LAB 2 - INTRODUCTION TO PYTHON PROGRAMMING II

1. For loops

In the previous lab we learned how to define functions, and how to use `if` statements to design ones that do more than just perform arithmetic operations. In this lab we will learn about another tool for writing functions, called *for loops*. We will illustrate this idea with the following simple example of a `for` loop.

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

```
0
1
2
3
4
```

We'll first describe what the `range(5)` command does. We can think of it as creating a list of integers `[0,1,2,3,4]`, which we will call `L`, which starts with `0` and ends at `4` (strictly speaking, `range(5)` is not a list in Python, it's a function, but we can think of it as behaving like one, and we will refer to it as a "list" anyway). Notice that the second line of code above is indented. We think of this as being code that is *inside* the `for` loop. It's possible to have multiple lines of indented code following a `for` statement like the one above.

When Python encounters the statement `for j in range(5):`, it starts by assigning `j` the first value in the list `L`, namely `0`, and then it proceeds to execute the commands which are indented inside the `for` loop. In this case, `print(j)` is the only command there, and since we have assigned `j` to be `0`, this prints `0` in the output.

Once Python has finished executing all of the code inside the `for` loop, it then returns to the top of the `for` loop and passes through it all again. This time, however, it assigns `j` to be the second entry in the list `L`, which is `1`. Python again executes the code inside the `for` loop, which again consists only of `print(j)`. This time, however, `j` = `1`, and hence we see a `1` printed in the output following the `0`.

After executing the code in the `for` loop with `j` = `1`, Python then returns again to the top of the `for` loop at the beginning of the cell. At this point `j` takes on the next highest value in the list `L`, namely `2`, and proceeds again to execute the code inside the `for` loop. This continues until `j` has cycled through every value in the list `L=[0,1,2,3,4]`, and executed the code inside the `for` loop for each value of `j`.

Before proceeding to some more interesting examples, we first point out that we can change the `range` function by adding a few optional values when we call it. For example, if we use `range(3,9)` then the list that is produced will contain every integer between `3` and `8`. If instead we write `range(10,26,5)`, then it will produce a list that starts at `10`, ends at `25`, and counts up with a step size of `5`.

- Verify the above statement, by executing the following code.

```
for j in range(10,26,5):
   print(j)
```

- What happens if we execute the following? Notice here that the number in the starting position (100) is larger than the ending position (13), and the step size (-7) is negative. Make a guess of what this command will do, before executing it.

```
for j in range(100,13,-7):
   print(j)
```

- We don't have to use the range function with for loops. We can replace range with any list we'd like. Try the following code out in your practice notebook.

```
my_list=[100,-3.14,42,-1]

for j in my_list:
   print(j)
```

Let's try something slightly more complicated. Consider the following function.

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

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. We will think of our function as adding one number at a time, so we initially define the variable `sum` so that it has value `0` since we 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 `for` loop, and returns the final value of `sum`.

- Why do we use `range(n+1)` in the above function, and not `range(n)`?

Consider the following `for` loop.

```
my_list=[1,2,3,4]

for i in range(len(my_list)):
  my_list[i]=2*my_list[i]

print(my_list)
```

```
[2, 4, 6, 8]
```

First, notice that we have introduced a new function, `len`. If `L` is a list, then `len(L)` will return the length of the list `L`. So, for example, `len(my_list)` returns a value of `4`, since `my_list` has 4 elements in it.

Thus, in the above `for` loop the variable `i` runs through the indices `0`, `1`, `2`, and `3` of the elements of the list `my_list`. For each index `i`, the command inside the `for` loop takes `my_list[i]` (which is the element of the list `my_list` in the `i` position) and replaces it with `2*my_list[i]`. In other words, it doubles the value of the entry `my_list[i]`. As we might expect, after the `for` loop finishes the values in the list `my_list` have been changed to twice their original values.

Notice that in this example we were able to iterate through the indices in the list `my_list` using the `for` loop statement

```
for i in range(len(my_list)):
```

This is a very common and handy way of performing operations on a list. Notice that we can also iterate through elements of a list directly, using a `for` statement of the form

```
for i in my_list:
```

We could try to double all of the entries in our list `my_list` by instead running the code

```
1 my_list=[1,2,3,4]
2
3 for i in my_list:
4   i=2*i
5
6 print(my_list)
```

Problem 1. The output of the above code is another list. Save the value of the first element of that list as a variable called `first_elem` (this will be a number, which you can save as `first_elem`). Is the list that is output from this cell what you expected? Does the above code double the values in `my_list`, like we saw in the previous example?

This example suggests that we need to be careful in how we loop through a list if we want to change the entries in that list. Usually we will loop through the values in `range(len(my_list))` as in the first example, instead of looping directly through the elements of `my_list` as in the second example.

Problem 2. Define a function `sum_list(L)` which takes as input a list `L` of numbers, and returns the sum of the values in the list. Check your code by executing the command `sum_list([1,3,7,-13])`, which should return `-2`, since $1+3+7+(-13) = -2$.

Hint: Try structuring your code like the `summation` *function above, with a variable* `sum` *that keeps track of the running total.*

Problem 3. Define a function `list_relu(L)` which takes as input a list `L` of numbers, and returns a list which is the same as `L` except that all negative values in `L` are replaced with `0`. So, for example, `list_relu([1,-2,17,-3.2,-15])` should return the list `[1,0,17,0,0]`.

Hint: Your code will probably need to use an `if` *statement inside a* `for` *loop.*

2. NumPy

Although there are a number of useful functions which are already defined in Python, like `range` and `len`, there are many common mathematical functions like $\sin(x)$ and $\log(x)$ which are not defined. In order to use these functions (and others), we need to import the NumPy package. A *package* is a collection of functions that have been written in Python, and are available to use in our programs so that we don't have to define these functions ourselves. NumPy is a particularly helpful package that contains many functions which are important for doing linear algebra and mathematics in general.

In order to use the functions in the NumPy package, we first must import the package. To do this we use the following command:

```
import numpy as np
```

Here we are telling Python to import NumPy. We are also telling Python that we will be referring to the NumPy package in our code by the shortened `np`, instead of its full name. *You will need to do this for every notebook you create that uses NumPy.* Furthermore, if you close a notebook which has imported NumPy, and then open it again, you will need to re-execute the cell containing the command `import numpy as np` in order to use any of NumPy's functions.

To use NumPy's functions in our code, we simply have to include `np.` at the beginning of the function name.

```
import numpy as np

np.sin(0.5)    # The sine function evaluated at 0.5.
```

```
0.479425538604203
```

Here are some more examples of commonly used functions in NumPy.

```
print(np.cos(1))      # The cosine function evaluated at 1.

print(np.sqrt(16))    # The square-root function evaluated at 16.

print(np.exp(10))     # The natural exponential function evaluated at 10.

print(np.log(116))    # The natural logarithm function evaluated at 116.
```

```
0.5403023058681398
4.0
22026.465794806718
4.7535901911063645
```

Note that the trigonometric functions in NumPy are computed in terms of *radians*, and that `np.log` is the *natural logarithm*, with base e.

Problem 4. Find the value of

$$\frac{e^5 - \log(\sqrt{5})}{e^{\cos(3)}}$$

using NumPy functions, and save its value as the variable `my_var`. Here $\log(\sqrt{5})$ denotes the natural logarithm of $\sqrt{5}$ (i.e. the logarithm with base e).

Another useful feature of the NumPy package is that it also contains functions for dealing with vectors and matrices. In NumPy we represent matrices and vectors as *arrays*. To define a NumPy array, we use the function `np.array`. For example, if we want to create the vector

$$\begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}$$

as a NumPy array, we first create the list `[1,2,-1]` in Python, and then plug it into the function `np.array`.

```
my_list=[1,2,-1]            # This is a good old-fashioned list.

my_vect=np.array(my_list)   # my_vect is a NumPy array now, which we think
                            # of as a vector.

print(my_vect)              # This prints the array my_vect.
```

```
[ 1  2 -1]
```

Alternatively, one could create `my_vect` simply by writing

```
my_vect=np.array([1,2,-1])  # Creates the same array as before, but without
                            # having to define a list in advance.
```

When working with vectors you may notice that they are defined and displayed in NumPy as row vectors, even though we will think of them as column vectors (and they behave like column vectors when we do linear algebra with them). For example, once we learn how to define matrices as NumPy arrays, we can create a 5×3 matrix `A` with NumPy, and multiply it by the 3-dimensional vector `my_vect` as we could with any 3-dimensional column vector.

On the surface, it looks like our array `my_vect` is not very different from the list we used to construct it. But we must be careful since Python treats NumPy arrays differently than it does lists.

- Compute the following in a cell:

```
list1=[1,2,3]
list2=[2,-1,0]

print(list1+list2)
```

How does Python combine the lists `list1` and `list2` when we use the command `list1+list2`?

- Now compute the same thing, but this time convert `list1` and `list2` into NumPy arrays first:

```
array1=np.array(list1)
array2=np.array(list2)

print(array1+array2)
```

How does Python combine the NumPy arrays `array1` and `array2` when we use the command `array1+array2`?

In linear algebra we usually want our vectors to behave like NumPy arrays, instead of Python lists. It is for this reason that we will use NumPy arrays to represent our vectors and matrices, instead of lists.

To define matrices in NumPy, we define them as "lists of lists". In other words, a matrix can be defined by creating a list, whose elements are all lists of the same size that represent the rows of the matrix, and then plugging it into the function `np.array`. For example, to define the matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ -5 & -6 & -7 & -8 \\ 1 & 5 & 2 & 3 \end{bmatrix}$$

we would create a list with three elements. The first element will be the list `[1, 2, 3, 4]`, which we think of as the first row of the matrix. The second element in our list will be `[-5, -6, -7, -8]`, representing the second row, and so on.

```
my_matrix=np.array([[1, 2, 3, 4],[-5, -6, -7, -8],[1, 5, 2, 3]])

print(my_matrix)
```

```
[[ 1  2  3  4]
 [-5 -6 -7 -8]
 [ 1  5  2  3]]
```

Besides being able to add NumPy vectors together as shown above, we can also multiply them by scalars in a straightforward way:

```
my_vect=np.array([1,2,-1])

3*my_vect               # Scalar multiplying my_vect by 3.
```

```
array([ 3, 6, -3])
```

Furthermore, if `a` and `b` are vectors of the same size (represented as NumPy arrays), then we can perform the dot product of `a` and `b` using the function `np.dot(a,b)`.

```
a=np.array([1,2,3,1])   # Defining two vectors of the same size, a and b.
b=np.array([1,1,1,1])

np.dot(a,b)             # Taking the dot product of a and b.
```

```
7
```

Problem 5. Let

$$\mathbf{v} = \begin{bmatrix} 1 \\ 3 \\ -2 \\ 4 \\ 5 \end{bmatrix}, \qquad \mathbf{u} = \begin{bmatrix} 1 \\ 1 \\ -2 \\ 1 \\ 1 \end{bmatrix}, \qquad \text{and} \qquad \mathbf{w} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}.$$

Compute the value of

$$\left(\frac{\mathbf{v}\cdot\mathbf{u}}{\mathbf{u}\cdot\mathbf{u}}\right)\mathbf{u} + \left(\frac{\mathbf{v}\cdot\mathbf{w}}{\mathbf{w}\cdot\mathbf{w}}\right)\mathbf{w}$$

and save its value as a NumPy array called `my_vect_var`.

Hint: Carefully identify which quantities in the above expression are scalars and which are vectors.

We can access elements of a NumPy array the same way we access elements in a list, by specifying indices or ranges of indices. Recall that Python lists (and NumPy arrays) begin at index 0. So if an element of a list or array has index 3, that really means it's the 4th element in the list or array. Furthermore, when we specify a range of indices, say `my_array[3:7]`, the object with index 3 is included, but the object with index 7 is not included (Python only includes up to index 6).

```
v=np.array([4,1,-5,3,-2,1,0,9])

print(v[3])     # Print out the entry at index 3 (i.e. the fourth entry)
                # in the vector v.

print(v[2:6])   # Print out all entries in v between (and including)
                # index 2 and index 5. Notice that the starting
                # index 2 is included, but the stopping index 6 is not.

print(v[3:])    # Print out the entries starting at index 3 to the end of
                # the vector.

print(v[:6])    # Print out the entries starting at the beginning of the
                # vector up to the entry at index 5.

print(v[:])     # If we don't specify a start or stop index we get back the
                # whole array.
```

```
3
[-5  3 -2  1]
[ 3 -2  1  0  9]
[ 4  1 -5  3 -2  1]
[ 4  1 -5  3 -2  1  0  9]
```

We can access the entries in a matrix in a similar way to accessing elements of a list, though for matrices we have to list two indices (or ranges of indices), to specify the location of the row(s) and/or column(s) in which we are interested.

```
my_matrix=np.array([[1, 2, 3, 4],[-5, -6, -7, -8],[1, 5, 2, 3]])

print(my_matrix[1,2])    # Print the element in the second row, third column
                         # of my_matrix.

print(my_matrix[2,1:3])  # Print everything in the third row, from the second
                         # to the third column.

print(my_matrix[:,3])    # If we don't specify a start or stop index in the
                         # first spot, we get the entire fourth column.

print(my_matrix[2,:])    # If we don't specify a start or stop index in the
                         # second spot, we get the entire third row.
```

```
-7
[5 2]
[ 4 -8  3]
[1 5 2 3]
```

Problem 6. Define a function `first_rpt(M)` which takes as input a NumPy matrix `M`, and outputs a matrix in which every row of `M` has been replaced with the first row.

For example, the output of `first_rpt(my_matrix)`, where `my_matrix` is the matrix defined directly above, should be the matrix

```
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]])
```

Hint: Your function should contain a `for` *loop, which loops over the number of rows in a matrix. Use* `len(M)` *to determine how many rows are in the matrix* `M`.

3. Nested for loops

Oftentimes when working with matrices it is helpful to use more than one `for` loop, with some loops sitting inside of others. We call these *nested for loops*. Consider the following simple code.

```
1 for i in range(4):
2   for j in range(3):
3     print('i =',i,' and j =',j)
```

In this code, there are two `for` loops, an outside loop with variable `i`, and an inside loop with variable `j`. When we first encounter the outside loop, we set the value of `i` to be `0`, before executing the code inside this loop. Executing the code inside the `i` loop involves running another `for` loop though, this time with variable `j`. The inner `j` loop is thus executed, and we cycle through all of the `j` values, while the `i` value stays fixed at `0`.

Once we've finished cycling through all of the `j` values, we then exit the inside `j` loop, and return to the top of the outside `i` loop. It is at this time that the variable `i` is assigned the value `1`, before the inner `j` loop is called again, and we cycle through all of the `j` values once again. This continues until we've run through all of the `i` values and the `j` values. The output of this code is shown below.

```
i = 0 and j = 0
i = 0 and j = 1
i = 0 and j = 2
i = 1 and j = 0
i = 1 and j = 1
i = 1 and j = 2
i = 2 and j = 0
i = 2 and j = 1
i = 2 and j = 2
i = 3 and j = 0
i = 3 and j = 1
```

```
i = 3 and j = 2
```

Consider the following, slightly more complex, code. Here we define a function that takes a matrix `M`, and replaces all of the negative entries with their absolute values (so for example, if a `-2` occurs somewhere in the matrix, that entry is replaced with `2`, while any nonnegative entries are left alone).

```
def abs_matrix(M):
  # This is the number of rows in M.
  n_rows=len(M)
  # The number of entries in the first row, which is the number of columns.
  n_cols=len(M[0,:])
  # i represents the row position.
  for i in range(n_rows):
    # j represents the column position.
    for j in range(n_cols):
      # If M[i,j] is negative, we make it positive.
      if M[i,j]<0:
        M[i,j]=-M[i,j]
  return M
```

In the above function, we first create two variables, `n_rows` and `n_cols` which store the number of rows and columns in `M` respectively. After defining these two variables there are two loops, one inside of the other. The outside loop uses the variable `i`, which loops through the different row indices in `range(n_rows)`. For each step in the outside `i` loop (which we think of as being a row of `M`), we run through another `for` loop, this time cycling through the column indices in `range(n_cols)`. For each combination of `i` and `j`, we test whether the entry `M[i,j]` in the `i`, `j` location is negative, and if it is we replace it with its absolute value.

We can test that the code does what we think it should using the following.

```
mat=np.array([[1,-1,2,-3,1,1],[-2,-2,0,1,1,-5],[1,1,1,1,-2,-1]])
print(mat)              # Define and print a matrix

abs_mat=abs_matrix(mat) # Plug mat into our function abs_matrix, and print
                        # its output to compare.
print(abs_mat)
```

```
[[ 1 -1 2 -3  1  1]
 [-2 -2 0  1  1 -5]
 [ 1  1  1  1 -2 -1]]
[[1 1 2 3 1 1]
 [2 2 0 1 1 5]
 [1 1 1 1 2 1]]
```

Problem 7. Define a function, called `matrix_sum(M)`, which takes as input a matrix `M` (as a NumPy array), and adds up all of the entries.

For example, if `mat` is the matrix we defined directly above, then `matrix_sum(mat)` should return `-5`, since all of the entries in `mat` add up to -5.

Hint: You will need to use two nested `for` *loops. Use the code in the function* `abs_matrix` *as a template for how to structure your code.*

4. List comprehension

One handy way to define lists (and NumPy arrays) is by using a *list comprehension*. To illustrate how this is done, consider the following.

```
a=[3*i for i in range(10)] # An example of defining a list using a list
                           # comprehension.
print(a)
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

The first part of the above list comprehension, namely `3*i`, tells Python that we are going to create a list and fill it with numbers of the form `3*i`, for some values of `i`. The second part of the list comprehension, the command `for i in range(10)`, tells Python what values of `i` to use. In other words, we are creating a list with the elements `3*i`, where `i` ranges between `0` and `9`.

Problem 8. Using a list comprehension, create a list

$$[0.5^1, 0.5^2, 0.5^3, \ldots, 0.5^{100}],$$

and save it as a variable called `long_list`.

Hint: Make sure that your list comprehension is defined using the correct range of values.

Much like nested `for` loops, we can use double list comprehensions to create more complicated lists. We can also have a list comprehension cycle through a list of functions instead of just a range of numbers (we will need to do this in a later lab). Suppose, for example, that we wanted to create a list of the form

$$[\sin(1), \cos(1), \log(1), \sin(2), \cos(2), \log(2), \ldots, \sin(99), \cos(99), \log(99)]\,.$$

We could do this using a double list comprehension as follows.

```
# An example of a double list comprehension.

a=[f(i) for i in range(1,100) for f in [np.sin, np.cos, np.log]]
```

In this example, the `for i in range(1,100)` acts similarly to an outer `for` loop, while `for f in [np.sin, np.cos, np.log]` acts like an inner `for` loop. For each `i` value, the function `f` cycles through the different function `np.sin`, `np.cos`, and `np.log`, before moving on to the value `i+1`.

- Another example of double list comprehension, without iteration over functions, is given by the following code.

```
a=[i*j for i in range(1,4) for j in range(2,5)]
print(a)
```

See if you can run through the list comprehensions in your mind, to predict the output of the above code. Then execute it in your practice notebook to see if your prediction was correct.

Problem 9. Using a double list comprehension, create a list

$$[1^1, 2^1, 3^1, 1^2, 2^2, 3^2, 1^3, 2^3, 3^3, \ldots, 1^{99}, 2^{99}, 3^{99}],$$

and save it as a variable called `very_long_list`.

Hint: By looking at the above example try to figure out the order in which the list comprehensions need to be stated. If you don't obtain the correct answer, try swapping the order of the list comprehensions.