Complementary exercises

Please Note: This assignment is **not compulsory** and will **not be graded**.The answers will be released later for your reference.

Question 1: About function [put your code in Question_1.py]

A **function** is a named sequence of statements that performs a specific task or useful operation.

In Python there are a few built-in functions such as

- **print** - outputs value to screen; returns nothing
- **input** - prompts user for input; returns a str
- **type** - returns the type of the value passed in; returns a type object
- **round** - rounds a number; returns an int (when called with one argument) …(we used this in a homework)
- **abs** - absolute value; returns a numeric type

Besides, you can create your own functions. To define a function, you need to use the keyword `def`, the syntax examples are as follows,

```
# without parameters (inputs)
def the_name_of_your_function():
    # some code
    print("do some useful stuff")

# with parameters (inputs)
def the_name_of_your_function(parameter_1, paramter_2):
    # some code
    print("do some useful stuff with parameter_1 and parameter_2")
```

Remember in Python the function body should have **a four-space indentation**.

A function can have a **return** value, The `return` statement in a function means to return immediately from this function and use the following expression as a return value. Moreover, a function can have multiple return statements, for example, we can define our own abs function like the following,

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

Since the `return` statements are in an alternative conditional, only one will be executed. As soon as one is executed, the function terminates without executing any subsequent statement. We can leave the `else` and just follow the if condition by the second `return` statement.

```
def absolute_value(x):
    if x < 0:
        return -x
    return x
```

1.1. Recall that the equation for a line can be written as y = mx + c, where m is the slope of the line and c is the y-intercept. For a line that passes through two points, (x_1, y_1) and (x_2, y_2), we have the following identities:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$
$$c = y_1 - mx_1$$

(a) Write a function `slope(x1, y1, x2, y2)` that returns the slope of the line through the points (x1, y1) and (x2, y2). [Hint: Your code need to handle the corner case.]

(b) Write a function `intercept(x1, y1, x2, y2)` that returns the y-intercept of the line through the points (x1, y1) and (x2, y2). [Hint: you may use the function you defined in (a)].

Question 2: About list and dict [put your code in Question_2.py]

- Python list and dict type data are mutable.

- To remove/insert some elements in a list in-place with loops, you need to start from the last element of the list

You are given the following list and dictionary:

```
l = [1, 1, 2, 2.5]
d = {1:3, 2:5.5, 6:2.0, 4:2.0}
```

2.1. Write a function `remove_non_int(l)`, that takes a list as input and gets rid of all the elements that are not integers. Use type to check (e.g. `type(5)== int` will return `True`). After we pass the list `l` to the `remove_non_int(l)` function, list l will become `[1,1,2]`. The following is an example (NOTE: the function has no return value, i.e., the modification should be in-place, without using any auxiliary list.):

```
>>> l = [1, 1, 2, 2.5]
>>> remove_non_int(l)
>>> l
[1, 1, 2]
```

2.2. Write a function `delete_duplicate(l)`, which takes a list and deletes duplicate elements from the list. You are **NOT** allowed to use "set" functions for this question. Examples (NOTE: the function has no return value, i.e., the modification should be in-place, without using any auxiliary list.):

```
>>> l = [1, 1, 2, 2.5]
>>> delete_duplicate(l)
>>> l
[1, 2, 2.5]
>>> l = [1, 0, 1, 1, 0, 2, 2.5]
>>> delete_duplicate(l)
>>> l
[1, 0, 2, 2.5]
```

2.3. Write a program `do_all(d)` that takes out all the values (not keys) in a dictionary, put them in a list. Then 1) get rid of all the duplicates, 2) remove the non integers, 3) return the list. Use the functions that you created in the previous 2 steps. For example, for the given dictionary `d ={1:3, 2:5.5, 6:2.0, 4:2.0}`, the list `[3]` will be returned. Example:

```
>>> d = {1:3, 2:5.5, 6:2.0, 4:2.0}
>>> do_all(d)
[3]
```

Question 3: About the recursion [put your code in Question_3.py]

3.1. Write a **recursive** function `maximum()` in `recursion_student.py` that takes a parameter representing a list `A` and returns the maximum in the list. (Note: you are **not** allowed to use the built-in function `max()`)

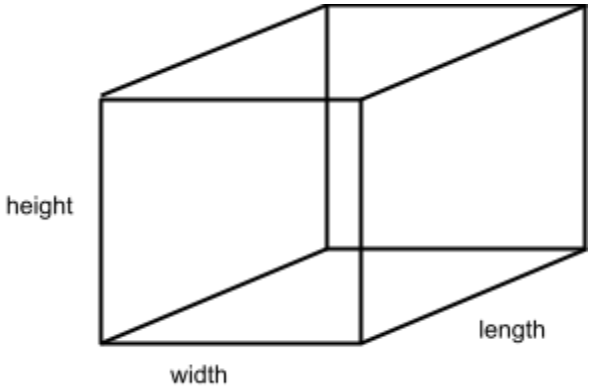
Hint: think recursively, the maximum is either the first value in list `A`, or the maximum of the rest of the list, whichever is larger. If the list has one integer, then naturally, its maximum is this single value. Also, "the rest of the list" above is simply `A[1:len(A)]`

Question 4: Exercise for OOP in Python [put your code in Question_4.py]

1. Write a simple `Cuboid` class with the following attributes and methods:

Attributes:

- `length, width, height` (all three are integers), and `color`



Methods (in the following table):

`__init__`	An initializer `__init__` which sets the Cuboid instance's `length`, `width`, `height`, and `color` from user-provided arguments.
	`In [2]: c = Cuboid(1, 2, 3, "blue")`
	
	`In [3]: c.length`
	`Out[3]: 1`
	
	`In [4]: c.width`
	`Out[4]: 2`
	
	`In [5]: c.height`
	`Out[5]: 3`
	
	`In [6]: c.color`
	`Out[6]: 'blue'`

`get_attribute`	It is a getter and returns the value of a specified attribute. (Note: it needs an input which is a string, i.e., the name of a specified attribute.)
	`In [7]: c.get_attribute("length")`
	`Out[7]: 1`
	
	`In [8]: c.get_attribute("width")`
	`Out[8]: 2`
	
	`In [9]: c.get_attribute("height")`
	`Out[9]: 3`
	
	`In [10]: c.get_attribute("color")`
	`Out[10]: 'blue'`

`set_attribute`	It is a setter and sets a specified attribute as the given value. In the following example, it sets the length of `c` to be 10 and the color of `c` to be red.
	`In [11]: c.set_attribute("length", 10)`
	
	`In [12]: c.set_attribute("color", "red")`

`get_area`	It returns the cuboid's surface area (reminder: a cuboid has 6 facets). For a cuboid with length = 10, width = 2 and height = 3. It returns 112.
	`In [17]: c.get_area()`
	`Out[17]: 112`

`get_volume`	It returns the cuboid's volume (length*width*height). For a cuboid with length = 10, width = 2 and height = 3. It returns 60.
	`In [18]: c.get_volume()`
	`Out[18]: 60`

`__str__`	Overriding the built-in `__str__` method so that it returns a string of the cuboid's information: length, width, height, color, area and volume. When printing the cuboid instance `c`, the printed message on the console should look like the following example, (For this method, the code is given)
	`In [19]: print(c)`
	`Length: 10, Width: 2, Height: 3, Color: red, Area: 112, Volume: 60`

`is_equal`	It compares the attributes of the attributes with those of another cuboid. If all attributes are the same, return True; otherwise, return False. (Note: for values of `color`, the letters in upper cases and lower cases are treated as the same.)
	`In [33]: c1 = Cuboid(1, 2, 3, "blue")`
	`In [34]: c2 = Cuboid(2, 3, 4, "blue")`
	`In [35]: c3 = Cuboid(1, 2, 3, "BLUE")`
	`In [36]: c1.is_equal(c2)`
	`Out[36]: False`
	`In [37]: c1.is_equal(c3)`
	`Out[37]: True`

2. Write a `CuboidList` class to store different cuboids. Generally, CuboidList stores a list of cuboids in a dictionary d. It also has a few methods to manipulate the cuboids it stores.

The CuboidList class has the following attributes and methods.

Attributes:

- d: a Python dictionary. The keys of d are colors and the values are lists of cuboid instances of the corresponding colors.

Methods:

The examples are based on the following list:

```
l = [Cuboid(1, 2, 3, "blue"), Cuboid(4, 2, 3, "blue"),
    Cuboid(1, 2, 3, "red"), Cuboid(2, 2, 3, "orange"),
    Cuboid(3, 2, 4, "red"), Cuboid(5, 4, 3, "blue"),
    Cuboid(3, 3, 3, "blue")]
```

`__init__`	An initializer `__init__` which sets the instance's attribute `d` from a user-provided list of cuboids. The keys of `d` are colors and the values are lists of cuboid instances of the corresponding colors.
	`In [40]: cList = CuboidList(l)`
	`In [41]: cList.d`
	`Out[41]:`
	`{'blue': [<__main__.Cuboid at 0x7fbab8fb2ca0>,`
	`<__main__.Cuboid at 0x7fbab8fb2c40>,`
	`<__main__.Cuboid at 0x7fbab8fb2130>,`
	`<__main__.Cuboid at 0x7fbab9045460>],`
	`'red': [<__main__.Cuboid at 0x7fbab8fb2e20>,`
	`<__main__.Cuboid at 0x7fbab8fb21c0>],`
	`'orange': [<__main__.Cuboid at 0x7fbab8fb2d30>]}`

`remove_cuboid`	It removes the specified cuboid from the `CuboidList`. In the following example, the cuboids that are the same to `c` are removed from the `d` of the `cList`. (reminder: in `d`, the values are lists.)
	`In [45]: cList = CuboidList(l)`
	`In [46]: c = Cuboid(1, 2, 3, "blue")`
	`In [47]: cList.remove_cuboid(c)`

`get_cuboids`	It returns a list of cuboids whose values of the specified attribute are the same as the given value. In the following example, it returns a list with each element as a cuboid whose height is 4.
	`In [62]: c_height4 = cList.get_cuboids("height", 4)`
	`In [63]: c_height4`
	`Out[63]: [<__main__.Cuboid at 0x7fbaaa212b20>, <__main__.Cuboid at 0x7fbaaa212a60>]`
	`In [64]: print(c_height4[0])`
	`Length: 5, Width: 4, Height: 4, Color: blue, Area: 112, Volume: 80`
	`In [65]: print(c_height4[1])`
	`Length: 3, Width: 2, Height: 4, Color: red, Area: 52, Volume: 24`

`bubble_sort`	It returns a list of cuboid instances that are sorted by the specified attribute in ascending order. You have to use the bubble sort algorithm for this task. Please modify the code of the bubble sort given in the class so that it can sort the cuboids on a specified attribute. (Note: you are not allowed to use any built-in sort methods such as `sorted()`.)
	`In [74]: c_sorted = cList.bubble_sort("length")`
	`In [75]: for c in c_sorted:`
	`...:  print(c)`
	`...:`
	`Length: 1, Width: 2, Height: 3, Color: blue, Area: 22, Volume: 6`
	`Length: 1, Width: 2, Height: 3, Color: red, Area: 22, Volume: 6`
	`Length: 2, Width: 2, Height: 3, Color: orange, Area: 32, Volume: 12`
	`Length: 3, Width: 2, Height: 4, Color: red, Area: 52, Volume: 24`
	`Length: 3, Width: 3, Height: 3, Color: blue, Area: 54, Volume: 27`
	`Length: 4, Width: 2, Height: 3, Color: blue, Area: 52, Volume: 24`
	`Length: 5, Width: 4, Height: 4, Color: blue, Area: 112, Volume: 80`

`sort_bonus`	It is a **bonus question**. A Python list has a built-in method `sort()` which sorts the elements in the list. When the elements are cuboids we defined, we can still sort them using `.sort()`. To do this, you need to specify the argument, namely, `key`, of the `sort()`. `key` is a function. When it is given, the `sort()` will sort the elements based on the return value of it. Use `help(list.sort)` to see the details of the `key`. Defining a lambda function for the `key` so that this method returns a list of sorted cuboids based on the specified attribute.
	
	Hint: A lambda function is a small anonymous function. It takes any number of arguments and has one expression. To define a lambda function, the syntax is as the following,
	`lambda arguments: expression`
	where `lambda` is the keyword for claiming a lambda function.
	
	`In [79]: c_sorted = cList.sort_bonus("length")`
	`In [80]: for c in c_sorted:`
	`...:  print(c)`
	`...:`
	`Length: 1, Width: 2, Height: 3, Color: blue, Area: 22, Volume: 6`
	`Length: 1, Width: 2, Height: 3, Color: red, Area: 22, Volume: 6`
	`Length: 2, Width: 2, Height: 3, Color: orange, Area: 32, Volume: 12`
	`Length: 3, Width: 3, Height: 3, Color: blue, Area: 54, Volume: 27`
	`Length: 3, Width: 2, Height: 4, Color: red, Area: 52, Volume: 24`
	`Length: 4, Width: 2, Height: 3, Color: blue, Area: 52, Volume: 24`
	`Length: 5, Width: 4, Height: 4, Color: blue, Area: 112, Volume: 80`

Question 5: evaluate the complexity using big-O notation

```
def one_iter_nop(in_list):
""" in_list contains a list of integers """
      sum = 0
      return sum
      for x in in_list:
            sum += x


L = list(range(N))
one_iter_nop(L)
```

Answer: O(1)