Lab 1

Before you start, keep in mind the following:

- You will be working on a platform called Google Colab notebook, a single environment where you can type texts, Python codes, and execute the codes. All you need to get access to Google Colab is a Google account. Read this instruction to open a Google Colab notebook. This video shows you how to make and submit a sample report.
- Google Colab notebook automatically saves every 30 seconds, so you don't have to worry about losing your work. After you close a session by closing the tab or web browser, all memory about the session gets reset. You will need to rerun (not retype) all the command after opening a new session.
- There are two ways to write/insert comments on Google Colab notebook.
  - In a *text cell*: anything written in a text cell is treated as text, not as Python code. This way is suitable if your comment is long or if it contains math formulas.
  - In a *code cell*: Python ignores everything on the line after the #, and hence it allows you to put instructions and explanations in your code to make it easier for others to read and understand.
- The next lab will be built on this lab, so please read all the instruction carefully and do all the assignments.

In this lab, you will learn how to:

- do arithmetic on Python,
- view outputs with **print** statement,
- use variables and functions,
- use lists and loops,
- solve numerically linear systems using Jacobi's iterative method.

1 To turn in

Do Problems 1-29 in a single Google Colab notebook. Write *your name* and *lab number* at the beginning of your report. Clearly label each problem to separate them from each other. Make sure to comment on each problem. If your code doesn't run correctly, clearly explain what you were trying to do. Once you are finished, download the .ipynb file by clicking on **File→Download→.ipynb** and download the .pdf file by clicking on **File→Print→Save as PDF**. Submit on Canvas both the *.ipynb file* and the *.pdf file*. Here is the breakdown of points:

Problems	Points
1, 8, 9, 15	1
2-4, 6, 10-14, 16-19, 23-28	2
5, 7, 20-22, 29	3
Readability of your report	3
Total: 29	Total: 63

2 Arithmetic

1. Python follows the usual order of mathematical operations, including the use of parentheses. For example, to compute $15 \times 3 - 81 \div 9$, you will enter the following in a *code cell*:

```
15*3 - 81/9
```

   Then press **Shift+Enter** to execute the code. Then add the phrase "#This will output 36" (without the quotation marks) to the end of the above code and re-execute. Does it change the result or show an error message? Write your answer in a *text cell*.

2. To compute exponentiation, use the ** operator. For example, you can compute 2^5 by executing the following:

```
2**5
```

   Write a code to compute $\frac{118+11\times 2}{(9-2)^4}$. Then write your comment in a *text cell*.

3. You can include as many statements as you want in a single *code cell*by putting each of them on a separate line. Notice, however, that only the result of the final command is included in the output displayed underneath the cell:

```
11+1
12-11
3*7
15/3
```

   If you would like to see the output of multiple commands, you can use the **print** command to make sure that those commands are included in the output display:

```
print(11+1)
print(12-11)
print(3*7)
print(15/3)
```

   Comment on what you see in a *text cell*.

3 Use variables and functions

4. Just like in mathematics, a variable in Python is a placeholder for some value. For example, we can define a variable called a and assign the value 2 to it simply by executing the following code:

```
a=2
```

   After executing this cell, the variable a can be used in other cells within this notebook, and when executing these statements Python will replace the variable a with the value currently stored there.

```
a+15
```

   Comment on what you see in a *text cell*.

5. You can also redefine the value of a at any time in your notebook, and you can even use the current value of a when you redefine it.

```
a=2
print(a)
a=3*a
print(a)
a=3*(5-2)
print(a)
```

Explain what each command does.

6. We can also use symbols such as $<$ and $>$ to compare various quantities and variables. We can use a double equal sign == to test whether two quantities are equal, and <= and >= to test quantities that are less than or equal to, or greater than or equal to each other.

```
a=5
print(7<=a)
print(a==5)
print(a<10)
```

Explain what each command does.
Notice that the commands a=5 and a==5 have different meanings in the above code. In the first case, we are assigning the value of 5 to the variable a, while in the second case we are checking the value of a to see if it equals the number 5.

7. In a *code cell*, define two variables $a = 5^6$ and $b = 6^5$ and write a code to test if $7a$ is greater than b. Comment in a *text cell*.

8. A function can include more lines of code inside of them, which dictate which steps to perform before returning the output of the function. You can also define new variables inside of a function. In this case, each step in the function should be on its own line, indented from the first line of the function. For example, the following code defines a function f whose variable is x.

```
def f(x):      # the keyword def is used to define a function
   j=x+2
   k=3*j
   w=k-5
   return w    # w will be the value of function f
```

Remember to indent all of the lines in the function definition from the second line on. Proper use of indentation is very important in Python.

9. You can evaluate the values of f at specified values of x. For example, execute the following:

```
print(f(2))
print(f(-0.5))
```

Comment on what you see in a *text cell*.

10. Define a function called g which does exactly the same thing as the function f defined above, but which only has a definition line and a return statement. In other words, write a function that does the exact same thing as f, but which fits in only two lines of code.

11. Define a function called `triple(y)` which takes a value y as input, and outputs 3 times y.

12. You can also define functions that accept multiple values as inputs, functions that output multiple return values, and functions that call other functions when they are being evaluated.

```
def multiply(x,y):     # This function accepts two numbers as inputs, and
     return x*y        # returns their product as its output.

def sumdiff(x,y):      # This function accepts two numbers as inputs,
     return x+y, x-y   # and returns both their sum and their difference.

multiply(3,7)
sumdiff(3,7)
```

    Comment on what you see in a *text cell*.

13. Define a function $f(x, y) = \frac{x+y}{x-y}$ and evaluate $f(3, 4)$.

14. Define a function called `avg(x,y)` which takes two values x and y as input, and outputs the mean of x and y. Recall that the mean of two numbers a and b is defined to be $\frac{a+b}{2}$. Use a few cases to test your function, such as `avg(3,5)`,...

15. You can call a previously defined function in another function.

```
def multadd(x,y):
   w=multiply(x,y)+x    # Here we call the function multiply(x,y) that we
   return w             # defined earlier. It is important that the cell
                        # containing the definition of multiply(x,y) has
                        # already been executed.

multadd(3,7)
```

    Comment on what you see in a *text cell*.

4 Use lists

Python can store data in several different forms. Numbers can be stored as integers using the `int` data type, or as decimal values using the `float` data type. Python does a lot of the work to handle these different data types, and for the most part we won't need to concern ourselves with distinguishing between the `int` and `float` data types.

Another very important data type in Python is the `list` data type. A list is an ordered collection of objects which we specify by enclosing them in square brackets `[ ]`.

16. Define a list called `list1` by executing the following:

```
list1 = [2, 5, 4.1, 7, 15/6, -9.4]
```

    The built-in function `len` returns the number of elements (length) of a list.

```
len(list1)
```

    Can you define a list of 7 elements which are integers from 9 down to 3 (in that order)? Name this list `list2`.

17. You can access any of the elements in a given list by *indexing*. The elements in a list are all labeled from left to right with an integer index, starting at zero. For example, the first element in `list1` is 2, which has index 0, the second element is 5 and has index 1, and so on. To access any of the individual objects in the list, we use a pair of square brackets `[ ]` as in the following example:

```
print(list1[0])
print(list1[4])
print(list1[-1])
print(list1[-2])
```

How do the negative indexes access the list?

18. If you want to access a range of elements in `list1`, you can *slice* the list using the notation `list1[start:stop]`, where `start` and `stop` are both integer index values. Using this command will return all of the elements in the `list1` that are between the positions `start` and `stop`.

```
print(list1[0:3])
print(list1[2:4])
print(list1[2:2])
print(list1[1:-2])
```

Comment on what you see.

19. You can change lists in a number of ways. One way is to use the index of a list element to access that element and to redefine it directly.

```
list2 = [9,8,7,6,5,4,3]
print(list2[2])     # This will print a 7.
list2[2] = -15      # This changes the element at position 2 to -15.
print(list2)        # The list now has -15 where the 7 was originally.
```

Can you modify `list2` again by changing the last element so that it is equal to the first element (index 0)?

20. Create a function `first(c)` which accepts as input any list `c`, and outputs the first element in the list `c`. Test your function with a few cases.

21. Define a function `first_last(c)` which accepts as input a list `c`, and outputs two values, the first element and the last element of `c` (in that order). Test your function with a few cases.

22. Define a function `middle(c)` which accepts as input a list `c`, and outputs a list which is the same as `c` except that the first element and the last element have been removed. In other words, the function call `middle(c)` should drop the first and last elements of the list `c` and return the resulting list. Test your function with a few cases.

5 Use loops

23. Execute the following:

```
for j in range(5):
    print(j)
```

In another *code cell*, execute the following:

```
for j in range(3,9):
    print(j)
```

In another *code cell*, execute the following:

```
for j in range(3,9,2):
    print(j)
```

In another *code cell*, execute the following:

```
for j in range(10,3,-2):
    print(j)
```

Comment on what you see in each case.

24. You don't have to use the `range` function in loops. You can replace `range` with any list you like. Try the following code out:

```
my_list=[100, -3.14, 42, -1, 5.3, 14/2]
for j in my_list:
   print(j)
```

Can you print this list in the reverse order? *Hint:* think of the last example of the previous exercise.

25. Consider the following function.

```
def summation(n):
   sum=0
   for i in range(n+1):
      sum=sum+i
   return sum

print(summation(5))
print(summation(6))
```

The function summation takes as input an integer n, and then adds up all of the integers between 0 and n. The function first creates a variable `sum`, which will keep track of the running total of our summation as we add everything up. You will think of this function as adding one number at a time, so you initially define the variable sum so that it has value 0 since you haven't added any of the numbers to it yet.

The variable `i` in the for loop then runs through the integers $0, 1, ..., n$, and at each step it adds the current value of `i` to the running total in the variable sum. Once we have looped through all of the integers $0, 1, ..., n$, the function exits the loop, and returns the final value of `sum`.

Why do we use `range(n+1)` in the above function, and not `range(n)`? Write your answer in a *text cell*.

26. Write a function `product(n)` which returns the products of integers from 1 to n. For example, `product`$(5) = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. *Hint:* modify the function `summation` above.

27. Write a function `product_odd(n)` which returns the products of odd integers from 1 to n. For example, `product_odd`$(5) = 1 \cdot 3 \cdot 5 = 15$ and `product_odd`$(6) = 1 \cdot 3 \cdot 5 = 15$.

6 Jacobi's iterative method

Consider the following system of two linear equations in two variables:

$$\begin{cases} 7x - y = 6 \\ x - 5y = -4 \end{cases} \tag{1}$$

You can solve the above system by hand and find that it has a solution `(x_val,y_val)=(1,1)`. Now, instead of solving the system (1) directly, we could instead start with an initial starting estimate, say $(x_0, y_0) = (0, 0)$, and then use this estimate to hopefully create better and better approximations to the actual solution `(x_val, y_val)` of the system. To see how this is done, solve the first equation in the above system for x, and the second equation for y. We obtain the following pair of equations:

$$x = \frac{1}{7}(y + 6) \tag{2}$$

$$y = \frac{1}{5}(x + 4) \tag{3}$$

By plugging our initial estimates $x_0 = 0$ and $y_0 = 0$ into the right hand side of these equations, we obtain a second improved estimate, which we call (x_1, y_1). More precisely, we plug the value $y_0 = 0$ into equation (2) to get x_1, and we plug the value of x_0 into equation (3) to get y_1:

$$\begin{aligned} x_1 &= \frac{1}{7}(y_0 + 6) = \frac{6}{7} \\ y_1 &= \frac{1}{5}(x_0 + 4) = \frac{4}{5} \end{aligned}$$

Since the values we obtained by plugging (x_0, y_0) into equations (2) and (3) seemed to improve our estimate, it is reasonable to think we might improve our estimates even further by plugging $(x_1, y_1) = (6/7, 4/5)$ into equations (2) and (3). Indeed, doing this gives us a new improved estimate (x_2, y_2) as follows

$$\begin{aligned} x_2 &= \frac{1}{7}(y_1 + 6) = \frac{34}{35} \\ y_2 &= \frac{1}{5}(x_1 + 4) = \frac{34}{35} \end{aligned}$$

You can see that our approximations seems to be improving each time we plug the preceding values into our equations, so we might as well continue on. We can define a sequence of estimated solutions (x_n, y_n) by iterating (repeating) the above steps. Once we have computed (x_{n-1}, y_{n-1}), we plug the value y_{n-1} into equation (2) to get x_n, and we plug the value x_{n-1} into equation (3) to get y_n.

We hope that the sequence of values (x_n, y_n) will converge to the actual solution of the system (i.e., we hope that the values of (x_n, y_n) will get closer and closer to `(x_val, y_val)` as n gets larger). This procedure for solving a system of linear equations is known as Jacobi's iterative method, and is an example of an iterative method for solving a system of linear equations.

28. Define a function, called `jacobi1(x,y)`, which accepts as input values x and y, and returns a list `[new_x,new_y]` as the results of performing one iteration of Jacobi's method for system (1) on the inputs x, y. Test your code by executing `jacobi1(3,5)`.

29. Define a function, called `jacobi2(n)` which accepts as input a single non-negative integer n, and returns a list `[x_n,y_n]`, where `x_n` and `y_n` are the values of x_n and y_n respectively for Jacobi's method when applied to system (1). Use $(x_0, y_0) = (0, 0)$ as your starting approximation. Test your code by executing `jacobi2(2)`.