PYTHON COURSES

Lambda functions

→ functions without any names;
→ often used as a pointer to function equivalent when dealing with other functions that expect a callback;
→ useful to implement closures;
→ lambda <list_of_parameters>: return_value
→ example:

```
addition = lambda x, y: x+y
print(addition(3,5))
```

→ a lambda with a specific behavior can be built at run-time using the data dynamically generated <=> lambdas are kind of closure;
→ example:

```
def create_function(n):
    return lambda x: x%n == 0
by3Div2 = create_function(2)
x = 14
print(x, by3Div2(x))
```

Sequences

→ data structure represented by a vector of elements that don't need to be of the same type;
→ 2 representations <=> list and type;
→ list <=> mutable vector, can be defined using [...] or the list keyword;

→ mutable vector (=> elements from that list can be added, deleted etc.;
→ immutable vector (=> the exact equivalent is a constant list (addition etc. can't be used);
→ tuple (=> immutable vector, usually defined using (..) or by using the tuple keyword;
→ the list and tuple keyword can be used to initialize a tuple or list from another list or tuple;
→ example:

```
x = list()
x = []
# [1,2,1,2,1,2]
x = [1,2] * 3
y = tuple()
y = ()
y = (1,2,"test")
# (1,10)
y = 1, 2*5
x = [1,2,"eva"]
```

→ elements from a list can be accessed in a similar way the elements from a tuple can be accessed;
→ example:

```
x = ['A', 'B', 2, 3, 'C']
y = ('A', 'B', 2, 3, 'C')
# A
x[0]
# C
y[-1]
# ['A', 'B', 2]
x[:3]
# ('B', 'C')
y[3:]
```

```
# ['B', 2]
x[-1:3]
# ('B')
y[1:-3]
```

→ the tuple and list keywords can be used to convert a type to a list and vice-versa;
→ lists and tuples can be concatenated but not with each other;
→ example:

```
x = ('A', 'B', 2, 3, 'C')
# y = ['A', 'B', 2, 3, 'C']
y = list(x)
a = ('A', 2)
b = ('B', 3)
# z = ('A', 2, 'B', 3)
~~z = x+y~~ z = a+b
c = ['C', 4]
d = ['D', 8]
# e = ['C', 4, 'D', 8]
e = c+d
```

→ tuples can be used to return multiple values from a function;
→ example:

```
def sumProd(x,y):
    s = x+y
    p = x*y
    return (s,p)
sum, prod = sumProd(4,5)
```

→ tuples and lists can be organized in matrixes;

→ example:

```
x = ((1,2,3), (4,5,6))
y = ([1,2,3], (4,5,6))
z = (((1,2,3), (4,5,6)), ((7,8), (9,10,11)))
a = [[1,2], [4,5]]
b = [(1,2,3), [4,5,6]]
```

→ lists and tuples can be enumerated with a for keyword;

→ example:

```
for i in (1,2,3,4):
    print(i)
```

→ lists and tuples have the for keyword that can be used to find out the size of a list/tuple;

→ the enumerate keyword can be used to enumerate a list and get the index of the item at the same time (or an external variable can be used);

→ example:

```
for i,n in enumerate(["Dr", "Mr", "Mr"]):
    print("Index: %d => %s" %(i,n))

i = 0
for n in ["Dr", "Mr", "Mr"]:
    print("Index: %d => %s" %(i,n))
    i += 1
```

→ enumerate functions allow a second parameter to specify the index base (default is 0);

→ example:

```
for i,n in enumerate(["Dr", "Mr", "Mr"],2):
    print("%d => %s" %(i,n))
# base index is 2
```

→ a list can be built using functional programming;

→ example:

```
x = [i for i in range(1,10)]
# x = [1,2,3,4,5,6,7,8,9]
# y = [23,46,69,92]
y = [i for i in range(1,100)
     if i%23 == 0]
# z = [[1,6],[2,5],[3,4],[4,3],[5,2],[6,1]]
z = [[x,y] for x in range(1,7) for y in
     range(1,7) if (x+y)%7 == 0]
# a = [(1,1),(1,2),(1,3),(2,1),(2,2),
# (2,3),(3,1),(3,2),(3,3)]
a = [(x,y) for x in range(1,4)
     for y in range(1,4)]
```

→ functional programming drastically reduces the size of code but depending on how large the expression is to build a list, it may not be advisable, if the program purpose is readability;

List functions

→ append(x) <=> add the new element x in the list (the operator += can be used too);

→ extend(y) <=> add y (list or tuple) to the list;

→ example:

```
x = [1,2,3]
# x = [1,2,3,4]
x[len(x):] = [4]
```

```
# x = [1,2,3,4,5]
x += [5]
# x = [1,2,3,4,5,6,7]
x += [6,7]   x += (6,7)
# x = [1,2,3,4,5,6,7,8,9]
x.extend((8,9))
```

→ insert (pos, elem) ⇐ insert a new element (elem) in the list);

→ example:

```
x = [1,2,3]
x.insert(1, "A")
# x = [1,"A",2,3]
x.insert(-1, "B")
# x = [1,"A",2,"B",3]
```

→ [:] ⇐ insert a new element or multiple elements;

→ [] ⇐ change the value of one element;

→ remove (a) ⇐ remove a given element (removes the first element with the value, if it exists and otherwise error);

→ example:

```
x = [1,2,3,4,5]
# x = [1,2,20,4,5]
x[2] = 20
# x = [1,2,20,"A","B","C"]
x[3:] = ["A","B","C"]
# x = [10,"B","C"]
x[:4] = [10]
# x = [10,"C"]
x.remove("B")
```

→ del list[pos] ⇒ remove an element given the its position given list;
→ pop(pos) ⇒ remove an element given the pos position and return it (it can be used without the pos parameter and in this case it refers to the last element);
→ example:

```
x = [1,2,3,4,5]
# x = [1,2,3,4]
del x[-1]
# x = [3,4]
del x[:2]
y = [1,2,3,4,5]
# y = [1,2,5]
del y[2:4]
# y = [1,2]
x = y.pop(-1), x = 5
# y = [1], x = 2
x = y.pop()
```

→ del x[:] same x.clear() ⇒ clear the entire list;
→ the = operator doesn't make a copy but only a reference of a list;
→ y = list(x) ⇒ y is the copy of x;
→ y = x.copy() same y = x[:] ⇒ y is a shallow copy of x;
→ index(elem) ⇒ find out the position of elem in the list (if elem doesn't exist, then error);
→ operator in (elem in x) ⇒ check if elem exists in x;

→ example :

```
x = [1, 2, 3, 4]
# y = 2
y = x.index(3)
# y = False
y = 5 in x
```

→ count(item) ⇒ find out how many item elements exist in a list ;

→ reverse() ⇒ reverse the order of the elements from a list ;

→ example :

```
x = [1, 2, 3, 2, 4, 2, 5]
# y = 3
y = x.count(2)
x = [1, 2, 3]
# x = [3, 2, 1]
x.reverse()
```

→ sort(key = None, reverse = False) ⇒ sort elements from a list ;

→ map(function, iterable Element, [iE1, -- iEn]) ⇒ create a new list where each element is obtained based on the lambda expression provided.

→ example :

```
x = [2, 1, 4, 3, 5]
# x = [1, 2, 3, 4, 5]
x.sort()
# x = [5, 4, 3, 2, 1]
x.sort(reverse = True)
# x = [5, 2, 4, 1, 3]
x.sort(key = lambda i: i%3, reverse=Tru
```

```
x=[1,2,3,4,5]
# y= [1,4,9,16,25]
y= list(map(lambda i: i*i, x))
a = [1,2,3]
b = [4,5,6]
# c = [5,7,9]
c= list(map(lambda a1,a2:
    a1+a2, a, b))
```

→ The map function returns an iterable object;
  ↳ list is used to create a list from an iterable object;
→ filter(function, iterableElement) ⇒ create a new list where each element is filtered based on the lambda expression provided;
→ filter and map can be used to create a list;
→ example:

```
x=[1,2,3,4,5]
#y=[2,4]
y= list(filter(lambda i: i%2==0,x))
# x = [1,4,9,16,25]
x= list(map(lambda x: x*x,
    range(1,6)))
# x1 = [2,4,6,8]
x1= list(filter(lambda x: x%2==0,
    range(1,9)))
```

→ max(iterableElement, [key]), max(x1, x2, ... [key])
(le plus grand ou min) ⇐ find out the biggest / smallest element given an iterable list based on the lambda expression provided;

→ sum(iterableElement, [startValue]) ⇐ add all elements from an iterable ~~list~~ object (the elements should allow the possibility of addition with other elements);
↳ startValue ⇐ the value from where to start summing the elements (default is 0)

→ example:

```
x = [1,2,3,4,5]
# y = 9
y = max(1,3,2,7,8,3,5)
# y = 2
y = max(x, key = lambda i: i%3)
# y = 1
y = min(x)
# y = 115 = 100 + 15 , x = 15
x = sum(x)
y = sum(x,100)
# error ⇐ can't add int and string
x = [1,2,"3"]
y = sum(x)
```

→ sorted(iterableElement, [key], [reverse]) ⇐ sort the element given a list (iterable object), where the key represents a compare function between 2 elements of the iterable object (reverse is default False);

→ reversed(x) <=> reverse the element given a list (iterable object);

→ any(x), all(x) <=> check if at least one/all elements given a list (iterable object) can be evaluated to true;

→ example:

```
x = [2,1,4,3,5]
# y = [1,2,3,4,5]
y = sorted(x)
# y = [2,5,1,4,3]
y = sorted(x, key = lambda i: i%3,
           reverse = True)
x = [2,1,4,3,5]
# y = [5,3,4,1,2]
y = list(reversed(x))
x = [2,1,0,3,5]
# y = True (all but 0 are True)
y = any(x)
# y = False
y = all(x)
```

→ zip(x,y) <=> group 2 or more iterable objects into one iterable object;

→ zip(*x) <=> unzip such a list (the unzip variables are tuples);

→ del x <=> delete x (a list/tuple);

→ example:

```
x = [1,2,3]
y = [10,20,30]
# z = [(1,10), (2,20), (3,30)]
z = zip(x,y) list(zip(x,y))
```

```
a = [(1,2), (3,4), (5,6)]
# c = (1,3,5) and d = (2,4,6)
c,d = zip(*x)
x = [1,2]
del x
# error (x doesn't exist now)
print(x)
```

Sets

→ a and b are unique elements, if a is different from b (a is a different type than b or a and b are the same type but have different values).

→ set <=> a list of unique data;

→ the set keyword or {---} can be used to build a set (the set keyword can be used to initialize a set from types, lists or strings);

→ elements from a set can't be accessed (they are unordered collections);

→ there is no addition operation defined between 2 sets;

→ add(elem) <=> add a new element (item) in the set;

→ remove(elem) or discard(elem) <=> remove elem from the set (remove throws an error, if the set doesn't contain the element);

→ clear() ⇒ empty the entire set;

→ example :

```
# x = {}
x = set()
x = {1, 2, 3, "A", "B"}
y = set((1, 2, 3, 2))
y = set([1, 2, 3, 2])
# y = {1, 2, 3}
x.add(4)
# x = {1, 2, 3, "A", "B", 4, "#"}
x.add("#")
# x = {2, 3, "A", "B", 4}
x.remove(1)
x.discard("#")
# y = {}
y.clear()
```

→ update(y) or the |= operator ⇒ add the elements given y to the set (update can be called with multiple parameters / sets);

→ union(y) or the | operator ⇒ perform the union operation (union can be called with multiple sets);

→ example :

```
x = {1, 2, 3}
# x = {1, 2, 3, 4, 5, 6}
x |= {4, 5, 6}
# x = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12}
x.update({8, 9})
x.update({10}, {11}, {12}, {10, 11})
```

```
a = {1, 2, 3}
b = {4, 5, 6}
t = {2, 4, 6}
# x = {1, 2, 3, 4, 5, 6}
x = x / y / x
y = {7, 8}
# w = {1, 2, 3, 4, 5, 6, 7, 8}
w = a. union (a, b, t)
```

→ intersection (.) or the & operator (=) the intersection operation with multiple sets (intersection can be called

→ difference (-) or the - operator (=) the difference operation (difference can be called with multiple sets);

→ symmetric_difference (^) or the ^ operator (=) (symmetric_difference operation (symmetric_difference can only be called with one set);

→ example:

```
x = {1, 2, 3, 4}
y = {2, 3, 4, 5}
t = {3, 4, 5, 6}
# x = {3, 4}
x = x & y & t
# w = {3, 4}
w = x. intersect (y, t)
# x = {1}, w = {6}
x = x - y
w = t. difference (x, y)
```

```
# x = {1, 5}, w = {2, 6}
z = x ^ y
w = y.symmetric_difference(x)
```

→ all set operations support some operators that apply to one variable (=> intersection - update operation - update or returns a new set without the unwanted items), difference - update or -=, symmetric_difference - update or ^=;

→ in operator (=> test if an element exists in a set;

→ len keyword (=> total number of elements from a set;

→ example:

```
x = {"a", "b", "c"}
y = {"c", "d", "a"}
z = {"g", "g", "c"}
# x = {"c"}
x.intersection_update(y, z)
# a = True
a = "d" in y
# b = 3
b = len(x)
```

→ isdisjoint(y) (=> test if a set has no common elements with the y set;

→ issubset(y) or the <= operator (=> test if a set is included in y;

→ issuperset(y) or the >= operator ⇐⇒ test if a set is included in y;
→ the > and < operators ⇐⇒ check if a set is included in another but is not identical to it;
→ example:

```
x = {1,2,3,4}
y = {10,20,30}
# x = True
x = x.isdisjoint(y)
a = {1,2,3,4,5,6}
# x = True, t = True
x = x.issubset(a)
t = x <= a
# x = True, t = True
x = a.issuperset(x)
t = a >= x
b = {1,2,3,4}
# x = False
x = x > b
```

→ pop() ⇐⇒ remove one element from the set;
→ copy() ⇐⇒ make a shallow copy of a set;
→ a set can be built by using functional programming;
→ the condition of the set (all elements are unique) still applies when using functional programming;

→ example :

```
x = {"A", "a", "B", "b", 1, 2, 3}
y = x.copy()
# y = {"A", "a", "B", "b", 1, 2, 3}
y.pop()  # ion randomly
# z = {23, 46, 63, 52}
# = {i for i in range(1, 100)
    if i%23 == 0}
# t = {0, 1, 2, 3, 4}
t = {i%5 for i in range(0, 100)}
```

→ map ⇐ create a new set where each element is obtained based on the lambda expression provided;

→ filter ⇐ creat a new set where each element is filtered based on the lambda expression provided;

→ both filter and map are used to creat a set;

→ min, max, sum, any, all, sorted, reversed ⇐ work in a similar way as the built-in functions for lists;

→ for statement can be used to enumerate between elements of a set;

→ frozenset ⇐ has all the characteristics of a normal set but it can't be modified (if you try to add a new element for example, then you will);

→ example:

```
x = {1,2,3,4,5}
y = {7,8,9,10,11}
# x = {8,10,12,14,16}
x = set(map(lambda x1,x2: x1+x2, x, y))
# t = {2,4}
t = set(filter(lambda i: i%2==0, x))
# a = {1,4,9,16,25}
a = set(map(lambda x: x*x,
        range(1,6)))
# b = {1,8,15,22}
b = set(filter(lambda x: x%7==1,
        range(1,25)))
for i in {1,2,3,4,5}:
    print(i)
x = frozenset({1,2,3})
```

Dictionaries

→ dictionary ⟷ python implementation of a hash-map container, designed as a key-value pair where key is an unique element within the dictionary;
→ the dict keyword or {...} ⟷ create a dictionary;
→ the [] operator ⟷ set a value in a dictionary or read an existing value (if a value doesn't exist, an exception will be thrown);
→ the in operator ⟷ check if a key exists in a dictionary;

→ len <=> find out how many keys a dictionary has;

→ example:

```
x = dict()
y = {}
z = {"A": 1, "B": 2}
z1 = dict(abc = 1, aaa = 2)
z2 = dict({"abc": 1, "aaa": 2})
z3 = dict([("abc", 1), ("aaa", 2)])
z4 = dict((("abc", 1), ("aaa", 2)))
z5 = dict(zip(["abc", "aaa"], [1, 2]))
y["ABC"] = 2
# a = 2
a = y["ABC"]
# True
"A" in z
# by = 2
len(z1)
```

→ setdefault <=> manipulate values from a dictionary;

→ update <=> change the value associated with a key;

→ del or clear <=> delete an element from a dictionary;

→ copy or static method fromkeys <=> create a new dictionary;

→ example:

```
x = {"A": 1, "B": 2}
# x = {"A": 1, "B": 2, "C": 3}, y = 3
y = x.setdefault("C", 3)
```

```
# y=None, x={"A":1,"B":2,"C":3,
# "D":None}
y=x.setdefault("D")
# x={"A":1,"B":2,"C":3,"D":None}
# y=4
y=x.setdefault("A")
# y=2
y=x.setdefault("B",20)
x1={"A":1,"B":2}
# x1={"A":10,"B":2}
x1.update({"A":10})
# x1={"A":10,"B":2,"C":3}
x1.update({"C":3})
#x1={"A":10,"B":2,"C":3,
# "D":12,"E":34}
x1.update(D=12,E=34)
# x1={"B":2,"C":3,"D":12,
# "E":34}
del x1["A"]
x1.clear()
# error (x1 is empty)
del x1["C"]
# y={"A":1,"B":2,"C":3}
# x2={"A":1,"B":2,"C":3}
x2={"A":1,"B":2}
y=x2.copy()
y["C"]=3
# x3={"A":2,"B":2}
x3=dict.fromkeys(["A","B"],2)
```

(22)

→ get (=) access elements from a dictionary;
→ pop (=) extract an element from a dictionary;
→ a dictionary can be built using functional programming;
→ example:

```
# x = {"A": None, "B": None}
x = dict.fromkeys(["A", "B"])
y = {"A": 1, "B": 2}
# x = 1, x1 = None, x2 = 123
x = y.get("A")
x1 = y.get("C")
x2 = y.get("C", 123)
# a1 = 1, a2 = 123, y = {"B": 2}, y = {"B": 2}
a1 = y.pop("A")
a2 = y.pop("C", 123)
# error (D doesn't exist and no
# default value is provided
a3 = y.pop("D")
x = {i: i for i in range(1, 9)}
# x = {1: 1, 2: 2, ---, 9: 9}
# x1 = {0: 6, 1: 7, 2: 8} (=) last
# values were updated
x1 = {i%3: i for i in range(1,9)}
# x2 = {2: "B", 4: "D", 6: "7", 2: "H"}
x2 = {i: chr(64+i) for i in
range(1,9) if i%2 == 0}
```

→ keys (=) obtain the keys from the dictionary;
→ values (=) obtain the values from the dictionary;
→ when iterating through the keys/values of a

dictionary, the output order may be different for various of python depending on how data is stored/indexed in memory;

```
x = {"A": 1, "B": 2}
# y = ["A", "B"] iterable object
y = x.keys()
# z = ["1", "2"] iterable object
z = x.values()
# iterat through keys <=> A, B
for i in x:
    print(i)
for i in x.keys():
# iterate print(i)
for i in x.values():
    print(i)
```

→ items <=> obtain all the pairs from a dictionary;

```
x = {"A": 1, "B": 2}
# y = [("A": 1), ("B": 2)] an iterable object in
# python 3 or a list of tuples for python 2
y = x.items()
# iterate through all the pairs
# ("A", 1), ("B", 2)
for i in x.items():
    print(i)
x = {"Dacia": 420, "BMW": 460, "Toyota": 440}
# ("Dacia", 420), ("Toyota, 440), ("BMW", 460)
for i in sorted(x.items(), key=lambda a: a[1]):
    print(i)
```

→ the ** operator (=> used in a function to specify that the list of parameters of that function should be treated as a dictionary;

→ filter can be used with dictionaries;

→ for keyword (=> delete an entire dictionary;

→ the enumerate keyword can be used with dictionaries;

→ enumerate can receive a secondary parameter just like in the case of ranges (lists) that states the initial index;

```
→ def get(**cars):
      min_speed = 0
      name = None
      for car_name in cars:
          if cars[car_name] > min_speed:
              name = car_name
              min_speed = cars[car_name]
      return name

  # fastest = "BMW"
  fastest = get(Dacia=120, BMW=160, Toyota=140)

  x={"D": 120, "B": 160, "T": 140}
  # y = {"T": 140, "B": 160}
  y = dict(filter(lambda x: x[1]>=140, x.items()))
  # (0, "D"), (1, "B"), (2, "T")
  for a in enumerate(x):
      print(a)

  # enumerate(x,2) will start with the index 2
```

Exceptions

```
try:
    # code
    x = 5/0
except:
    # code that will be executed in case
    # of any exception
    print("Exception")
else:
    # code that will be executed, if
    # there is no exception
    print("All okay")
```

→ all exceptions in python are derived from BaseException class;

→ a custom (user-defined) exception type can be used;

```
def Test(y):
    try:
        x = 5/y
    except ArithmeticError:
        print("A.E")
    except:
        print("Genric")
    else:
        print("Okay")

# A.E, Genric, Okay
Test(0)   Test("aaa")   Test(1)
          another line  another line
```

→ simple except: <=> after all the other exception types (exception Type n:) and before else:;

→ finally keyword <=> can be used to execute something at the end of the try block (after else:);

```
def test(y):
    try:
        x=5/y
    except:
        print("Error")
    else:
        print("okay")
    finally:
        print("Finally")

# Error, Finally (different lines)
Test(0)
# okay, Finally
Test(1)
```

```
def test1(y):
    try:
        x=5/y
    except (ArithmeticError, TypeError):
        # except (Type1, Type2, --- Type n)
        print("Exception")
    except:
        print("Generic")
    else:
        print("All okay")
```

```
def test2():
    try:
        x = 5/0
    except Exception as e:
        print(str(e)) #division by 0
    # except Type1 as <var_name>:


#Exception
test1(0)
# Exception
test1("aaa")
#Aee okay
test1(1)
```

```
try:
    x = 5/0
except (Exception, ArithmeticError, TypeError) as e:
    print(str(e)) # division by 0
    # except (Type1, Type2, ... Typen) as <var>:
```

→ the raise keyword ⇐ used / throws an exception;

- any `raise ExceptionType (parameters)`
- `raise ExceptionType (param) from <exception_var>`

```
try:
    raise Exception("Testing")
except Exception as e:
    # Testing
    print(e)
```

```
try:
    raise Exception("Param1", 10, "Param3")
except Exception as e:
    params = e.args
    # 3
    print(len(params))
    # Param1
    print(params[0])
```

→ the raise keyword can be used without parameters (it indicates that the current exception should be re-raised);

```
try:
    try:
        x = 5/0
    except Exception as e:
        print(e) # division by 0
        raise
except Exception as e:
    # raise: division by 0
    print("raise:", e)
```

→ the assert keyword = raise an exception based on the evaluation of a condition;

```
age = -1
try:
    assert (age>0), "Age>0"
except Exception as e:
    # Age>0
    print(e)
```

→ the pass keyword <=> usually used if you want to catch an exception but don't want to process it;

→ some exceptions (if not handled) can be used for various purposes;

→
```
try:
    x = 10 / 0
except:
    pass
print("Test")
raise SystemExit
print("Test 1")
# Test
# this exception, if not handled, will
# immediately terminate the program
```

Modules

→ modules <=> python's libraries and extend python functionality;

→ the import keyword <=> import modules;

→ classes and items from a module can be imported separately or from module import * ;

→ when importing a module, aliases can be used using the as keyword <=> import module1 as alias1, [m2 as a2, ... mn as an]

→ the dir keyword <=> obtain a list of all the

functions and objects that a module exports
(dir(module))
→ sys.argv ⇐ provides a list of all parameters that have been sent from the command line to a python script, the first parameter is the name/path of the script (sys.argv[1:]);
→ os.listdir ⇐ returns a list of child files and folders (os.listdir("."));
→ os.mkdir or os.makdirs ⇐ create folders;
→ os.chdir ⇐ change the current path;
→ os.rmdir or os.removedirs ⇐ delete a folder;
→ os.remove or os.unlink ⇐ delete a file;
→ os.rename or os.renames ⇐ rename/move operations;
→ the path submodule ↔ os ⇐ can be used to perform different operations with file & ~~directories~~ directories, paths;
→ the os module can be used to execute a system command or run an application via a system function (os.system("dir *.* /a"))

Input/Output

→ the content read from the input is considered to be a string and returned;

```
x = input("Enter: ") # from keyboard → 123
# Enter: 123
print(x)
```

→ print can be used to print a string or an object/variable that can be converted into a string;
→ a file can be opened using the open keyword (FileObject = open(path, mode="r", buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None));
→ "r" ⇐ read; "w" ⇐ write; "x" ⇐ exclusive creation (fails, if the file exists); "a" ⇐ append;
"b" ⇐ binary mode; "t" ⇐ text mode;
"+" ⇐ update (read and write);
encoding ⇐ if the file is opened in text mode and you need translation from different encodings (UTF etc.);
→ default mode is "r";
→ errors ⇐ specify the way conversion errors for different encodings should be processed;
→ newline ⇐ for text mode, specify what should be considered a new line (if this value is set to None, the character that is specific for the current operating system will be used;
→ f ⇐ file object;
→ f.close ⇐ close current file;
→ f.tell ⇐ return the current file position;
→ f.seek ⇐ set the current file position;
→ f.read ⇐ read a number of bytes from the file;
→ f.write ⇐ write a number of bytes into the file;

→ f.readline() ⟷ read a line from the file;
→ a file object is iterable and returns all
txt lines from a file;
→ while or strip ⟷ remove the line-feed terminator
(lines read using this method contain it);

```
for line in open("a.py"):
    print(line.strip())
```

→ the read method returns a string in python 2
and a buffer or string depending on how the
file is opened in python 3;

```
# functional programming
x = [l for l in open("f.txt") if "for" in
     l.strip()]
# read the entire content of the file in a buffer
data = open("f.txt", "rb").read()
print(len(data))
print(data[0])
# the output in python 3 → a character,
# in python 3 → a number (the ascii code
# of that character
# obtain a string in python 3 → "rt" (instead
# of "rb"
```

→ write ⟷ create a file and write content in it
(open("f.txt", "wt").write("New file."))
→ it's a good policy to embed file operations
in a try block;

→ once a file is opened, the file object handle can be used to retrieve different information regarding that file (f.name, f.mode etc);

Re Module

→ module for regular expressions;

Character	Match
.	all characters except newline
^	matches at the start of the string
$	matches at the end of the string
*	>= 0 repetition(s)
.?	0 or 1 occurance
+	>= 1 repetition(s)
{x}	matches <x> times
{x,y}	matches between <x> and <y> times
[ ]	group of characters
|	or condition
\d	decimal characters 0,1, ---, 9
\D	all except characters designated by \d
\s	space, tab, newline (CR/LF) characters
\S	all except characters designated by \s
\w	word characters (a-z, A-Z, 0-9, _)
\W	all except characters designated by \w
\	escape character
[^...]	not specified group of characters
(...)	grouping
[..-..]	'-' interval for a group of characters

→ re.compile(regular_expression, flags) ⇐ compile a regular expression into its binary form;

→ match ⇐ check if a string matches a regular expression;

```
r = re.compile("07[0-9]{8}")
if r.match("0740123456"):
    # Match
    print("Match")

# Match
if re.match("07[0-9]{8}", "0740123456"):
    print("Match")
```

→ re.match ⇐ starts the matching from the beginning of the string and stops one the matching ends (except for the case where the regular expression pattern is using the "$" character);

→ search ⇐ check if a regular expression pattern is matching a part of a string (stops after the first match is achieved); can be used with a compiled object;

```
r = re.compile("\d+")
# Found
if r.search("Price is 123 USD"):
    print("Found")

# Found
if re.search("\d+", "Price is 123 USD"):
    print("Found")
```

→ search and match return a match object (always evaluated to True);

→ if the search doesn't find a match, None is returned and will be evaluated to False;

→ group(index) => member of a match object that returns the substring that matches that specific group (if the index is 0, the substring refers to the entire string that was matched by the regular expression);

→ lastindex => member of a match object that returns the index of the last object that was matched by the regular expression (to count a group within the regular expression, we must use (...));

→

```
r = re.search("\d+", "Price is 123 USD")
if r:
    # 123
    print(r.group(0))
# if re.search
r = re.search(".*(\d+)", "The 12345 $")
if r:
    #5
    print(r.group(1))
r = re.search("(\d+)[^\d]*(\d+)", "Price is 123 USD approx 110 EUR")
if r:
    #2
    print(r.lastindex)
    for i in range(0, r.lastindex+1):
        # 0 => 123 USD, approx 110
        # 1 => 123
        # 2 => 110
        print(i, "=>", r.group(i))
```

→ in case of some quantifiers (like *, ?, +), they can be preceded by ? (this will specify a non-greedy behaviour);

→ (...) is usually used to delimit specific sequences of substrings within the regular expression pattern;

→ findall(regular_expression, string) <=> find all substrings that match a specific regular expression given a string (the result is a vector containing all substrings);

```
r = re.findall("\d+", "Cele doua nr 20,30 si 123")

if r:
    #['20','30','123']
    print(r)
```

→ using group(...) is allowed (they will be executed to a tuple in each list element);

```
r = re.findall("(\d)(\d+)", "Cele doua nr 20,30 si 123")

if r:
    #[('2','0'),('3','0'),('1','23')]
    print(r)
```

→ split(regular_expression, string) <=> split a string using a regular expression (the result is a vector with all elements that substrings that were obtained after the split);

→ groups can be used (the split is done after each group that matches);

```
r = re.split("[colour]+", "Color Green pixel 20,30
    is 123")
#['C', 'l', 'r Gr', 'n p', 'x', 'l 20,30', 's 123']
print(r)
r = re.split("((\d)(\d)", "Color Green pixel 20,
    30 is 123")
#['Color Green pixel ', '2', '0', ',', '3', '0', 'is', '1', '2', '3']
print(r)
# ['', '']
print(re.split("\d|\d+", "12345"))
# ['', '', '', '2', '', '3', '', '4', '', '5', '']
print(re.split("(\d)", "12345"))
# ['Color Green ', '2', '0', ',', '3', '0', ' is ', '123', '']
print(re.split("(\d|\d+)", "Color of 20,30 is 123"))
```

→ if () are used in the pattern, the text of all groups in the pattern are also returned as part of the resulting list;

→ split(pattern, string, max-split=0, flags=0) ⇐ max-split specifies how many times a split can be performed;

```
s = "Today I'm having a python course"
#['', 'oday', "I'm", 'having', 'a', 'python', 'course']
print(re.split("[^a-z']+", s))
#['', 'oday', "I'm having a python course"]
print(re.split("[a-z']+", s, 1))
#['Today', "I'm", 'having a python course']
print(re.split("[a-z']+", s, 2, flags=re.IGNORECASE))
```

→ sub(pattern, replace, string, count = 0, flags = 0) ⇐
replace a matched string with another string,
where pattern is a regular expression to search for,
replace is either a string or function to search for,
the string where the pattern will be searched,
count is how many times the replacement can
occur (if missing or 0, it means all matches)
and flags is for some flags (like re.IGNORECASE)

```
s = "Today I'm having a python course"
# Today I'm not doing anything
print(re.sub("having|s+a|s+|w+|s+course",
      "not doing anything", s))
# Today I'm not doing the python course
print(re.sub("having|s+a|s+((|w+)|s+course",
      r"not doing the 1! course", s))
```

→ if the replace parameter is a string, the
\<number> greater that is found within the
replacement string will be replaced with the
group from the match search (1 will be replaced
by group(1));

→ if the replace parameter is a function, it
receives the match object and usually that function
will use group(0) to get the string that was
matched and convert it to the replacement value;

→ (?P<name> ...) ⇐ sets the name of a group to a
given string (in case of a match, that group
can be accessed based on its name);

```
s = "File size is 12345 bytes"
r = re.search("(?P<file_size>\d+)", s)
if r:
    # Size is 12345
    print("Size is ", r.group("file_size"))
```

→ (?i)(...) ⇐ ignores case will be applied for the current block match;

→ (?s)(...) ⇐ dot (.) will match everything;

→ (?=...) ⇐ match the previous expression only if the next expression is ... (look ahead assertion);

→ (?!...) ⇐ will match only if the next expression will not match ...;

→ (?#...) ⇐ represents a comment / information that can be added in the regular expression to reflect the purpose of a specific group;

Dynamic code

→ exec ⇐ used to dynamically compile and execute python code;

→ exec(code, [global], [local]) ⇐ global and local represents a list of global and local definition that should be used when executing the code;

→
```
exec("x = 100")
# 100
print(x)
exec("def sum(x,y): return x+y")
print(sum(10,20))  # 30
```