Complementary Assignment

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

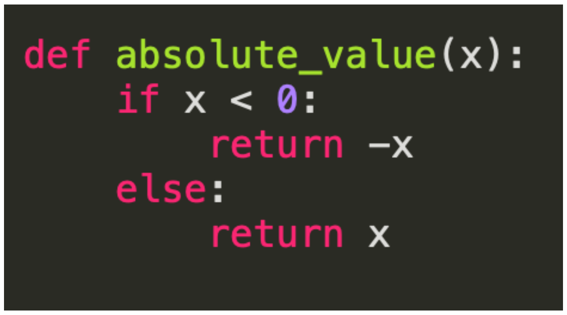
Besides, you can create your own functions. To define a function, you need to use the keyword `def`, the syntax examples are as follows,

```
# 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")
```

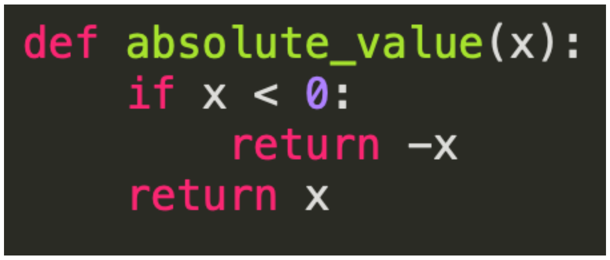
Remember in Python the function body should have a four-space indentation.

A function can have a **return** value, The `return` statement in a function means 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.



```python
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:

$$\begin{aligned} m &= \frac{y_2 - y_1}{x_2 - x_1} \\ c &= y_1 - mx_1 \end{aligned}$$

(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: A taste of functional programming [put your code in Quiestion_4.py]

In Python, a function can be an argument to another function. The following code snippet takes Python's built-in function `print` and passed it as an argument:

```
>>> def my_print(p, a_str):
        p('hello {}'.format(a_str))


>>> my_print(print, "ICS")
hello ICS
```

4.1 Write a function `order(p, q, n)` that takes two multi-element tuples `p` and `q`, and compares the element at index `n`, returns True if `p[n] > q[n]`. Examples:

```
>>> order((1, 2, 3), (2, 1, 4), 0)
False
>>> order((1, 2, 3), (2, 1, 4), 1)
True
>>> order((1, 2, 3), (2, 1, 4), 2)
False
```

4.2 Use `order` to implement a function `first_max(order_f, l, n)`. `l` is a list of tuples, and the function returns the ***first largest*** tuple in `l`. The comparison of order is done using `order` you implemented in B.1 (i.e. a tuple `p` is larger than `q` if `p[n] > q[n]`).

Example -- `('B', 6)`, `('X', 6)` and `('P', 6)` are all the largest on index 1, but `('B', 6)` is the correct result because it is the *first largest*:

```
>>> tuplst = [('X', 5), ('B', 6), ('P', 4), ('X', 3), ('B', 5),('P', 6)]
>>> print(first_max(order, tuplst, 1))
('B', 6)
```

4.3 Implement a function `last_max(order_f, l, n)`, this time returns the ***last*** one. Constraints: 1) you can *not* invoke `first_max` on a reversed list, 2) you ***must not*** modify the order function, and you can ***only*** use `order_f` to test order of two tuples.

Example: this time `('P', 6)` is the *last largest*

```
>>> tuplst = [('X', 5), ('B', 6), ('P', 4), ('X', 3), ('B', 5),('P', 6)]
>>> print(last_max(order, tuplst, 1))
('P', 6)
```