesson2 $ python unique_together.py
[' ', 'w', 's', 'i', 'h', 'N']
```

Code Solution

Click to see our solution

Union

In the following exercise, we will be working with the following 2 strings:

- str1 = 'New York'
- str2 = 'Yorkshire'

Write a short script (we can call it `both.py`) that will print the list of unique letters (symbols), that can be found in str1 or str2.

Example of running the script:

```
~/PythonBeginner/Lesson2 $ python both.py
[' ', 'k', 'e', 'r', 'w', 's', 'i', 'o', 'h', 'Y', 'N']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

School subject attendance

Here we have the information:

```
classes = {'Biology' :
['Adam','Chelsea','Marcus','Oliver','Alex','Sandra','Ann'],
            'Math' :
['Marcus','Alex','Glenn','Samuel','Clara','Chelsea'],
            'PE'  : ['Adam','Tyler', 'Alex','Clara'],
            'Social Sciences':
['Abraham','Marcus','Alex','Glenn','Clara'],
            'Chemistry' : ['Alfred',
'Curt','Oliver','Alex','Tyler','Ann']}
```

Example of running the code:

```
~/PythonBeginner/Lesson2 $ python class_stats.py
Students inscribed into all the subjects: {'Alex'}
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

DICTIONARY

Creating dictionary

There are quite a few ways how to create a dictionary - we can use:

1. empty dictionary constructor:

```
>>> d1 = dict()
>>> d1
{}
```

2. empty curly braces:

```
>>> d2 = {}
>>> d2
{}
```

3. filled curly braces with key: value pairs:

```
>>> d3 = {'First_Name': 'John','Last_Name': 'Smith','Age':56}
>>> d3
{'First_Name': 'John','Last_Name': 'Smith','Age':56}
```

4. constructor filled with keyword arguments:

```
{ Age : 56,  First_Name :  John ,  Last_Name :  Smith }
```

Keys

Dictionary keys **can be only objects of immutable** data type - number, string, tuple, bytes.

```
>>> my_dict = {'name' : 'John', 'age': 32}
>>> my_dict['name']
'John'
```

It's clear then that we **cannot use any mutable** data type as a key:

```
>>> my_dict = {['a','b','c'] : 'name'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

List `['a','b','c']` is mutable - it means we can add and remove items from it. Dictionary keys, however, have to be constant. We cannot permit a key, which serves as a unique identifier, to be able to be changed. It is like allowing to change your internet banking credentials. The IB system would not be able to identify you anymore.

Fun Fact: While reading python documentation, you may encounter a word **hashable** or unhashable. Hashable is a synonym to immutable and unhashable is a synonym to mutable - object that can be changed.

Why not integers as keys?

Using integers as keys is unnecessary as in that case we would better **use one of the sequence data types instead**. Items of sequences are by default indexed by (and moreover ordered by) integers.

```
>>> my_dict[1]
'age'
```

Integers in list:

```
>>> data = ['name', 'age']
>>> data[1]
'age'
```

Values

Each key can has just one associated value. However, unlike keys, value **can be a mutable** as well as immutable collection of multiple objects if needed (list, tuple, dictionary, set, but also numbers, strings, bytes, bytearray).

```
>>> my_dict = {'volume' : [3,4,7],'surface': [3,4], 'lengths': [3,4,7]}
>>> my_volume = my_dict['volume'][0] * my_dict['volume'][1] *
my_dict['volume'][2]
>>> my_volume
84
```

Above we repeat the list object [3,4,7] twice, and that is absolutely permitted for dictionary values. Any given value can be stored under any number of keys - meaning, **values can repeat in a dictionary, but keys not**. Keys have to be unique identifiers of every key:value pair.

Another **example**, where values can repeat themselves is a attendance record stating which students were present at last week's lesson:

```
>>> lesson1_presence = {'John':True, 'Bob': True, 'Kate' : False, 'Fred'
: True}
```

To insert a new value into a dictionary we assign the value to a variable referring our dictionary. We have to use the square brackets specifying the key we want to associate the value with:

```
>>> my_dict = {}
>>> my_dict['name'] = 'John'
>>> my_dict['surname'] ='Smith'
>>> my_dict
{'name': 'John', 'surname': 'Smith'}
```

Order?

It is important to note that in dictionaries **key-value pairs are not ordered** at least up to Python 3.5 From Python 3.6 dictionary values are already ordered.

Python 3.5

If for example we created a new dictionary as follows in Python 3.5, the key order will be changing:

```
>>> employee = {'id': 'X5342', 'name': 'Clark', 'surname' : 'Kent',
'age': 34}
>>> employee
{'id': 'X5342', 'name': 'Clark', 'age': 34, 'surname': 'Kent'}
```

That way we cannot be sure what key:value pair will be removed when using `dict.popitem()` method:

```
>>> employee.popitem()
('id', 'X5342')
>>> employee.popitem()
```

```
( age , 34)
>>> employee.popitem()
('surname', 'Kent')
```

Now we turn off our Python 3.5 terminal and run it again. Then we create the same dictionary as before. However, the order of items is popped out in different order:

```
>> employee = {'id': 'X5342', 'name': 'Clark', 'surname' : 'Kent', 'age':
34}
>>> employee.popitem()
('surname', 'Kent')
>>> employee.popitem()
('id', 'X5342')
>>> employee.popitem()
('name', 'Clark')
>>> employee.popitem()
('age', 34)
```

Python 3.6

On the other hand, using Python 3.6 and higher, the order of items will be the same each time we recreate the dict and destroy it.

```
>>> employee = {'id': 'X5342', 'name': 'Clark', 'surname' : 'Kent',
'age': 34}
>>> employee
{'id': 'X5342', 'name': 'Clark', 'surname': 'Kent', 'age': 34}
>>> employee.popitem()
('age', 34)
>>> employee.popitem()
('surname', 'Kent')
>>> employee.popitem()
('name', 'Clark')
>>> employee.popitem()
('id', 'X5342')
```

As dictionary is a collection of key:value pairs, the `dict()` constructor arguments have to provide a **pairwise structure** as well. This prerequisite is met, for example, by passing two-dimensional tuple. Dict constructor expects 1 argument to be passed in. Therefore a single collection of data has to be passed in.

Now we will answer to the question: What can and cannot be entered into `dict()` constructor?

CAN

1. In the example below, we have entered 1 tuple containing 3 other two-dimensional tuples

```
>>> dict(((1,2),(3,4),(5,6)))
{1: 2, 3: 4, 5:6}
```

2. Here we are passing a tuple containing only 1 nested tuple (note the comma after the tuple `(1,2)`):

```
>>> dict(((1,2),))
{1: 2}
>>>
```

CANNOT

1. Each tuple serves as a key - value pair. If there would be more than 2 items in each tuple, we would get an error:

```
>>> dict((('1','2','3'),(1,2,3)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: dictionary update sequence element #0 has length 3; 2 is
required
```

2. Also one tuple with even number of items would not work:

```
>>> dict((1,2,3,4))
```

```
TypeError: cannot convert dictionary update sequence element #0 to a
sequence
```

3. The `dict()` constructor expects **one object as input** and therefore passing multiple tuples inside won't work:

```
>>> dict(('1','2'),(1,2))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: dict expected at most 1 arguments, got 2
```

4. The first item in the input tuple is made a key and the second is stored in the value part of the pair. Therefore the first item has to be of immutable data type. For example, list `[1]` is **not immutable** - therefore an error is raised:

```
>>> dict([[[1],2],[3,4]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Nesting

PYTHON ACADEMY / 3. DICTIONARIES & SETS / DICTIONARY / NESTING

We can nest dictionaries inside dictionaries as values (not keys). This allows for modelling of more complex categorizations. In the example below, we can see dictionary `{'Job Title': 'System Admin', 'Level' : 3}` nested under the key `'Job'`. Or also the `'Address'` key refers to a another nested dictionary:

```
my_db = {'Name': 'John Smith',
         'Age': 34,
         'Address': {'Street': 'Main',
                     'Street #': 241,
                     'City': 'Boston',
```

100% z Lekce 3

```
        'Job' : { 'Job Title' : 'System Admin',
                  'Level' : 3}
    }
```

If we had a database of employees stored in this way in a dictionary and we wanted to know, what is this employee's home country, we would be very happy to have the address key to contain another dictionary with more specific categorization:

```
>>> my_db['Address']['Country']
'Venezuela'
```

Code Task

1. Try to access the value stored under the `'Level'` key, which is stored under the `'Job'` key inside the `my_db` dictionary.
2. Try to concatenate the whole address into one string. City and Country should be separated by comma.

The result could look like this:

```
3
'241 Main Street, Boston, Venezuela'
```

```
# Database
my_db = {   'Name': 'John Smith',
            'Age': 34,
            'Address': {'Street': 'Main',
                'Street #': 241,
                'City': 'Boston',
                'Country': 'Venezuela'},

            'Job': {'Job Title': 'System Admin',
                    'Level' : 3}
}

# 1

# 2

# Print
```

100% z Lekce 3

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Accessing values

PYTHON ACADEMY / 3. DICTIONARIES & SETS / DICTIONARY / ACCESSING VALUES

Dictionary methods, as we will see, many times duplicate the functionality, we have already seen performed with the square brackets operation as accessing, updating or removing values. However, the main benefit in using them is that they handle situations when keys do not exists and they do not raise errors.

In the following examples we will use the variable `my_db` that contains the following information:

```
>>> my_db
{'Name': 'John Smith',
 'Age': 34,
 'Address': {'Street': 'Main',
             'Street #': 241,
```

```
    'Job' : { 'Job Title' : 'System Admin',
              'Level' : 3}
}
```

Method .get()

One of the methods that handle such situation is the method `.get()`.

Syntax: `my_db.get(key[,default_value=None])`

Arguments accepted by the method are key and optional default value, for case if key is not present in the dictionary. That way the error is not raised if the key does not exist in the dictionary.

```
>>> my_db.get('Address')
{'City': 'Boston',
 'Country': 'Venezuela',
 'Street': 'Main',
 'Street #': 241
}
```

Why do we need a special method for something that already exists as `my_db['Address']`? The reason is, that the get() method allows us to avoid errors in our programs - if the key is not found, the default value specified as a second input argument is returned:

```
>>> my_db.get('Salary',0)
0
```

Code Task

- Try to use the `get()` method to retrieve value stored under they key `'Birth'`. If such key does not exist, value '0.0.0000' should be returned.
- Try to retrieve value stored under the key `'Age'`, if no such key found, value 0 should be returned.

```
1
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Updating a dictionary

keyword arguments.

Syntax: `my_db.update([other])`

We will continue to work with a variable `my_db` that contains the following data:

```
>>> my_db
{'Name': 'John Smith',
 'Age': 34,
 'Address': {'Street': 'Main',
             'Street #': 241,
             'City': 'Boston',
             'Country': 'Venezuela'},
 'Job': { 'Job Title': 'System Admin',
          'Level' : 3}
}
```

Update method

First, let's updated the dictionary inside `my_db` variable with key 'Performance' by passing a dictionary into `update()`

```
>>> my_db.update({'Performance': {'Q1': 1, 'Q2':1, 'Q3':2, 'Q4':1}})
>>> my_db
{'Name': 'John Smith',
 'Age': 34,
 'Address': {'Street': 'Main',
             'Street #': 241,
             'City': 'Boston',
             'Country': 'Venezuela'},
 'Job': {'Job Title': 'System Admin','Level' : 3},
 'Performance' : {'Q1': 1, 'Q2':1, 'Q3':2, 'Q4':1}
}
```

Update keyword arguments

```
>>> my_db
{'Name': 'John E. Smith',
 'Age': 34,
 'Address': {'Street': 'Main',
             'Street #': 241,
             'City': 'Boston',
             'Country': 'Venezuela'},
 'Job': {'Job Title': 'System Admin',
         'Level' : 3},
 'Performance' : {'Q1': 1, 'Q2':1, 'Q3':2, 'Q4':1}
}
```

In both cases the operation has been handled without any error or complaint and that is what we aimed for.

Code Task

Try to add the manager name to `my_db` dictionary under the key `'Manager'` and value `'Samuel, Hunt'` using the `update()` method.

```
1
```

100% z Lekce 3

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Removing items

PYTHON ACADEMY / 3. DICTIONARIES & SETS / DICTIONARY / REMOVING ITEMS

To remove key-value pairs (items) from a dictionary, we can use the `del` keyword as follows: `del my_dict[key]`

In the following section we will keep working with dictionary `my_db` :

```
>>> my_db
{'Name': 'John Smith',
 'Age': 34,
 'Address': {'Street': 'Main',
             'Street #': 241,
             'City': 'Boston',
             'Country': 'Venezuela'},
 'Job': {'Job Title': 'System Admin','Level' : 3},
 'Performance' : {'Q1': 1, 'Q2':1, 'Q3':2, 'Q4':1}
```

the following command:

```
>>> del my_db['Performance']
```

Now if we print the content of `my_db` variable, we can see `Performance` key no more:

```
>>> my_db
{'Name': 'John Smith',
 'Age': 34,
 'Address': {'Street': 'Main',
             'Street #': 241,
             'City': 'Boston',
             'Country': 'Venezuela'},
 'Job': {'Job Title': 'System Admin','Level' : 3}
}
```

Removing Data

We have some additional methods for removing data from dictionary.

Method .pop()

The `.pop()` method removes the specified key and returns the associated value. If the key does not exist, we can specify the default value that should be returned in such case.

The syntax is `my_db.pop(key[,default_value])` and an example would be:

```
>>> my_db.pop('FTE',1)
1
```

Let's add the 'FTE' category into our dictionary and then immediately pop it - if we try to retrieve it, we will get an error:

```
>>> my_db['FTE'] = 0.5
>>> my_db.pop('FTE',1)
0.5
>>> my_db['FTE']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'FTE'
```

Method .popitem()

This method removes some (arbitrary) key-value pair from the dictionary and returns the pair as a tuple. The specific pair cannot be specified, therefore this method is used for continual destruction of a dictionary.

The syntax is `my_db.popitem()` and an example:

```
>>> my_db.popitem()
('Address', {'Country': 'Venezuela', 'City': 'Boston', 'Street': 'Main',
'Street #': 241})
```

If there are no more items (key-value pairs) inside the dict, KeyError is raised.

```
>>> my_db = {}
>>> my_db.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

The `.popitem()` method can be used, when we gradually want to empty the whole dictionary and every time we want to take one key:value pair for processing. This will be more usefully demonstrated with so called loops.

The syntax is `my_db.clear()` and an example:

```
>>> my_db.clear()
>>> my_db
{}
```

Copying dictionary

To create a copy (a new object) of a dictionary, we have to use its `.copy()` method. It returns so called shallow copy of the original dictionary. Shallow copy means, that the nested mutable collections (any dictionaries or lists or sets) are still the same objects and so if we will change them in any way, the original instance of the dictionary will also reflect these changes.

In our example below we are working with the dictionary `my_db` :

```
my_db = { 'Name': 'John E. Smith',
          'Age': 34,
          'Address': {'Street': 'Main',
                      'Street #': 241,
                      'City': 'Boston',
                      'Country': 'Venezuela'},

          'Job': {'Job Title': 'System Admin','Level' : 3},
          'Performance' : {'Q1': 1, 'Q2':1, 'Q3':2, 'Q4':1}
        }
```

Now we copy `my_db` to `new_db` :

```
>>> new_db = my_db.copy()
>>> new_db
{'Name': 'John E. Smith',
```

```
                'Street # ': 241,
               'City': 'Boston',
               'Country': 'Venezuela'},
 'Job': {'Job Title': 'System Admin','Level' : 3},
 'Performance' : {'Q1': 1, 'Q2':1, 'Q3':2, 'Q4':1}
}
```

Watch out for mutable values

Keys `'Performance'`, `'Job'` and `'Address'` refer to mutable data types. Therefore if we change them in `new_db`, **the change will be reflected in** `my_db` as well, because `new_db` is just a shallow copy of `my_db`:

```
>>> del new_db['Performance']['Q1']
>>> new_db
{'Name': 'John E. Smith',
 'Age': 34,
 'Address': {'Street': 'Main',
             'Street #': 241,
             'City': 'Boston',
             'Country': 'Venezuela'},
 'Job': {'Job Title': 'System Admin',
         'Level' : 3},
 'Performance' : {'Q2':1, 'Q3':2, 'Q4':1}
}
>>> my_db
{'Name': 'John E. Smith',
 'Age': 34,
 'Address': {'Street': 'Main',
             'Street #': 241,
             'City': 'Boston',
             'Country': 'Venezuela'},
 'Job': {'Job Title': 'System Admin',
         'Level' : 3},
 'Performance' : {'Q2':1, 'Q3':2, 'Q4':1}
```

Copying inner dictionary

We could copy also one of the nested dictionaries:

```
>>> address = my_db['Address'].copy()
>>> address
{'Street': 'Main',
 'Street #': 241,
 'City': 'Boston',
 'Country': 'Venezuela'
}
```

However, if we were to delete an item from that dictionary, the original dictionary, when the `'Address'` was nested would remain **the same**.

```
>>> del address['Street']
>>> address
{'Street #': 241,
 'City': 'Boston',
 'Country': 'Venezuela'
}
>>> my_db
{'Name': 'John E. Smith',
 'Age': 34,
 'Address': {'Street': 'Main',
             'Street #': 241,
             'City': 'Boston',
             'Country': 'Venezuela'},
 'Job': {'Job Title': 'System Admin','Level' : 3},
 'Performance' : {'Q1': 1, 'Q2':1, 'Q3':2, 'Q4':1}
}
```

Dictionary Views

Dictionary views is a new concept in Python 3 compared to Python 2. Dictionary views are three types of objects, that are returned by the following three dictionary methods:

- `.keys()` - returns objects of **type dict_keys** that refer to all top level keys in dictionary
- `.values()` - returns objects of **type dict_values** that refer to all top level values in dictionary
- `.items()` - returns objects of **type dict_items** that refer to all tuples of top level key-value pairs in dictionary

In order to see the contents of keys, values and items objects, we need to **convert them into list or a tuple**

Bellow, you'll have a task using the following dictionary:

```
my_db = {   'Name': 'John Smith',
            'Age': 34,
            'Address': {'Street': 'Main',
                'Street #': 241,
                'City': 'Boston',
                'Country': 'Venezuela'},

            'Job': {'Job Title': 'System Admin',
                    'Level' : 3}
        }
```

Why to use views?

We will use the dictionary views when working with **for loop**, but will get to the in the future lesson.

Code Task

1. get the list of the keys from `my_db`
2. get the list of the values from `my_db`

Bonus

Try to get **sorted** version of the values.

```
my_db = {   'Name': 'John Smith',
            'Age': 34,
            'Address': {'Street': 'Main',
                'Street #': 241,
                'City': 'Boston',
                'Country': 'Venezuela'},

            'Job': {'Job Title': 'System Admin',
                    'Level' : 3}
        }

# 1.

# 2.

# 3.
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

DICTIONARY +

Creating Dictionary

Alternatively, we could use the method `.fromkeys()`. It returns a new dictionary with keys taken from the sequence argument and values set to value specified as optional argument. All the keys will have the same value assigned.

The method can be used **only directly on dict class** and **not on a specific dictionary object**. Therefore this method will be always written as `dict.fromkeys()`.

The syntax is `dict.fromkeys(sequence[,value])` and an example:

```
>>> d = dict.fromkeys(('Account1','Account2','Account3'),0)
>>> d
{Account1':0,'Account2':0,'Account3':0}
```

we will use it more.

Using `zip()` function on 2 iterables, all enclosed in dict constructor:

```
>>> dict(zip(('Name','Age'),('John',45)))
{'Age': 45, 'Name': 'John'}
```

```
>>> dict(zip(range(10),range(10)))
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
```

Zip function connects together into key:value pairs objects at the same index in both sequeces - that way we get a tuple of 2-item tuples from which `dict()` creates a dictionary.

Don't worry about this function too much though. This is only a demonstration of other interesting ways of working with dictionary :)

SETS

Introduction

Definition of set is unordered collections of immutable and unique objects. Important defining feature is the uniqueness of each item in a set. Each object (item) appears only once in a set no matter how many times it has been added.

There are **2 types** of set data in Python - **set & frozenset**. The difference among the two is that set is mutable collection whereas frozenset is immutable. Otherwise the essence of both is the same.

Use

As sets are not ordered and do not contain any keys, therefore we **cannot use**:

- indexing,
- slicing,
- but neither repetition,
- or concatenation.

Therefore the **main use** of sets is:

- membership testing,
- assuring the uniqueness of each item in a collection.

Syntax

Items in set are enclosed in **curly braces** - similarly to a dictionary. However, there are **no key:value pairs**, what distinguishes dictionaries from sets. Dictionary keys have to be also **unique** as the set items and maybe therefore the similarity with curly braces.

Type	Syntax

Frozenset	`frozenset({'a', 'b', 'c'})`

Creating set

Empty sets can be created by using the **set constructor** - `set()`

```
>>> my_set = set()
>>> type(my_set)
<class 'set'>
```

```
>>> my_set = frozenset()
>>> type(my_set)
<class 'frozenset'>
```

However, we **cannot use curly braces** `{}` to create an empty set. We would be creating a new dictionary:

```
>>> data = {}
>>> type(data)
<class 'dict'>
```

Non-empty sets (not frozensets) can be created by placing a comma-separated elements inside the curly braces, for example:

```
>>> my_set = {'John', 'Rob'}
>>> type(my_set)
<class 'set'>
```

Example of use

```
>>> set('Popocatepetl')
{'P', 'a', 'c', 'e', 'l', 'o', 'p', 't'}
```

Or maybe we have a sequence of integers mixed with strings, that also works for sets:

```
>>> s = set([1,2,4,2,5,6,4,3,3,'a'])
>>> s
{1, 2, 3, 4, 5, 6, 'a'}
>>> type(s)
<class 'set'>
```

Mutable Set Operations

If we want to keep and change a collection of unique elements, we will surely need to know, how to add and remove items from it. We know we need a set to keep a collection of unique elements.

By using sets, we do not have to care, whether currently added element is already present in the collection, because set will always keep only one instance of that item.

Adding items to a set

To add a new element to a set we have to use `add()` method: `set1.add(item)`

Example where the item being added is not present yet in the set

```
>>> names = {'Marcus', 'Alex'}
>>> names.add('Oliver')
>>> names
{'Marcus', 'Oliver', 'Alex'}
```

Example where the item being added is already present in the set:

```
>>> names
{'Marcus', 'Oliver', 'Alex'}
```

Set takes care we do not have duplicates in it.

Removing items from a set

To remove an element from a set we have to use `discard()` method: `set1.discard(item)`

```
>>> names = {'Marcus', 'Oliver', 'Alex'}
>>> names.discard('Oliver')
>>> names
{'Marcus', 'Alex'}
```

Common collection operations

PYTHON ACADEMY / 3. DICTIONARIES & SETS / SETS / COMMON COLLECTION OPERATIONS

Among the most common operations with sets are:

1. Membership testing - with the use of operator `in` :

```
>>> 5 in {1,6,13,54,21,654,10,5}
True
```

2. Checking length - with the use of `len()` function to determine the number of unique items contained:

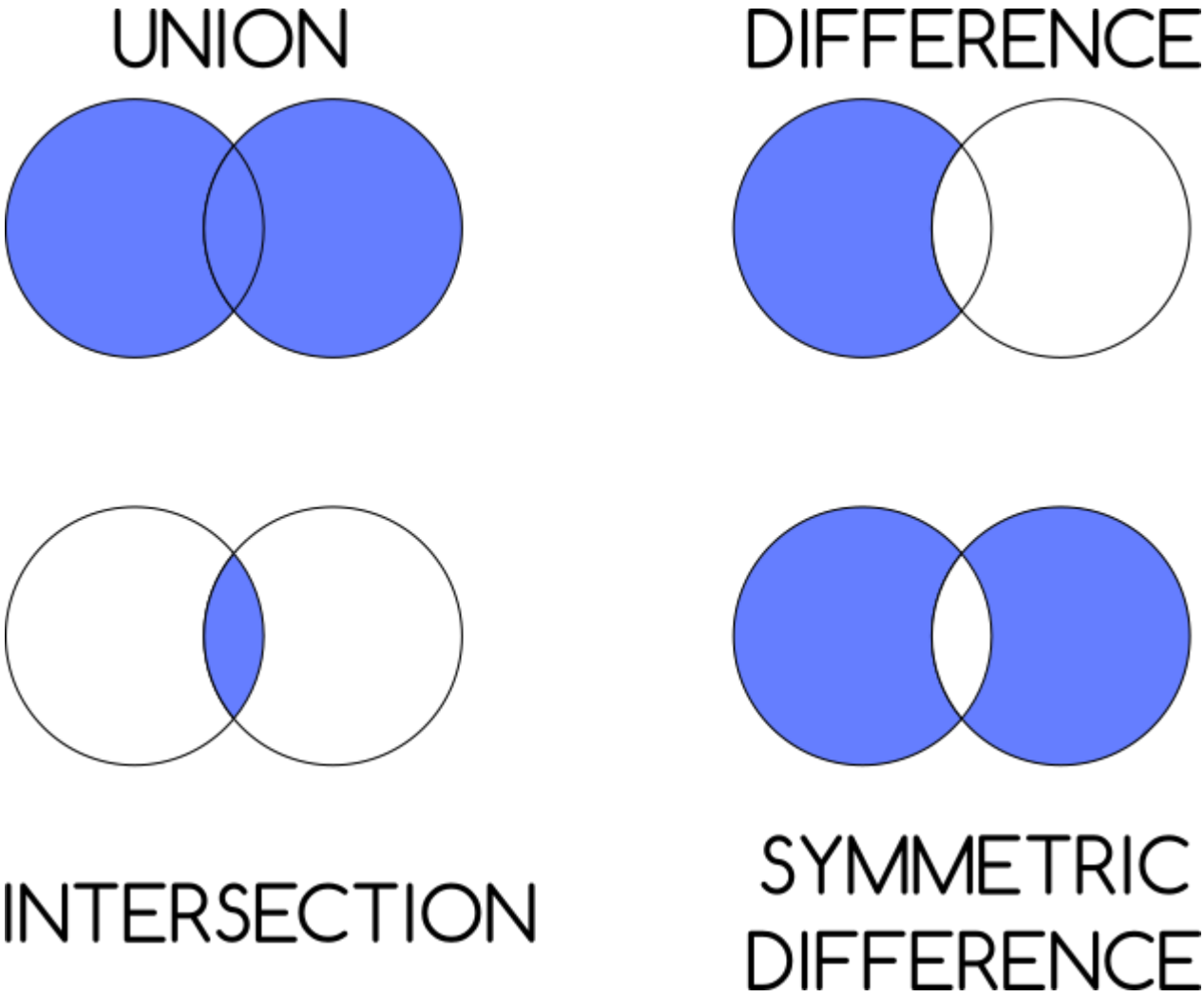
```
>>> len({5,1,5,6,7})
4
```

Set Operations

Sets can be combined in a number of different ways to produce another set. We normally want to find out, what items:

- are in all sets, we combine in the operation (**union**)
- what items are in one but not in the other set (**difference**)
- what items are not shared by the combined sets (**symmetric difference**)
- what items are shared by the sets being combined (**intersection**)

In the following few sections we will learn, how to perform these operations in Python.



Set Operations - Union

The union operation **unites** the elements of all the sets included in the union operation. It returns a new set. We use the `|` operator to perform union on sets: `set1 | set2 | set3 | ...`

```
>>> set('Hello') | set('Yellow') | set ('Fellow')
{'l', 'e', 'F', 'H', 'w', 'o', 'Y'}
```

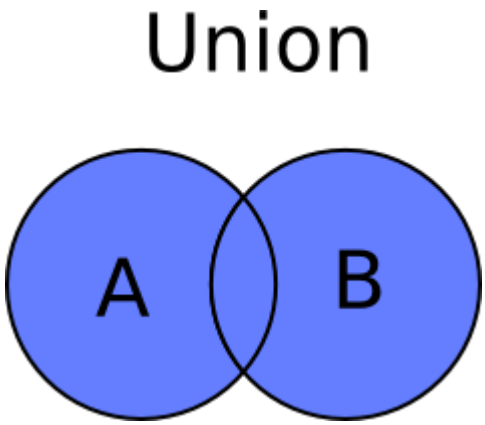
We **cannot use** the union operator with other data type than sets:

```
>>> 'Hello' | 'Yellow' | 'Fellow'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for |: 'str' and 'str'
```

To execute union operation, we can also use a **method**: `set1.union(others)`

Example:

```
>>> set('Hello').union(set('Yellow'),set ('Fellow'))
{'l', 'e', 'F', 'H', 'w', 'o', 'Y'}
```



Set Operations - Difference

set3 ...

```
>>> set('Hello') - set('Yellow') - set('Fellow')
{'H'}
```

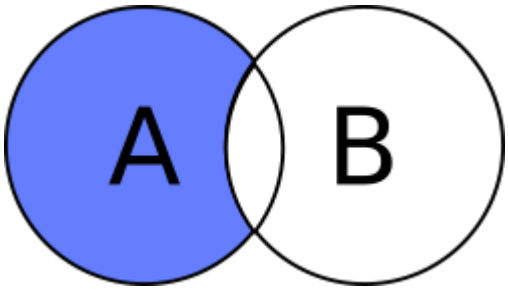
We cannot use the **difference** operator with other data type than sets:

```
>>> 'Hello' -'Yellow'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

To execute difference operation, we can also use a **method**: `set1.difference(others)`

Example:

```
>>> set('Hello').difference(set('Yellow'),set ('Fellow'))
{'H'}
```



Set Operations - Symmetric Difference

PYTHON ACADEMY / 3. DICTIONARIES & SETS / SETS / SET OPERATIONS - SYMMETRIC DIFFERENCE

The **symmetric difference** operation produces a new set with elements in either set but not in both. We use the `^` operator to perform symmetric difference on sets: `set1 ^ set2`

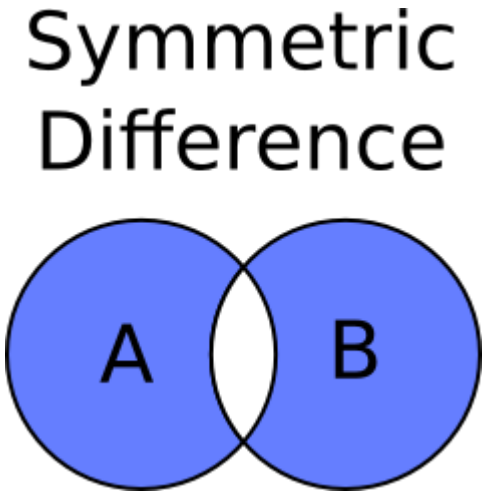
```
>>> set('Hello') ^ set('Yellow')
{'H', 'w', 'Y'}
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ^: 'str' and 'str'
```

To execute symmetric difference operation, we can also use a **method**: `set1.symetric_difference(others)`

Example:

```
>>> set('Hello').symmetric_difference(set('Yellow'))
{'H', 'w', 'Y'}
```



Set Operations - Intersection

PYTHON ACADEMY / 3. DICTIONARIES & SETS / SETS / SET OPERATIONS - INTERSECTION

The intersection operation produces a new set containing only **elements that are shared** by all sets implicated in the operation. We use the `&` operator to perform intersection on sets: `set1 & set2 & set3 & ...`

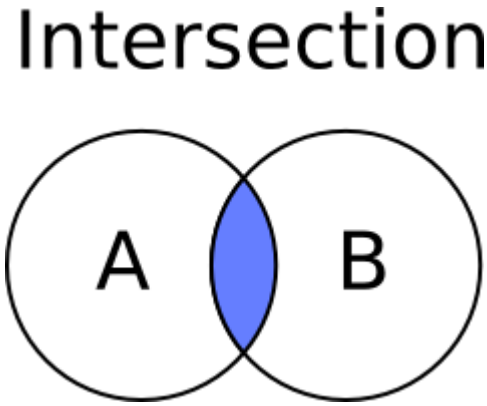
```
>>> set('Hello') & set('Yellow') & set('Fellow')
{'e', 'o', 'l'}
```

```
>>> "Hello" & "Yellow" & "Fellow"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for &: 'str' and 'str
```

To execute intersection operation, we can also use a **method**: `set1.intersection(others)`

Example:

```
>>> set('Hello').intersection(set('Yellow'),set ('Fellow'))
{'e', 'o', 'l'}
```



Set Operations - Subset

An extra set operation is subset. This is when we test whether every element in **set1 is in the set2**. For this we can use less than `<` and equal to `=` operators:

```
>>> set('Hello') <= set('Yellow H')
True
```

Subset operation tests the relation in **only one way** - whether all elements of set1 are included in set2 - but this does not imply that all elements of set2 are in set 1

```
>>> set('Hello') >= set('Yellow H')
```

method `issubset()` : `set1.issubset(set2)`

```
>>> set('Hello').issubset(set('Yellow H'))
True
```

If we wanted to get a **proper subset** and make sure, that all elements of set1 are present in set2, but not all elements of set2 are present in set1, we can omit the equal sign `=` .

Set Operations - Disjoint

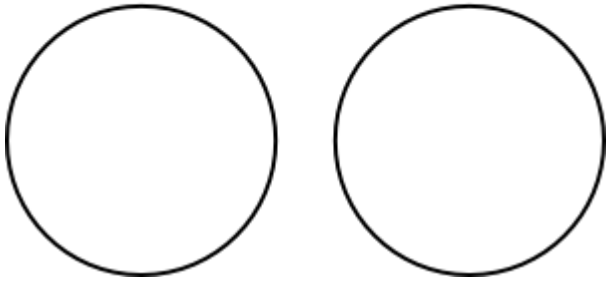
Lastly, to find out, whether two sets have **no elements in common**, we can use the intersection operator. If the result of intersection of two sets, is an empty set, then we call this relation to be disjoint: `len(set1 & set2) == 0`

Example:

```
>>> set((1,2,3,4)) & set((6,7,8))
set()
>>> len({1,2,3,4} & {6,7,8})
0
```

There is also a method we can use to find out, whether 2 sets have any elements in common: `set1.isdisjoint(set2)`

```
>>> {1,2,3,4}.isdisjoint({6,7,8})
True
```



QUIZ

Dictionary

Nepodařilo se zobrazit část obsahu. Chyba je na naší straně.

Set

1/5

Select the correct set definitions:

A. Sets are ordered

B. Items in sets can repeat

C. Sets contain only unique items

HOME EXERCISES - DICTIONARY

Delete value

At the beginning we have a dictionary: `myNewDict = {'m': 12345, 'n': 32145, 'o': 54321, 'p': 23232, 'q': 43210, 'r': 13579}`

Your task is to complete the following instructions:

- First of all we want to get the key which has alphabetically the maximal value - `maximalValueofKey` ,
- second step is print that value out,
- if any value in our dictionary is greater than the value of our maximal key (`maximalValueOfKey`), we want to delete the whole item under the key `maximalValueOfKey`
- finally we want to print our new modified dictionary.

Online Python Editor

```
1
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Password check 2

In this task, we will try to verify if the user enters a password that belongs to his account.

The output should looks like:

```
~/PythonBeginner/Lesson3 $ python verify.py
Please enter username: Mark
Please enter password: 1234
Permission continue, GRANTED!
```

If the password will be type incorrectly or is incorrect in general:

```
~/PythonBeginner/Lesson3 $ python verify.py
Please enter username: Mark
Please enter password: 444
Password or username are WRONG!
```

Online Python Editor

```
1
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

HOME EXERCISES - SETS

Now let's practice what we've learned about sets.

We will work with these two strings:

```
String01 = 'Bratislava'
String02 = 'Budapest'
```

Write a script that will find and print only elements using a suitable operator or method:

1. Common - which have `String01` and `String02` common.
2. Unique - which charaters are present in `String01` but not in `String02` .

The output could look like this:

```
~/PythonBeginner/Lesson3 $ python common_unique_chars.py
Common characters: {'s', 'B', 't', 'a'}
Unique characters: {'i', 'r', 'l', 'v'}
```

Online Python Editor

```
1
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Difference & All

Now, we will work once again with these two strings:

```
String01 - 'Bratislava'
String02 - 'Budapest'
```

Write a script that will find and print only the elements using a suitable operator or method:

1. Difference - which `String01` and `String02` do not share. In other words, the elements that are located in `String01` , `String02` , but not in both.
2. All - which `String01` and `String02` share and do not share - all elements

The output could look like this:

```
~/PythonBeginner/Lesson3 $ python different_all_chars.py
```

```
}
```

Online Python Editor

```
1
```

spustit kód

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

100% z Lekce 3