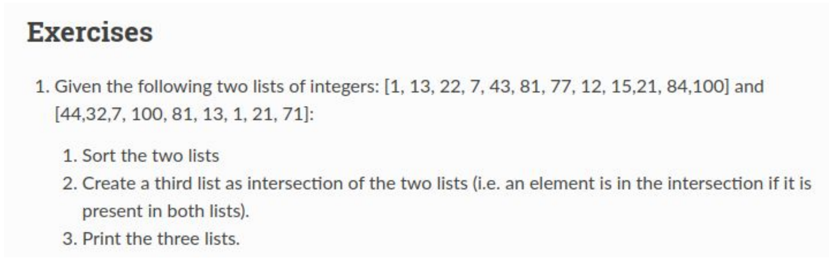
Scientific Programming Practical 6

Introduction

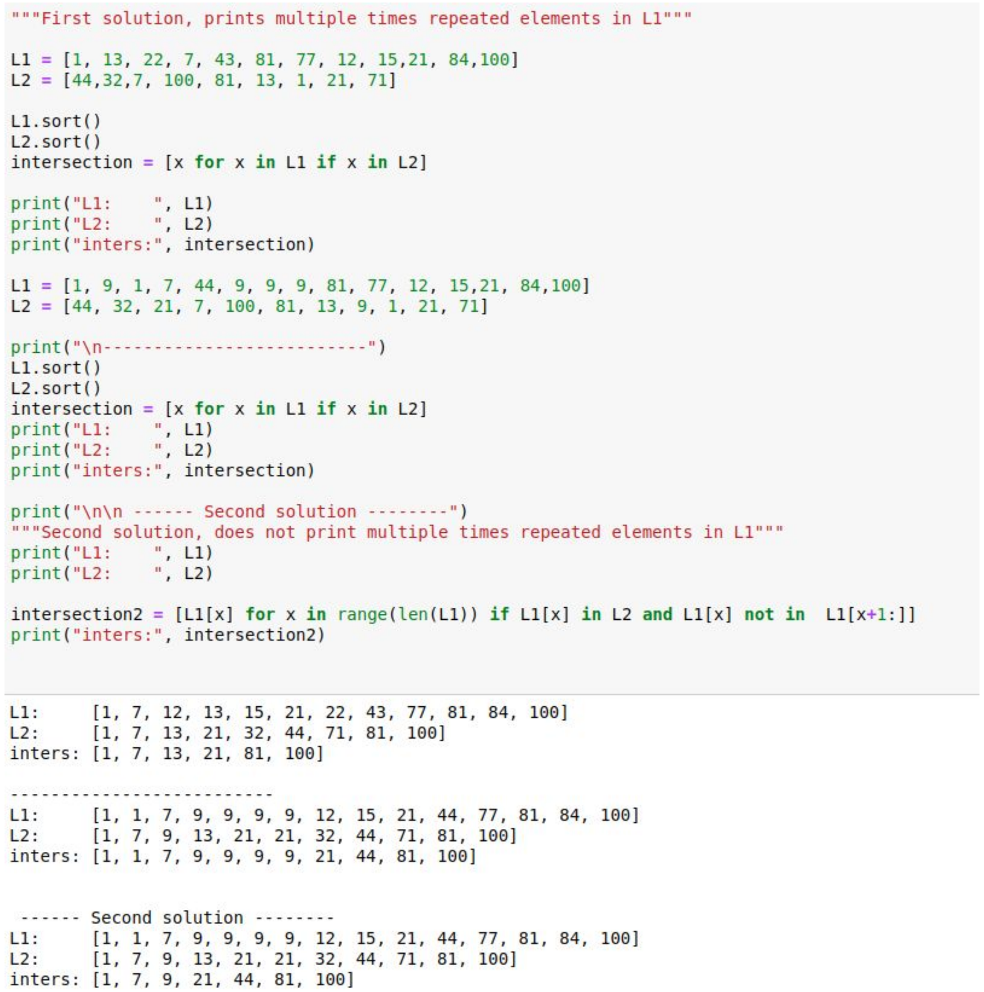
Luca Bianco - Academic Year 2020-21
luca.bianco@fmach.it

List comprehension: Ex.1



Exercises

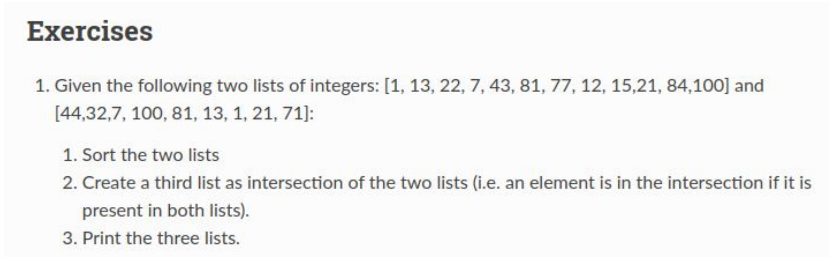
1. Given the following two lists of integers: [1, 13, 22, 7, 43, 81, 77, 12, 15,21, 84,100] and [44,32,7, 100, 81, 13, 1, 21, 71]:
    1. Sort the two lists
    2. Create a third list as intersection of the two lists (i.e. an element is in the intersection if it is present in both lists).
    3. Print the three lists.



```python
"""First solution, prints multiple times repeated elements in L1"""

L1 = [1, 13, 22, 7, 43, 81, 77, 12, 15,21, 84,100]
L2 = [44,32,7, 100, 81, 13, 1, 21, 71]

L1.sort()
L2.sort()
intersection = [x for x in L1 if x in L2]

print("L1:    ", L1)
print("L2:    ", L2)
print("inters:", intersection)

L1 = [1, 9, 1, 7, 44, 9, 9, 9, 81, 77, 12, 15,21, 84,100]
L2 = [44, 32, 21, 7, 100, 81, 13, 9, 1, 21, 71]

print("\n-------------------------")
L1.sort()
L2.sort()
intersection = [x for x in L1 if x in L2]
print("L1:    ", L1)
print("L2:    ", L2)
print("inters:", intersection)

print("\n\n ------ Second solution --------")
"""Second solution, does not print multiple times repeated elements in L1"""
print("L1:    ", L1)
print("L2:    ", L2)

intersection2 = [L1[x] for x in range(len(L1)) if L1[x] in L2 and L1[x] not in  L1[x+1:]]
print("inters:", intersection2)
```

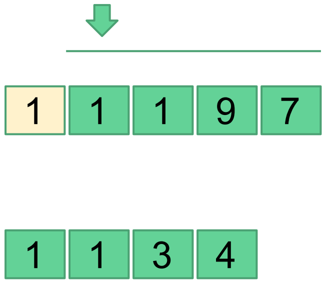
```
L1:     [1, 7, 12, 13, 15, 21, 22, 43, 77, 81, 84, 100]
L2:     [1, 7, 13, 21, 32, 44, 71, 81, 100]
inters: [1, 7, 13, 21, 81, 100]

-------------------------
L1:     [1, 1, 7, 9, 9, 9, 9, 12, 15, 21, 44, 77, 81, 84, 100]
L2:     [1, 7, 9, 13, 21, 21, 32, 44, 71, 81, 100]
inters: [1, 1, 7, 9, 9, 9, 9, 21, 44, 81, 100]


 ------ Second solution --------
L1:     [1, 1, 7, 9, 9, 9, 9, 12, 15, 21, 44, 77, 81, 84, 100]
L2:     [1, 7, 9, 13, 21, 21, 32, 44, 71, 81, 100]
inters: [1, 7, 9, 21, 44, 81, 100]
```

List comprehension: Ex.1

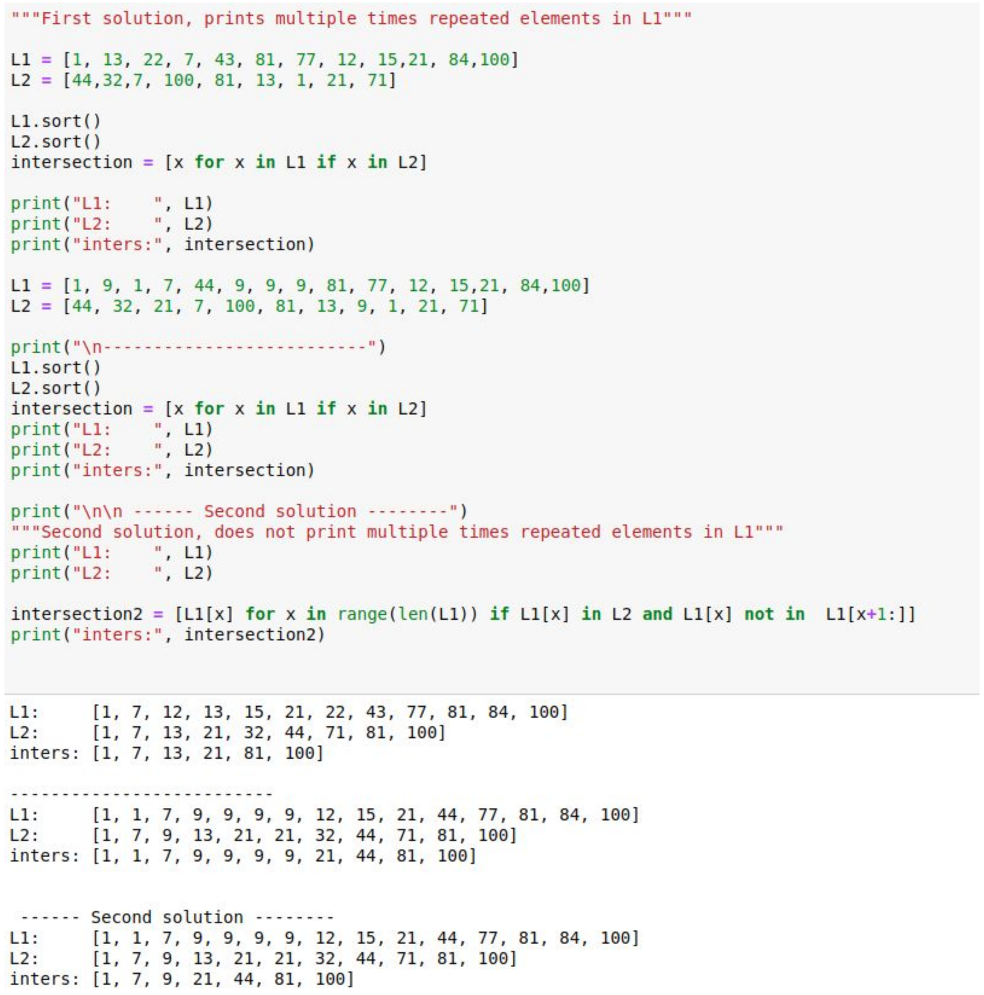


Exercises

1. Given the following two lists of integers: [1, 13, 22, 7, 43, 81, 77, 12, 15,21, 84,100] and [44,32,7, 100, 81, 13, 1, 21, 71]:
   1. Sort the two lists
   2. Create a third list as intersection of the two lists (i.e. an element is in the intersection if it is present in both lists).
   3. Print the three lists.



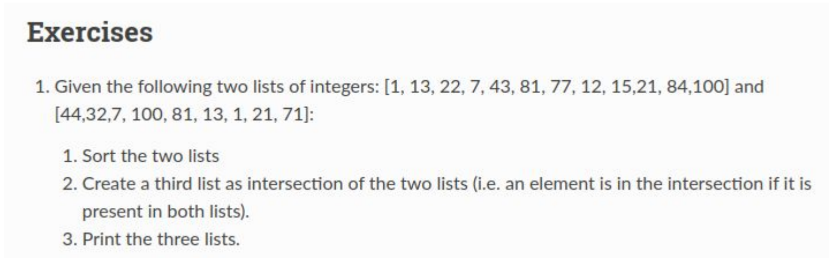
1 appears down the list. We will process it later!



```python
"""First solution, prints multiple times repeated elements in L1"""

L1 = [1, 13, 22, 7, 43, 81, 77, 12, 15,21, 84,100]
L2 = [44,32,7, 100, 81, 13, 1, 21, 71]

L1.sort()
L2.sort()
intersection = [x for x in L1 if x in L2]

print("L1:    ", L1)
print("L2:    ", L2)
print("inters:", intersection)

L1 = [1, 9, 1, 7, 44, 9, 9, 9, 81, 77, 12, 15,21, 84,100]
L2 = [44, 32, 21, 7, 100, 81, 13, 9, 1, 21, 71]

print("\n-------------------------")
L1.sort()
L2.sort()
intersection = [x for x in L1 if x in L2]
print("L1:    ", L1)
print("L2:    ", L2)
print("inters:", intersection)

print("\n\n ------ Second solution --------")
"""Second solution, does not print multiple times repeated elements in L1"""
print("L1:    ", L1)
print("L2:    ", L2)

intersection2 = [L1[x] for x in range(len(L1)) if L1[x] in L2 and L1[x] not in  L1[x+1:]]
print("inters:", intersection2)
```

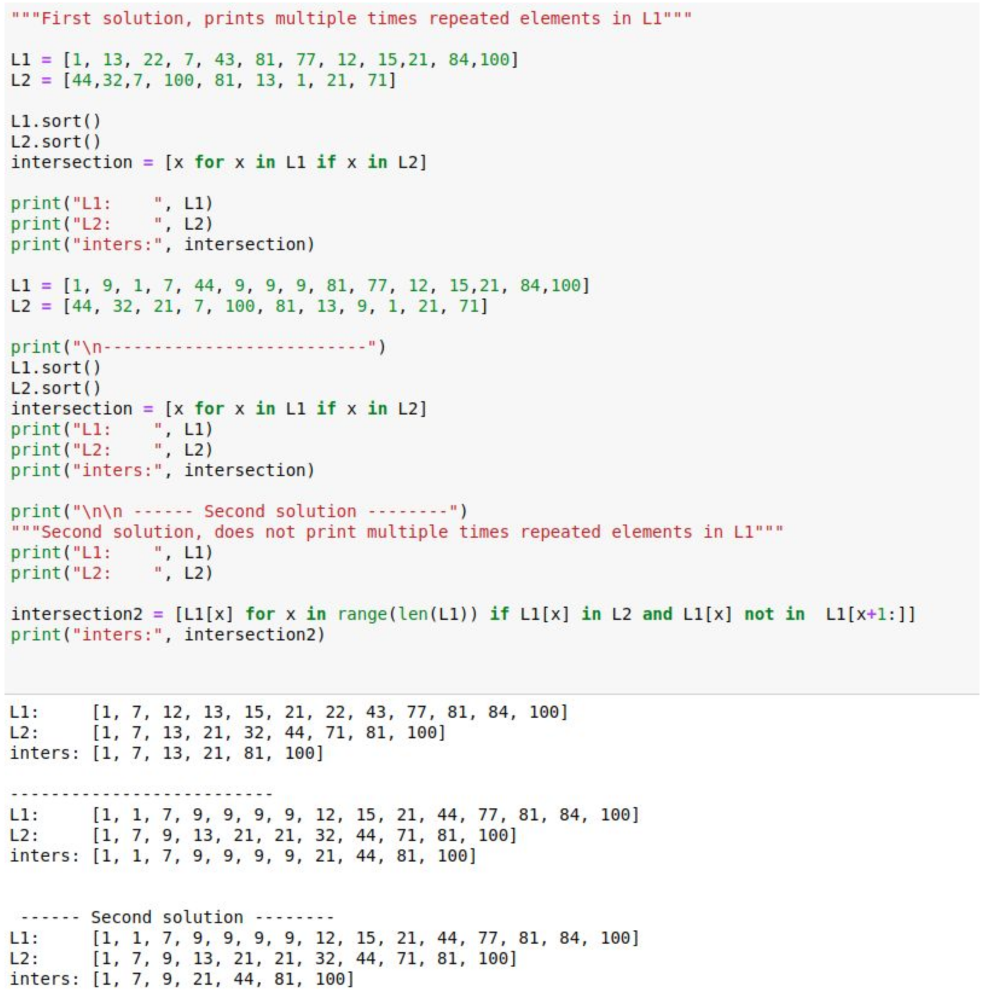
```
L1:     [1, 7, 12, 13, 15, 21, 22, 43, 77, 81, 84, 100]
L2:     [1, 7, 13, 21, 32, 44, 71, 81, 100]
inters: [1, 7, 13, 21, 81, 100]

-------------------------
L1:     [1, 1, 7, 9, 9, 9, 9, 12, 15, 21, 44, 77, 81, 84, 100]
L2:     [1, 7, 9, 13, 21, 21, 32, 44, 71, 81, 100]
inters: [1, 1, 7, 9, 9, 9, 9, 21, 44, 81, 100]


 ------ Second solution --------
L1:     [1, 1, 7, 9, 9, 9, 9, 12, 15, 21, 44, 77, 81, 84, 100]
L2:     [1, 7, 9, 13, 21, 21, 32, 44, 71, 81, 100]
inters: [1, 7, 9, 21, 44, 81, 100]
```

List comprehension: Ex.1



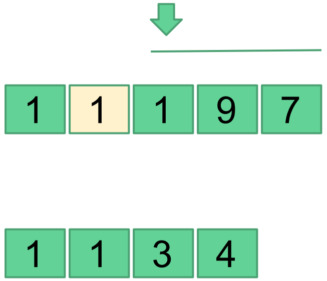
Exercises

1. Given the following two lists of integers: [1, 13, 22, 7, 43, 81, 77, 12, 15,21, 84,100] and [44,32,7, 100, 81, 13, 1, 21, 71]:
    1. Sort the two lists
    2. Create a third list as intersection of the two lists (i.e. an element is in the intersection if it is present in both lists).
    3. Print the three lists.



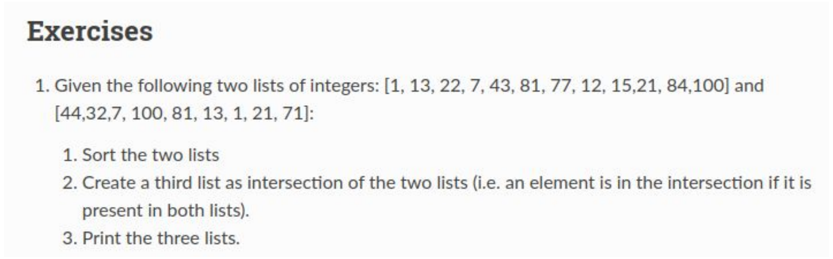
```python
"""First solution, prints multiple times repeated elements in L1"""

L1 = [1, 13, 22, 7, 43, 81, 77, 12, 15,21, 84,100]
L2 = [44,32,7, 100, 81, 13, 1, 21, 71]

L1.sort()
L2.sort()
intersection = [x for x in L1 if x in L2]

print("L1:     ", L1)
print("L2:     ", L2)
print("inters:", intersection)

L1 = [1, 9, 1, 7, 44, 9, 9, 9, 81, 77, 12, 15,21, 84,100]
L2 = [44, 32, 21, 7, 100, 81, 13, 9, 1, 21, 71]

print("\n-------------------------")
L1.sort()
L2.sort()
intersection = [x for x in L1 if x in L2]
print("L1:     ", L1)
print("L2:     ", L2)
print("inters:", intersection)

print("\n\n ------ Second solution --------")
"""Second solution, does not print multiple times repeated elements in L1"""
print("L1:     ", L1)
print("L2:     ", L2)

intersection2 = [L1[x] for x in range(len(L1)) if L1[x] in L2 and L1[x] not in  L1[x+1:]]
print("inters:", intersection2)
```

```
L1:     [1, 7, 12, 13, 15, 21, 22, 43, 77, 81, 84, 100]
L2:     [1, 7, 13, 21, 32, 44, 71, 81, 100]
inters: [1, 7, 13, 21, 81, 100]

-------------------------
L1:     [1, 1, 7, 9, 9, 9, 9, 12, 15, 21, 44, 77, 81, 84, 100]
L2:     [1, 7, 9, 13, 21, 21, 32, 44, 71, 81, 100]
inters: [1, 1, 7, 9, 9, 9, 9, 21, 44, 81, 100]


 ------ Second solution --------
L1:     [1, 1, 7, 9, 9, 9, 9, 12, 15, 21, 44, 77, 81, 84, 100]
L2:     [1, 7, 9, 13, 21, 21, 32, 44, 71, 81, 100]
inters: [1, 7, 9, 21, 44, 81, 100]
```



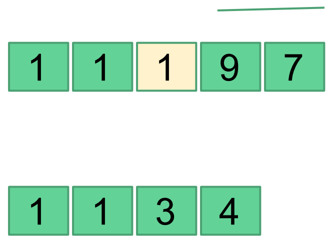
1 appears down the list. We will process it later!

List comprehension: Ex.1

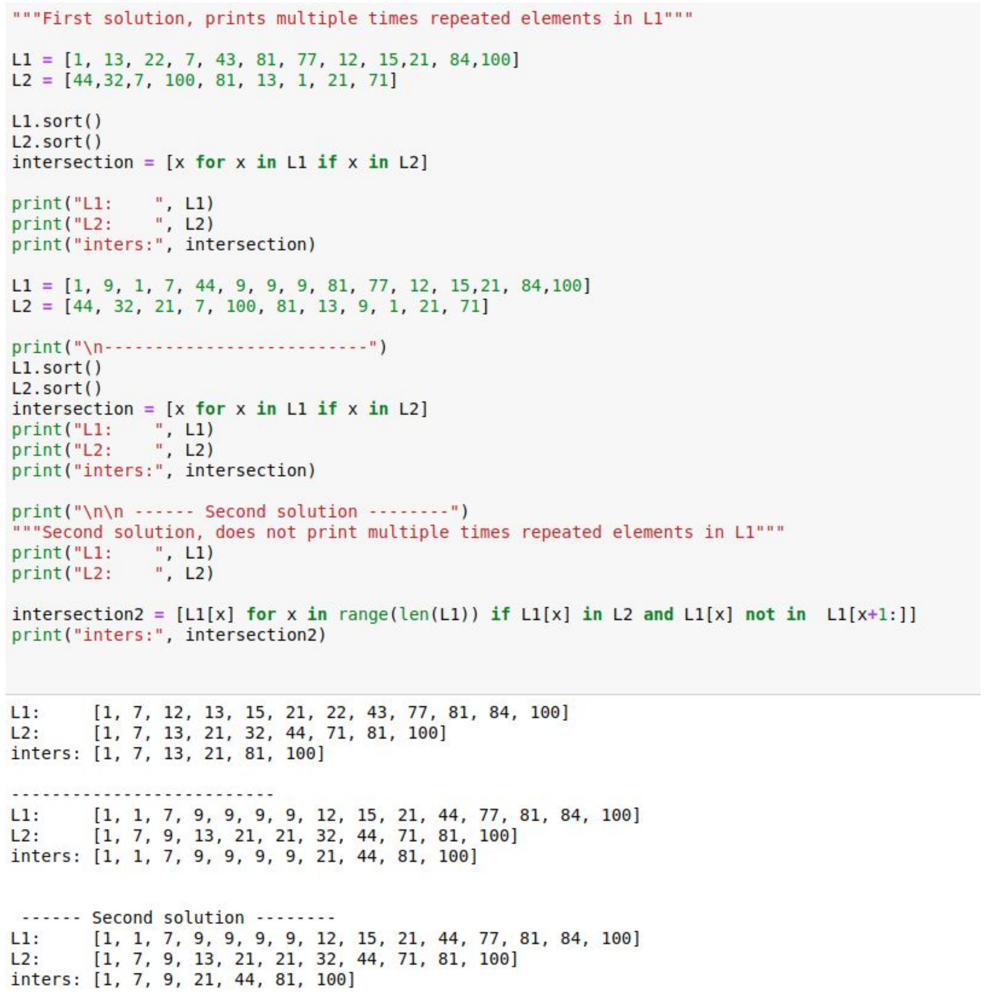


Exercises

1. Given the following two lists of integers: [1, 13, 22, 7, 43, 81, 77, 12, 15,21, 84,100] and [44,32,7, 100, 81, 13, 1, 21, 71]:
   1. Sort the two lists
   2. Create a third list as intersection of the two lists (i.e. an element is in the intersection if it is present in both lists).
   3. Print the three lists.



ok check intersection



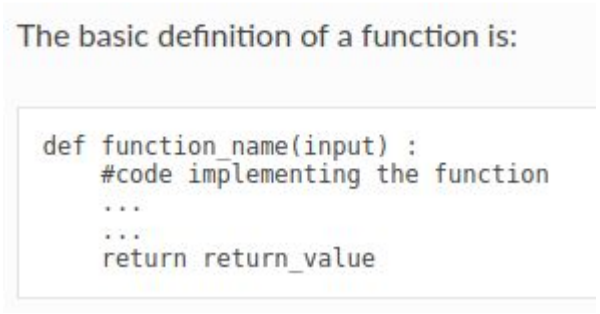
```python
"""First solution, prints multiple times repeated elements in L1"""

L1 = [1, 13, 22, 7, 43, 81, 77, 12, 15,21, 84,100]
L2 = [44,32,7, 100, 81, 13, 1, 21, 71]

L1.sort()
L2.sort()
intersection = [x for x in L1 if x in L2]

print("L1:     ", L1)
print("L2:     ", L2)
print("inters:", intersection)

L1 = [1, 9, 1, 7, 44, 9, 9, 9, 81, 77, 12, 15,21, 84,100]
L2 = [44, 32, 21, 7, 100, 81, 13, 9, 1, 21, 71]

print("\n-------------------------")
L1.sort()
L2.sort()
intersection = [x for x in L1 if x in L2]
print("L1:     ", L1)
print("L2:     ", L2)
print("inters:", intersection)

print("\n\n ------ Second solution --------")
"""Second solution, does not print multiple times repeated elements in L1"""
print("L1:     ", L1)
print("L2:     ", L2)

intersection2 = [L1[x] for x in range(len(L1)) if L1[x] in L2 and L1[x] not in  L1[x+1:]]
print("inters:", intersection2)
```

```
L1:     [1, 7, 12, 13, 15, 21, 22, 43, 77, 81, 84, 100]
L2:     [1, 7, 13, 21, 32, 44, 71, 81, 100]
inters: [1, 7, 13, 21, 81, 100]

-------------------------
L1:     [1, 1, 7, 9, 9, 9, 9, 12, 15, 21, 44, 77, 81, 84, 100]
L2:     [1, 7, 9, 13, 21, 21, 32, 44, 71, 81, 100]
inters: [1, 1, 7, 9, 9, 9, 9, 21, 44, 81, 100]


 ------ Second solution --------
L1:     [1, 1, 7, 9, 9, 9, 9, 12, 15, 21, 44, 77, 81, 84, 100]
L2:     [1, 7, 9, 13, 21, 21, 32, 44, 71, 81, 100]
inters: [1, 7, 9, 21, 44, 81, 100]
```

Functions

A function is a block of code that has a name and that performs a task.

A function can be thought of as a **box** (even as a black box: e.g. print()) that gets an **input** and returns an **output (or None)**.



The basic definition of a function is:

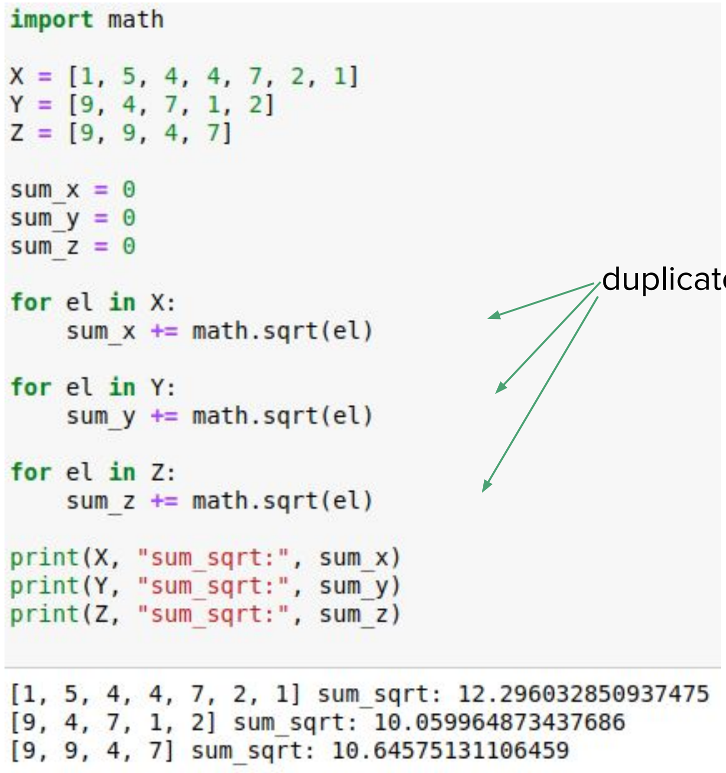
```
def function_name(input) :
    #code implementing the function
    ...
    ...
    return return_value
```

1. ***Reduce code duplication***: put in functions parts of code that are needed several times in the whole program so that you don't need to repeat the same code over and over again;

2. ***Decompose a complex task***: make the code easier to write and understand by splitting the whole program into several easier functions

Example: compute the sum of the square root of the values in lists X, Y, Z.

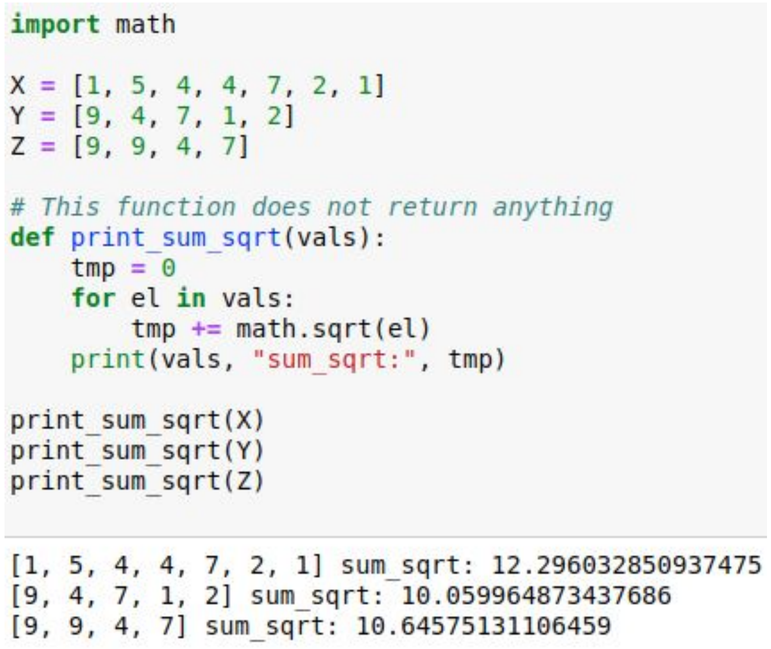
Functions



```python
import math

X = [1, 5, 4, 4, 7, 2, 1]
Y = [9, 4, 7, 1, 2]
Z = [9, 9, 4, 7]

sum_x = 0
sum_y = 0
sum_z = 0

for el in X:
    sum_x += math.sqrt(el)

for el in Y:
    sum_y += math.sqrt(el)

for el in Z:
    sum_z += math.sqrt(el)

print(X, "sum_sqrt:", sum_x)
print(Y, "sum_sqrt:", sum_y)
print(Z, "sum_sqrt:", sum_z)
```

```
[1, 5, 4, 4, 7, 2, 1] sum_sqrt: 12.296032850937475
[9, 4, 7, 1, 2] sum_sqrt: 10.059964873437686
[9, 9, 4, 7] sum_sqrt: 10.64575131106459
```

duplicated code

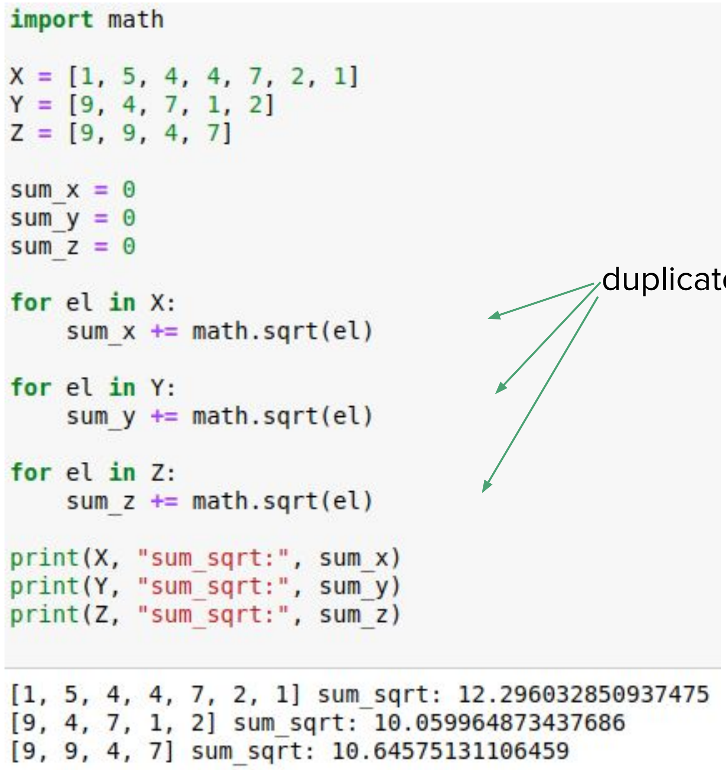


```python
import math

X = [1, 5, 4, 4, 7, 2, 1]
Y = [9, 4, 7, 1, 2]
Z = [9, 9, 4, 7]

# This function does not return anything
def print_sum_sqrt(vals):
    tmp = 0
    for el in vals:
        tmp += math.sqrt(el)
    print(vals, "sum_sqrt:", tmp)

print_sum_sqrt(X)
print_sum_sqrt(Y)
print_sum_sqrt(Z)
```

```
[1, 5, 4, 4, 7, 2, 1] sum_sqrt: 12.296032850937475
[9, 4, 7, 1, 2] sum_sqrt: 10.059964873437686
[9, 9, 4, 7] sum_sqrt: 10.64575131106459
```

Example: compute the sum of the square root of the values in lists X, Y, Z.

Functions

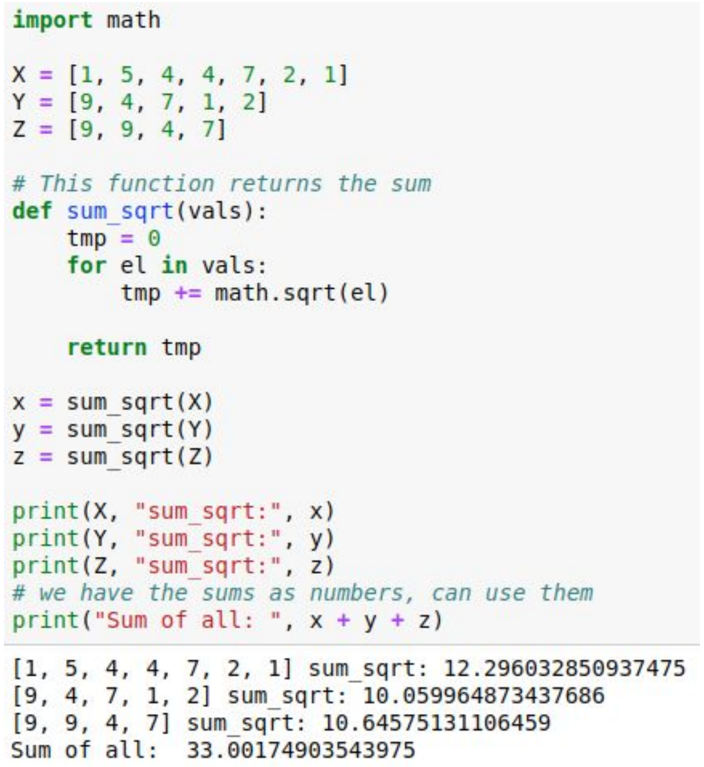


```python
import math

X = [1, 5, 4, 4, 7, 2, 1]
Y = [9, 4, 7, 1, 2]
Z = [9, 9, 4, 7]

sum_x = 0
sum_y = 0
sum_z = 0

for el in X:
    sum_x += math.sqrt(el)

for el in Y:
    sum_y += math.sqrt(el)

for el in Z:
    sum_z += math.sqrt(el)

print(X, "sum_sqrt:", sum_x)
print(Y, "sum_sqrt:", sum_y)
print(Z, "sum_sqrt:", sum_z)
```

```
[1, 5, 4, 4, 7, 2, 1] sum_sqrt: 12.296032850937475
[9, 4, 7, 1, 2] sum_sqrt: 10.059964873437686
[9, 9, 4, 7] sum_sqrt: 10.64575131106459
```

duplicated code

Another function returning the sum



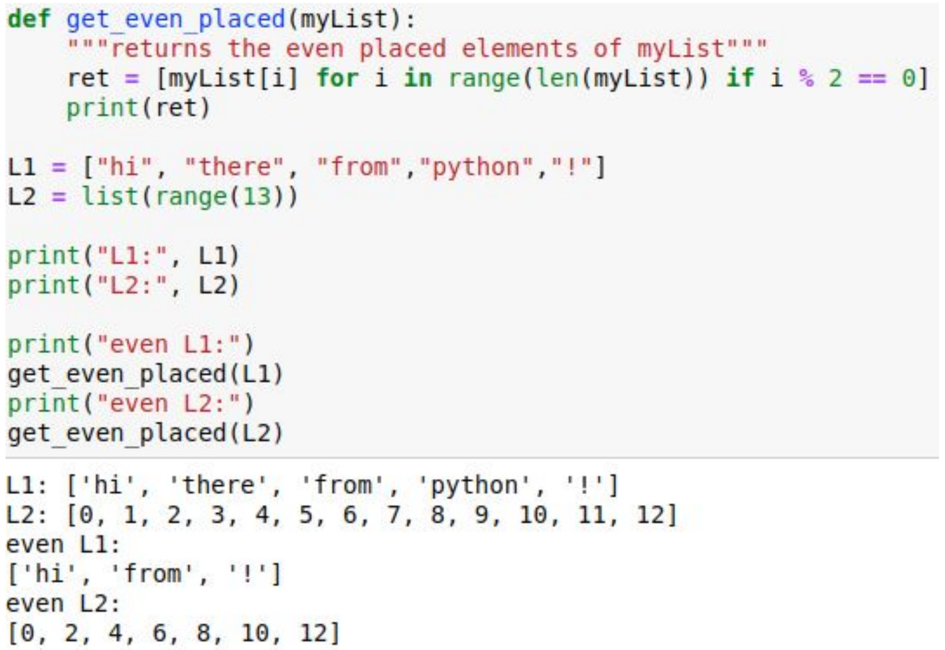
```python
import math

X = [1, 5, 4, 4, 7, 2, 1]
Y = [9, 4, 7, 1, 2]
Z = [9, 9, 4, 7]

# This function returns the sum
def sum_sqrt(vals):
    tmp = 0
    for el in vals:
        tmp += math.sqrt(el)

    return tmp

x = sum_sqrt(X)
y = sum_sqrt(Y)
z = sum_sqrt(Z)

print(X, "sum_sqrt:", x)
print(Y, "sum_sqrt:", y)
print(Z, "sum_sqrt:", z)
# we have the sums as numbers, can use them
print("Sum of all: ", x + y + z)
```

```
[1, 5, 4, 4, 7, 2, 1] sum_sqrt: 12.296032850937475
[9, 4, 7, 1, 2] sum_sqrt: 10.059964873437686
[9, 9, 4, 7] sum_sqrt: 10.64575131106459
Sum of all:  33.00174903543975
```

Functions

Example: Let's write a function that, given a list of elements, prints only the even-placed ones without returning anything.

This is a polymorphic function (i.e. it works on several data types, provided that we can iterate through them)**!**



```python
def get_even_placed(myList):
    """returns the even placed elements of myList"""
    ret = [myList[i] for i in range(len(myList)) if i % 2 == 0]
    print(ret)

L1 = ["hi", "there", "from","python","!"]
L2 = list(range(13))

print("L1:", L1)
print("L2:", L2)

print("even L1:")
get_even_placed(L1)
print("even L2:")
get_even_placed(L2)
```

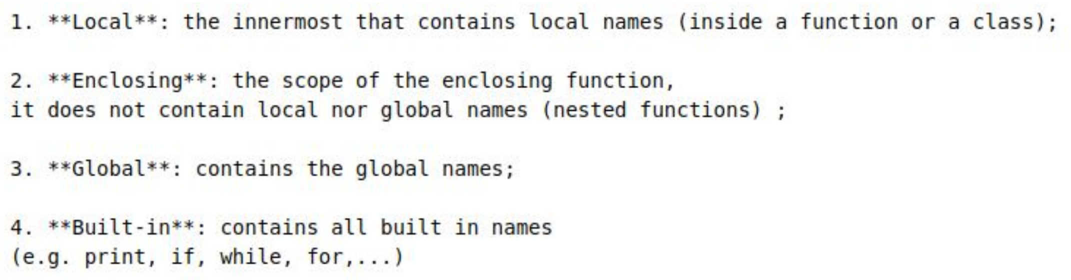
```
L1: ['hi', 'there', 'from', 'python', '!']
L2: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
even L1:
['hi', 'from', '!']
even L2:
[0, 2, 4, 6, 8, 10, 12]
```

Namespaces and scope

Namespaces are mappings from *names* to objects, or in other words places (i.e. dictionaries) where names are associated to objects.

Namespaces can be considered as **the context**.

According to Python’s reference a **scope** is a *textual region of a Python program, where a namespace is directly accessible*



```
1. **Local**: the innermost that contains local names (inside a function or a class);

2. **Enclosing**: the scope of the enclosing function,
it does not contain local nor global names (nested functions) ;

3. **Global**: contains the global names;

4. **Built-in**: contains all built in names
(e.g. print, if, while, for,...)
```

LEGB order for finding variable

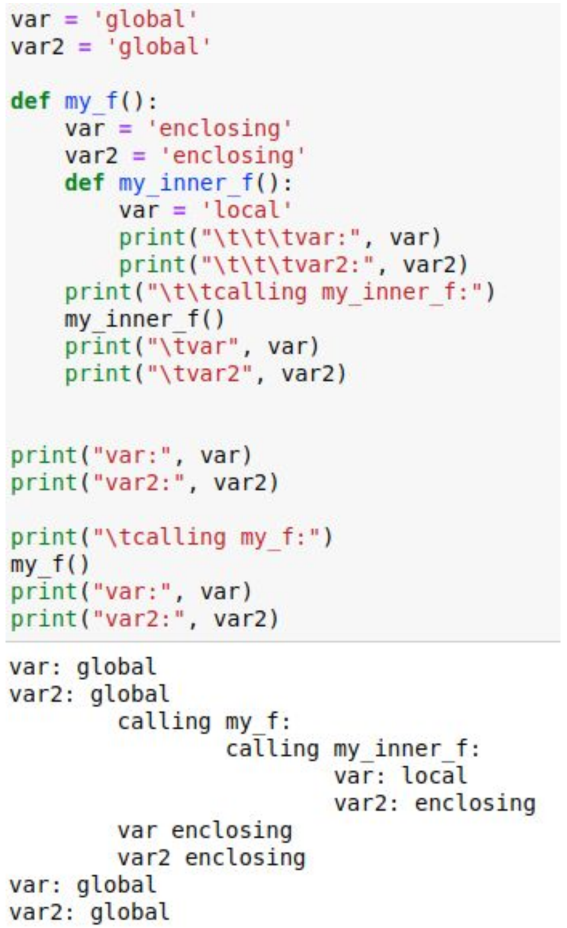
Namespace and scope

1. **Local**: the innermost that contains local names (inside a function or a class);

2. **Enclosing**: the scope of the enclosing function,
it does not contain local nor global names (nested functions) ;

3. **Global**: contains the global names;

4. **Built-in**: contains all built in names
(e.g. print, if, while, for,...)



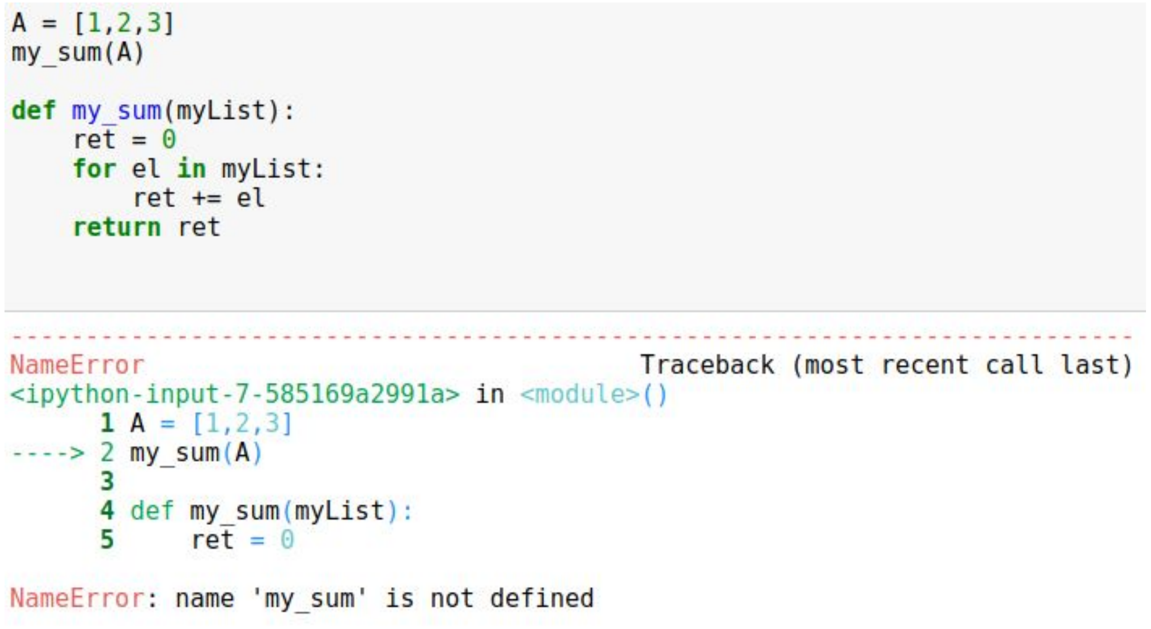
```python
var = 'global'
var2 = 'global'

def my_f():
    var = 'enclosing'
    var2 = 'enclosing'
    def my_inner_f():
        var = 'local'
        print("\t\t\tvar:", var)
        print("\t\t\tvar2:", var2)
    print("\t\tcalling my_inner_f:")
    my_inner_f()
    print("\tvar", var)
    print("\tvar2", var2)


print("var:", var)
print("var2:", var2)

print("\tcalling my_f:")
my_f()
print("var:", var)
print("var2:", var2)
```

```
var: global
var2: global
        calling my_f:
                calling my_inner_f:
                        var: local
                        var2: enclosing
        var enclosing
        var2 enclosing
var: global
var2: global
```

Functions

Example: define a function that gets a list of integers and returns its sum.

Importantly enough, **a function needs to be defined** (i.e. its code has to be written) **BEFORE** it can actually be used.

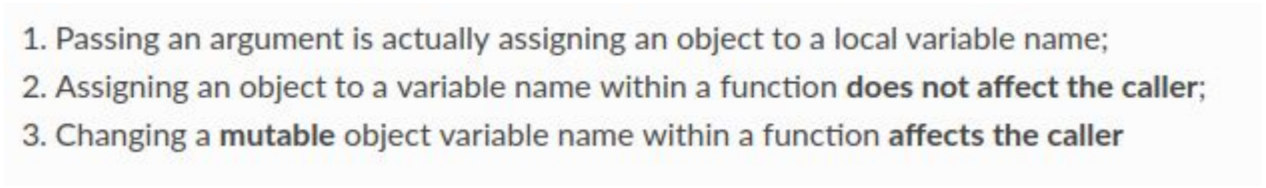


```
A = [1,2,3]
my_sum(A)

def my_sum(myList):
    ret = 0
    for el in myList:
        ret += el
    return ret
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-585169a2991a> in <module>()
      1 A = [1,2,3]
----> 2 my_sum(A)
      3 
      4 def my_sum(myList):
      5     ret = 0

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

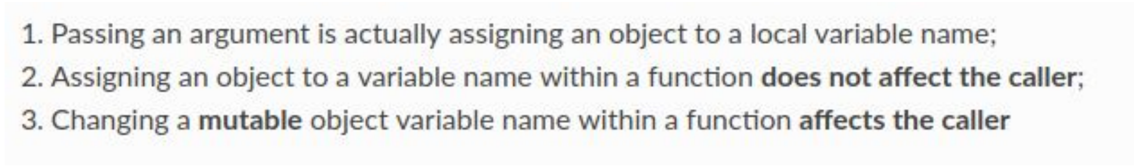
Argument passing

Things to remember

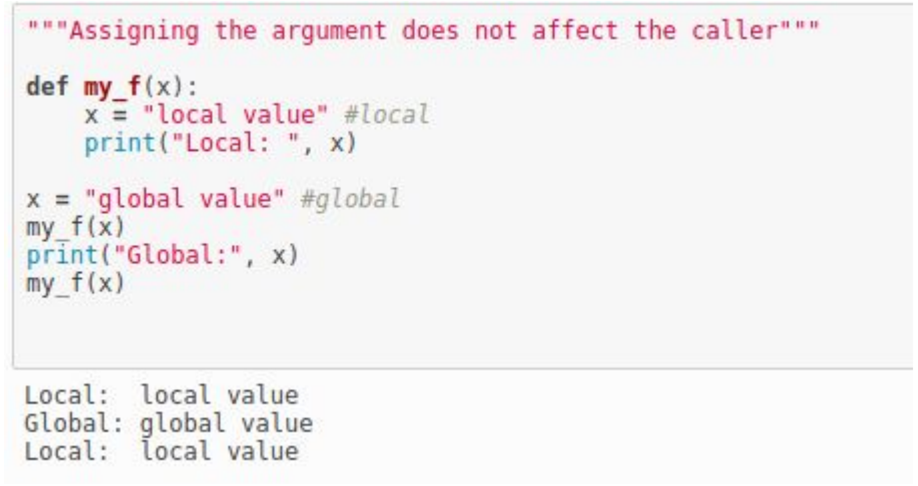


1. Passing an argument is actually assigning an object to a local variable name;
2. Assigning an object to a variable name within a function **does not affect the caller**;
3. Changing a **mutable** object variable name within a function **affects the caller**

Argument passing



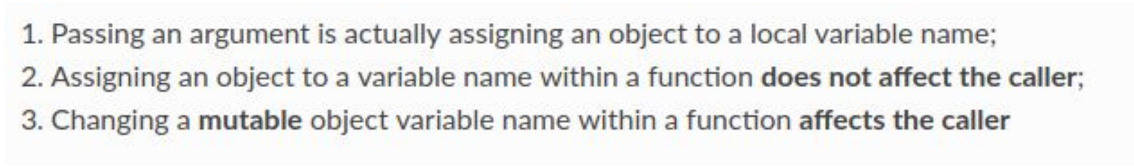
1. Passing an argument is actually assigning an object to a local variable name;
2. Assigning an object to a variable name within a function **does not affect the caller**;
3. Changing a **mutable** object variable name within a function **affects the caller**

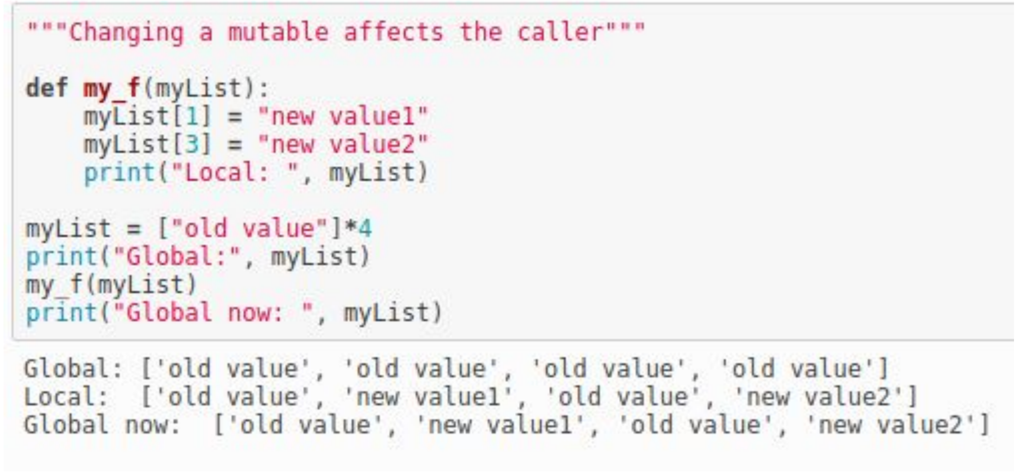


```python
"""Assigning the argument does not affect the caller"""

def my_f(x):
    x = "local value" #local
    print("Local: ", x)

x = "global value" #global
my_f(x)
print("Global:", x)
my_f(x)
```

```
Local:  local value
Global: global value
Local:  local value
```

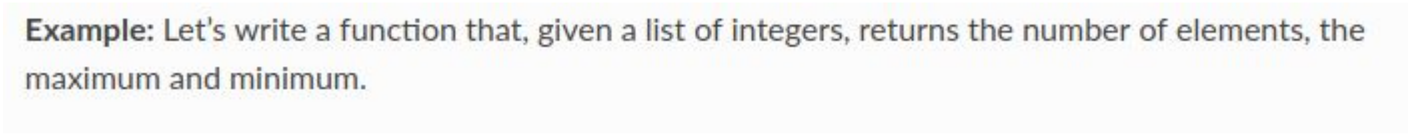
Argument passing

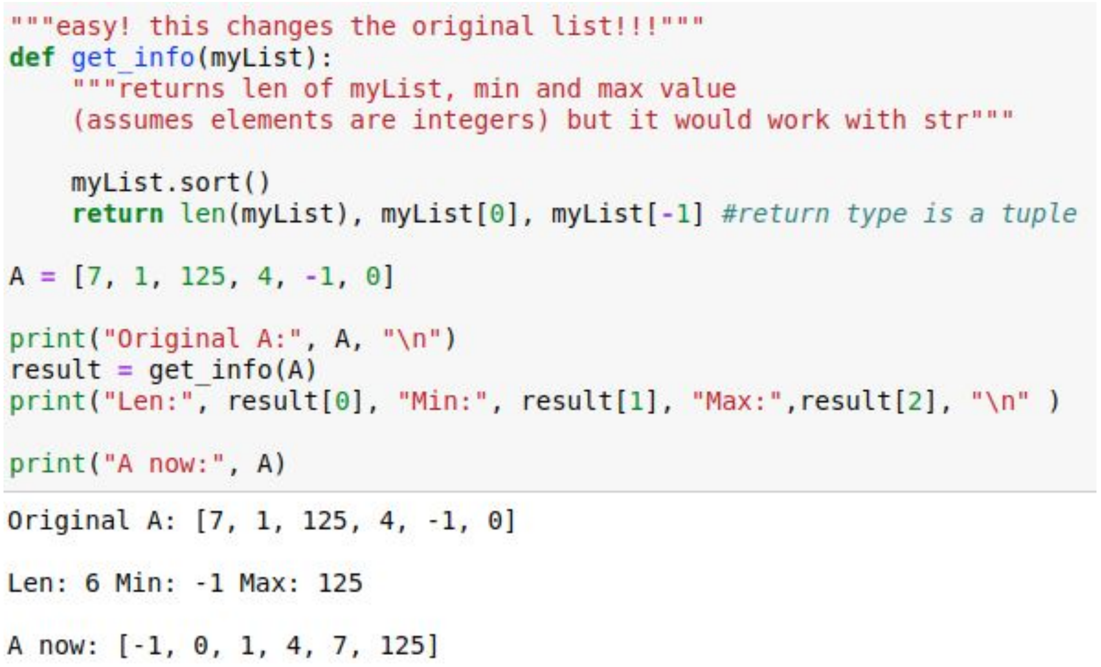

1. Passing an argument is actually assigning an object to a local variable name;
2. Assigning an object to a variable name within a function **does not affect the caller**;
3. Changing a **mutable** object variable name within a function **affects the caller**


```
"""Changing a mutable affects the caller"""

def my_f(myList):
    myList[1] = "new value1"
    myList[3] = "new value2"
    print("Local: ", myList)

myList = ["old value"]*4
print("Global:", myList)
my_f(myList)
print("Global now: ", myList)
```

```
Global: ['old value', 'old value', 'old value', 'old value']
Local:  ['old value', 'new value1', 'old value', 'new value2']
Global now:  ['old value', 'new value1', 'old value', 'new value2']
```

Functions

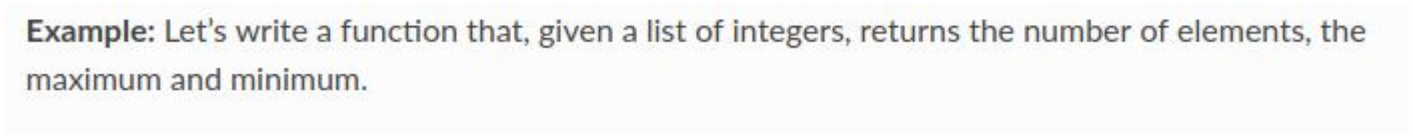

Example: Let's write a function that, given a list of integers, returns the number of elements, the maximum and minimum.


```python
"""easy! this changes the original list!!!"""
def get_info(myList):
    """returns len of myList, min and max value
    (assumes elements are integers) but it would work with str"""

    myList.sort()
    return len(myList), myList[0], myList[-1] #return type is a tuple

A = [7, 1, 125, 4, -1, 0]

print("Original A:", A, "\n")
result = get_info(A)
print("Len:", result[0], "Min:", result[1], "Max:",result[2], "\n" )

print("A now:", A)
```

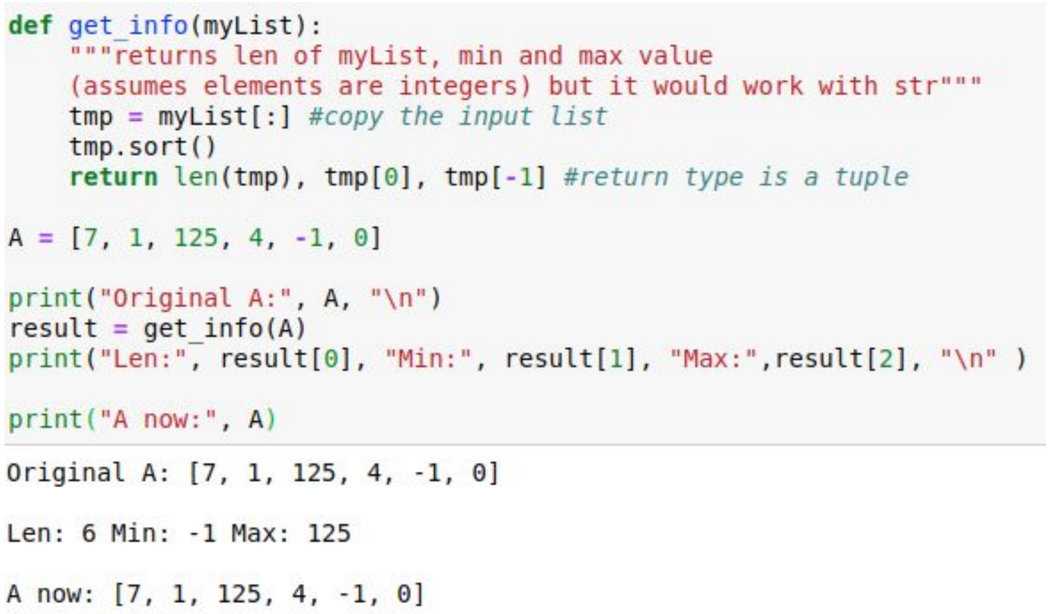
```
Original A: [7, 1, 125, 4, -1, 0]

Len: 6 Min: -1 Max: 125

A now: [-1, 0, 1, 4, 7, 125]
```

We need to **make a copy** if we want **to modify a mutable** within a function **without affecting the orginal object**

Functions



Example: Let's write a function that, given a list of integers, returns the number of elements, the maximum and minimum.



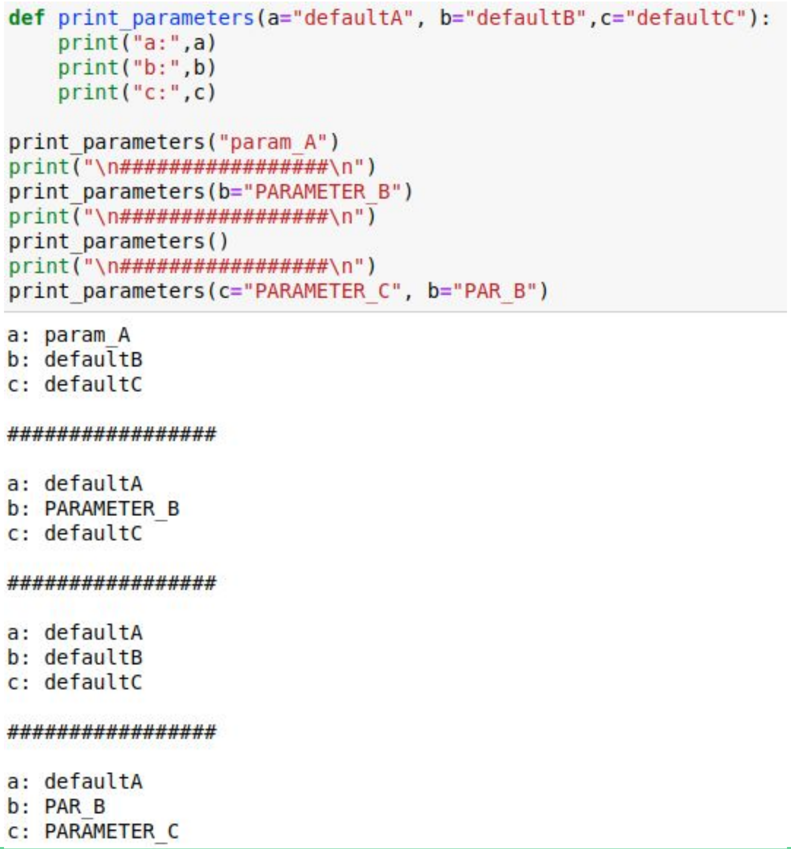
```
def get_info(myList):
    """returns len of myList, min and max value
    (assumes elements are integers) but it would work with str"""
    tmp = myList[:] #copy the input list
    tmp.sort()
    return len(tmp), tmp[0], tmp[-1] #return type is a tuple

A = [7, 1, 125, 4, -1, 0]

print("Original A:", A, "\n")
result = get_info(A)
print("Len:", result[0], "Min:", result[1], "Max:",result[2], "\n" )

print("A now:", A)
```

```
Original A: [7, 1, 125, 4, -1, 0]

Len: 6 Min: -1 Max: 125

A now: [7, 1, 125, 4, -1, 0]
```

We need to **make a copy** if we want **to modify a mutable** within a function **without affecting the orginal object**

Argument passing by keyword and defaults

We can specify default values (that can be overridden) and name the parameters of a function...

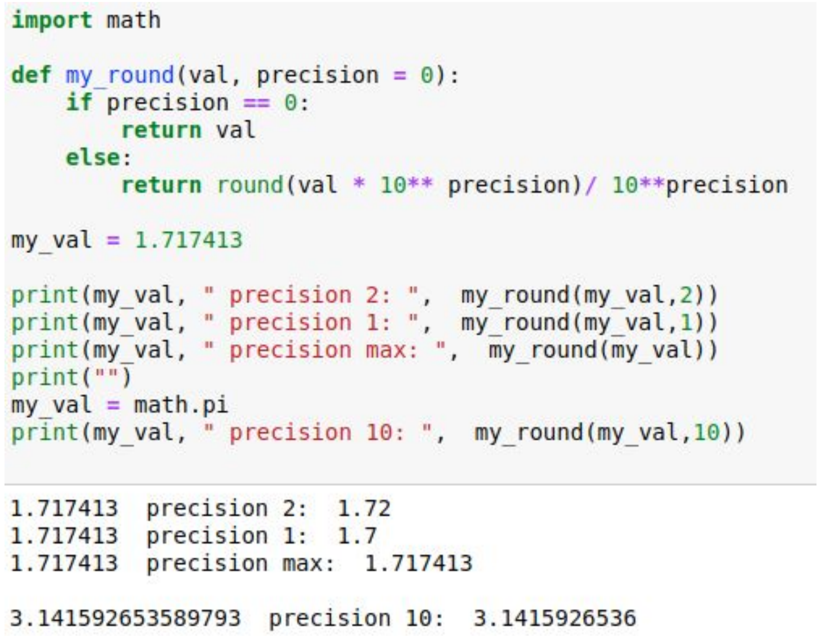


```python
def print_parameters(a="defaultA", b="defaultB",c="defaultC"):
    print("a:",a)
    print("b:",b)
    print("c:",c)

print_parameters("param_A")
print("\n#################\n")
print_parameters(b="PARAMETER_B")
print("\n#################\n")
print_parameters()
print("\n#################\n")
print_parameters(c="PARAMETER_C", b="PAR_B")
```

```
a: param_A
b: defaultB
c: defaultC

#################

a: defaultA
b: PARAMETER_B
c: defaultC

#################

a: defaultA
b: defaultB
c: defaultC

#################

a: defaultA
b: PAR_B
c: PARAMETER_C
```

Functions

Example. Write a function that rounds a float at a precision (i.e. number of decimals) specified in input. If no precision is specified then the whole number should be returned. Examples:
my_round(1.1717413, 3) = 1.172
my_round(1.1717413, 1) = 1.2
my_round(1.1717413) = 1.1717413

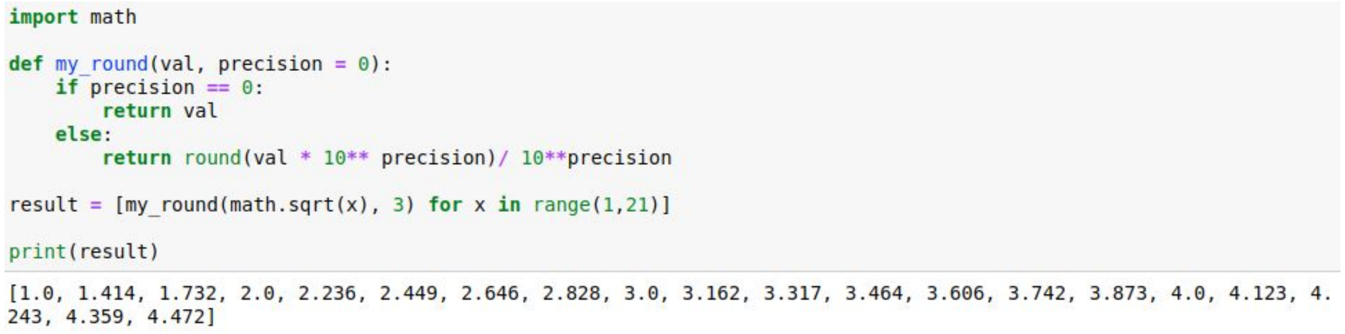


```python
import math

def my_round(val, precision = 0):
    if precision == 0:
        return val
    else:
        return round(val * 10** precision)/ 10**precision

my_val = 1.717413

print(my_val, " precision 2: ",  my_round(my_val,2))
print(my_val, " precision 1: ",  my_round(my_val,1))
print(my_val, " precision max: ",  my_round(my_val))
print("")
my_val = math.pi
print(my_val, " precision 10: ",  my_round(my_val,10))
```

```
1.717413  precision 2:  1.72
1.717413  precision 1:  1.7
1.717413  precision max:  1.717413

3.141592653589793  precision 10:  3.1415926536
```

Let’s create now a list with the square root values of the first 20 integers with 3 digits of precision.

Functions



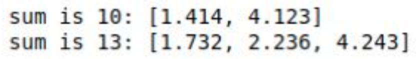
```
import math

def my_round(val, precision = 0):
    if precision == 0:
        return val
    else:
        return round(val * 10** precision)/ 10**precision

result = [my_round(math.sqrt(x), 3) for x in range(1,21)]

print(result)
```

```
[1.0, 1.414, 1.732, 2.0, 2.236, 2.449, 2.646, 2.828, 3.0, 3.162, 3.317, 3.464, 3.606, 3.742, 3.873, 4.0, 4.123, 4.
243, 4.359, 4.472]
```

we can apply functions in a list comprehension...

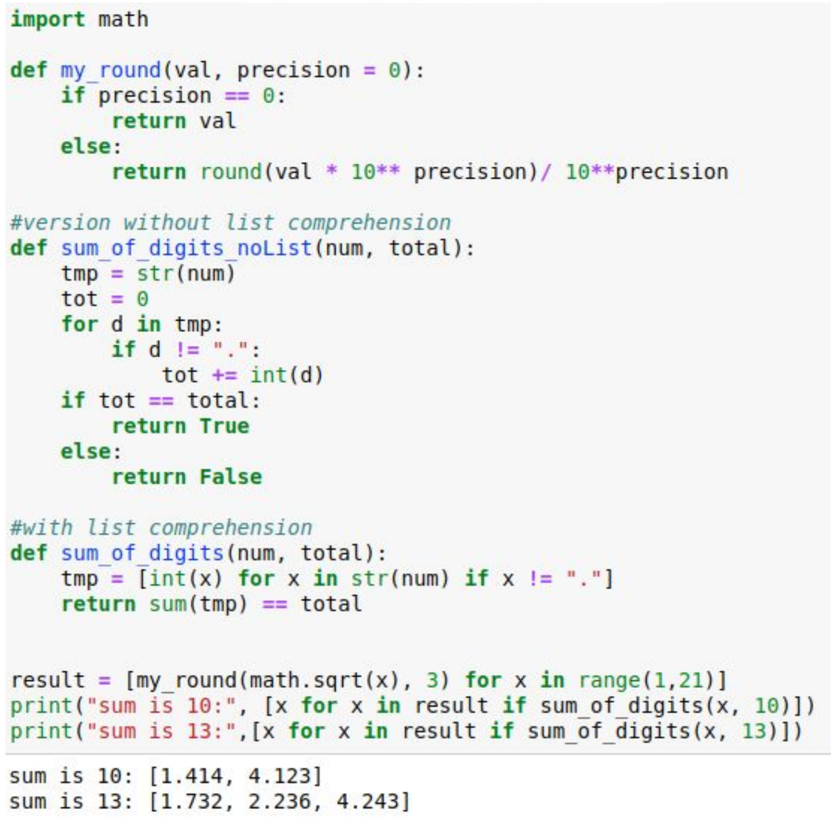
Functions

Let's print only the values of the list result above whose digits sum up to a certain value x. Hint: write another function!



```
sum is 10: [1.414, 4.123]
sum is 13: [1.732, 2.236, 4.243]
```

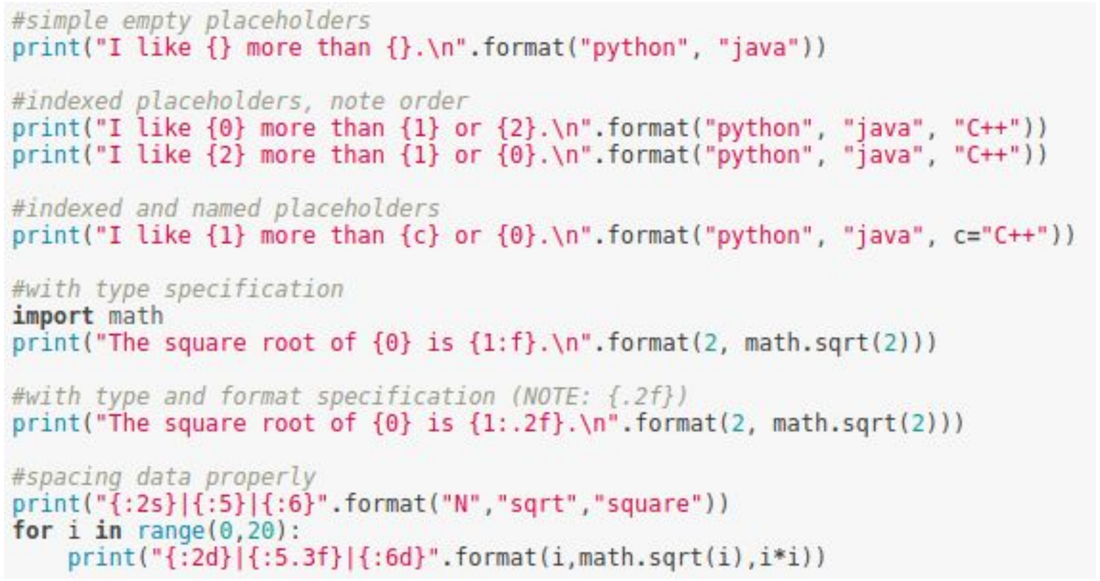
Another example... with and without list comprehension



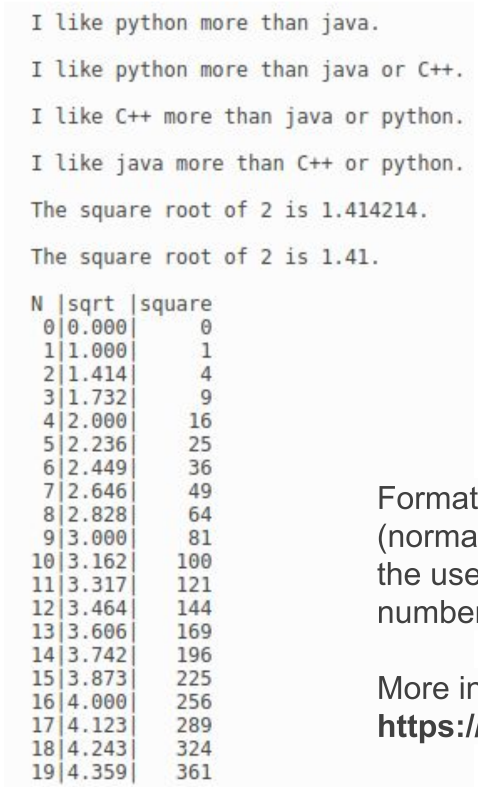
```python
import math

def my_round(val, precision = 0):
    if precision == 0:
        return val
    else:
        return round(val * 10** precision)/ 10**precision

#version without list comprehension
def sum_of_digits_noList(num, total):
    tmp = str(num)
    tot = 0
    for d in tmp:
        if d != ".":
            tot += int(d)
    if tot == total:
        return True
    else:
        return False

#with list comprehension
def sum_of_digits(num, total):
    tmp = [int(x) for x in str(num) if x != "."]
    return sum(tmp) == total


result = [my_round(math.sqrt(x), 3) for x in range(1,21)]
print("sum is 10:", [x for x in result if sum_of_digits(x, 10)])
print("sum is 13:",[x for x in result if sum_of_digits(x, 13)])
```

```
sum is 10: [1.414, 4.123]
sum is 13: [1.732, 2.236, 4.243]
```

String formatting



```python
#simple empty placeholders
print("I like {} more than {}.\n".format("python", "java"))

#indexed placeholders, note order
print("I like {0} more than {1} or {2}.\n".format("python", "java", "C++"))
print("I like {2} more than {1} or {0}.\n".format("python", "java", "C++"))

#indexed and named placeholders
print("I like {1} more than {c} or {0}.\n".format("python", "java", c="C++"))

#with type specification
import math
print("The square root of {0} is {1:f}.\n".format(2, math.sqrt(2)))

#with type and format specification (NOTE: {.2f})
print("The square root of {0} is {1:.2f}.\n".format(2, math.sqrt(2)))

#spacing data properly
print("{:2s}|{:5}|{:6}".format("N","sqrt","square"))
for i in range(0,20):
    print("{:2d}|{:5.3f}|{:6d}".format(i,math.sqrt(i),i*i))
```



```
I like python more than java.

I like python more than java or C++.

I like C++ more than java or python.

I like java more than C++ or python.

The square root of 2 is 1.414214.

The square root of 2 is 1.41.

N |sqrt |square
 0|0.000|     0
 1|1.000|     1
 2|1.414|     4
 3|1.732|     9
 4|2.000|    16
 5|2.236|    25
 6|2.449|    36
 7|2.646|    49
 8|2.828|    64
 9|3.000|    81
10|3.162|   100
11|3.317|   121
12|3.464|   144
13|3.606|   169
14|3.742|   196
15|3.873|   225
16|4.000|   256
17|4.123|   289
18|4.243|   324
19|4.359|   361
```

Format can be used to add values to a string in specific placeholders (normally defined with the syntax **{}**) or to format values according to the user specifications (e.g. number of decimal places for floating point numbers).

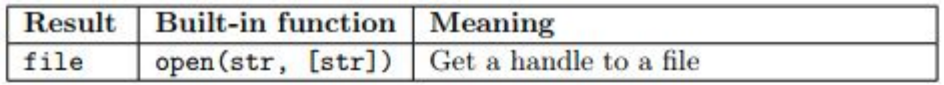
More info
https://docs.python.org/3/library/string.html#format-string-syntax

File Input/Output

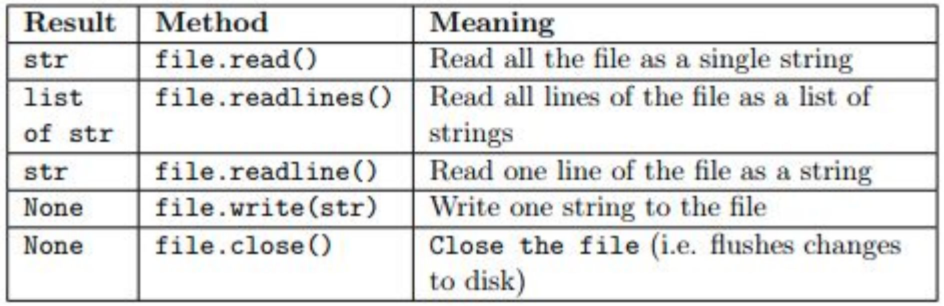
In the next practical!

With files you need to perform 3 steps:

Open the file**, read/write, close**



Result	Built-in function	Meaning
`file`	`open(str, [str])`	Get a handle to a file



Result	Method	Meaning
`str`	`file.read()`	Read all the file as a single string
`list of str`	`file.readlines()`	Read all lines of the file as a list of strings
`str`	`file.readline()`	Read one line of the file as a string
`None`	`file.write(str)`	Write one string to the file
`None`	`file.close()`	`Close the file` (i.e. flushes changes to disk)

File Input/Output

```
file_handle = open("file_name", "file_mode")
```

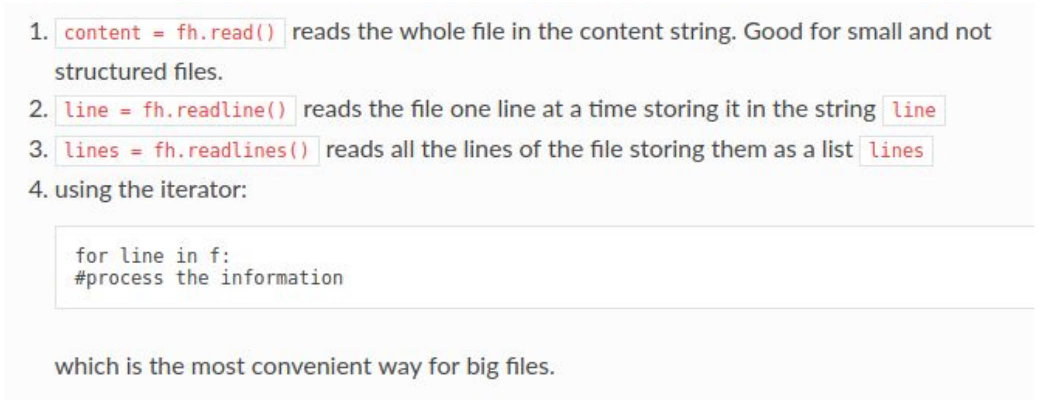
With files you need to:

Open, read/write, close

Opening mode: “r”, “w”, “a”,”b”,...

overwrites!

Read



1. `content = fh.read()` reads the whole file in the content string. Good for small and not structured files.
2. `line = fh.readline()` reads the file one line at a time storing it in the string `line`
3. `lines = fh.readlines()` reads all the lines of the file storing them as a list `lines`
4. using the iterator:

```
for line in f:
#process the information
```

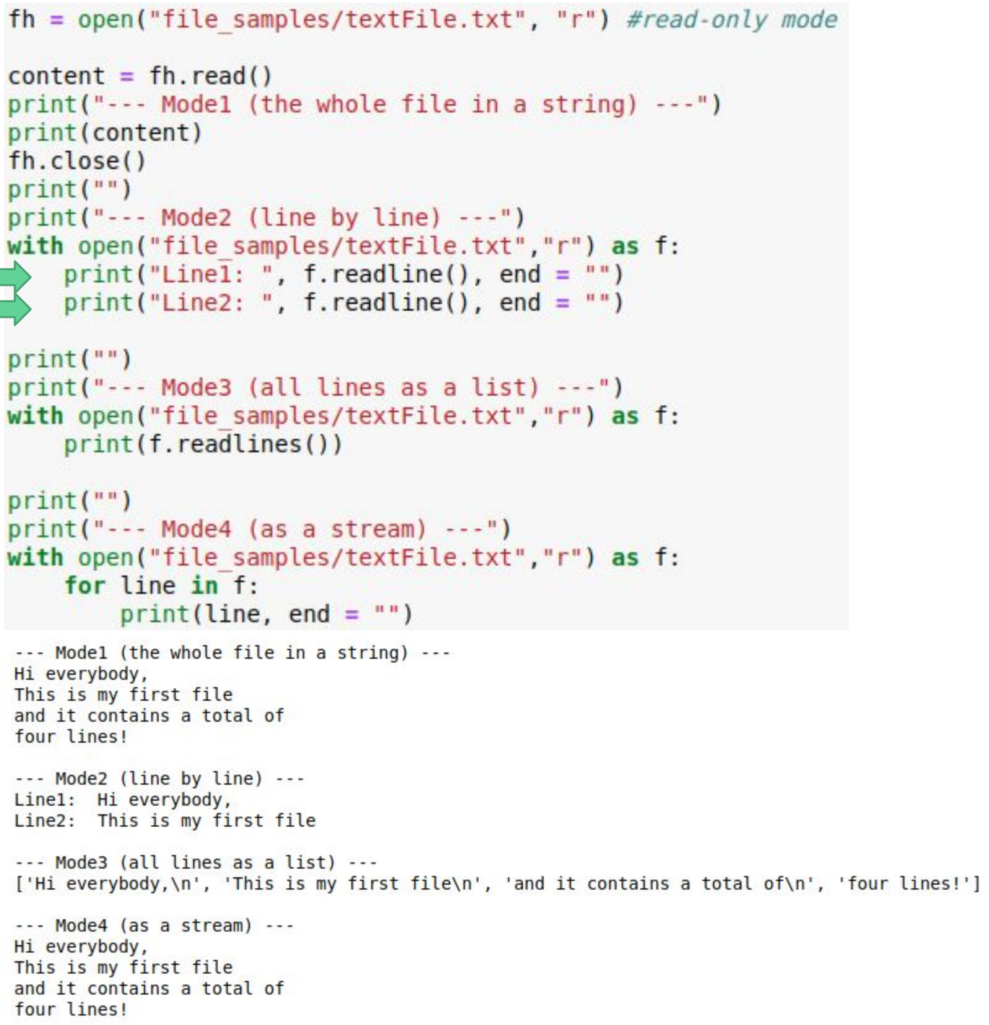
which is the most convenient way for big files.

Write

```
file_handle.write(data_to_be_written)
```

```
file_handle.close()
```

File Input/Output



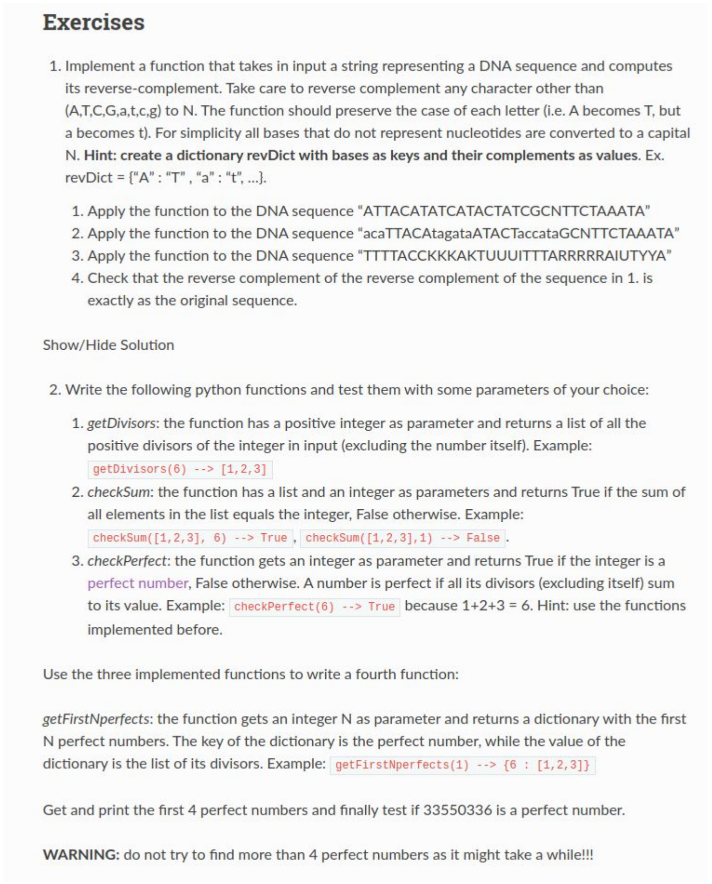
```python
fh = open("file_samples/textFile.txt", "r") #read-only mode

content = fh.read()
print("--- Mode1 (the whole file in a string) ---")
print(content)
fh.close()
print("")
print("--- Mode2 (line by line) ---")
with open("file_samples/textFile.txt","r") as f:
    print("Line1: ", f.readline(), end = "")
    print("Line2: ", f.readline(), end = "")

print("")
print("--- Mode3 (all lines as a list) ---")
with open("file_samples/textFile.txt","r") as f:
    print(f.readlines())

print("")
print("--- Mode4 (as a stream) ---")
with open("file_samples/textFile.txt","r") as f:
    for line in f:
        print(line, end = "")
```

```
--- Mode1 (the whole file in a string) ---
Hi everybody,
This is my first file
and it contains a total of
four lines!

--- Mode2 (line by line) ---
Line1:  Hi everybody,
Line2:  This is my first file

--- Mode3 (all lines as a list) ---
['Hi everybody,\n', 'This is my first file\n', 'and it contains a total of\n', 'four lines!']

--- Mode4 (as a stream) ---
Hi everybody,
This is my first file
and it contains a total of
four lines!
```

more info in the
Practical6 notes...

http://qcbsciprolab2020.readthedocs.io/en/latest/practical6.html

?



Exercises

1. Implement a function that takes in input a string representing a DNA sequence and computes its reverse-complement. Take care to reverse complement any character other than (A,T,C,G,a,t,c,g) to N. The function should preserve the case of each letter (i.e. A becomes T, but a becomes t). For simplicity all bases that do not represent nucleotides are converted to a capital N. **Hint: create a dictionary revDict with bases as keys and their complements as values**. Ex. revDict = {“A” : “T” , “a” : “t”, ...}.

    1. Apply the function to the DNA sequence “ATTACATATCATACTATCGCNTTCTAAATA”
    2. Apply the function to the DNA sequence “acaTTACAtagataATACTaccataGCNTTCTAAATA”
    3. Apply the function to the DNA sequence “TTTTACCKKKAKTUUUITTTARRRRRAIUTYYA”
    4. Check that the reverse complement of the reverse complement of the sequence in 1. is exactly as the original sequence.

Show/Hide Solution

2. Write the following python functions and test them with some parameters of your choice:

    1. *getDivisors*: the function has a positive integer as parameter and returns a list of all the positive divisors of the integer in input (excluding the number itself). Example: `getDivisors(6) --> [1,2,3]`
    2. *checkSum*: the function has a list and an integer as parameters and returns True if the sum of all elements in the list equals the integer, False otherwise. Example: `checkSum([1,2,3], 6) --> True`, `checkSum([1,2,3],1) --> False`.
    3. *checkPerfect*: the function gets an integer as parameter and returns True if the integer is a perfect number, False otherwise. A number is perfect if all its divisors (excluding itself) sum to its value. Example: `checkPerfect(6) --> True` because 1+2+3 = 6. Hint: use the functions implemented before.

Use the three implemented functions to write a fourth function:

getFirstNperfects: the function gets an integer N as parameter and returns a dictionary with the first N perfect numbers. The key of the dictionary is the perfect number, while the value of the dictionary is the list of its divisors. Example: `getFirstNperfects(1) --> {6 : [1,2,3]}`

Get and print the first 4 perfect numbers and finally test if 33550336 is a perfect number.

WARNING: do not try to find more than 4 perfect numbers as it might take a while!!!