LAB 1 - INTRODUCTION TO PYTHON PROGRAMMING I

1. The Python programming language

Python is a powerful, general-purpose programming language that is widely used in science and engineering. It has many libraries and packages available to add on that make it a powerful tool for scientific computing and applied mathematics. Even better, it is a great first language to learn for beginners to computer programming! The goal of these labs will be to introduce you to a variety of interesting linear algebra applications through Python programming.

There are several ways to write programs in Python. In these labs we will be using *Google's Colaboratory notebooks* to write our code and create the files we will submit for grading. This allows us to write and run Python programs in an internet browser without having to worry about downloading and installing any software on our computers. This will be described in more detail below.

A second way to write Python programs is to save your commands in a file, called a *script*, and then pass that file to a Python interpreter. The Python interpreter is a program on your computer that reads and executes the commands you saved in your script, and outputs the results of those commands. This is a common way to use Python, and in fact many computers you can buy come with Python interpreters already installed. We won't use this approach in Math 215, but if you continue to write more complex software after this course you will likely use this method.

Instead of writing all of your commands in a single script which you pass to the interpreter, yet another way to code is by opening Python first and then entering your commands directly into the interpreter. This is called *interactive mode*, and it's a great way to play around with different commands and test out different ideas.

2. Google Colab Notebooks

We will now familiarize ourselves with Google's Colab notebooks. In what follows, any text inside a green box indicates instructions for you to follow. These will include instructions for creating and downloading your Colab notebook, and testing different bits of code throughout the lab. In general, any code that we ask you to execute in a green box can be done in a practice notebook which won't be handed in. Any text inside a blue box, on the other hand, is a problem you'll need to answer and submit for grading. You can write this code initially in a practice notebook, and then once you've checked to make sure that it works, copy and paste it inside the prepared lab notebook. We'll give you instructions on how to find the prepared lab notebook and how to set up a practice notebook in the instructions to follow.

- The first step is to make sure you have a Google account (i.e. a Gmail account). You can set one up at the following webpage

  `accounts.google.com/SignUp`

- Once your Google account is setup, navigate to the page

  `colab.research.google.com`

  This will take you to a page with some introductory material about Google Colab.
- Click the button in the top right corner to sign into your Google account if you are not already signed in.

From the Google Colab welcome page you can navigate through a variety of introductory material to learn about Google Colab, or you can create a new notebook of your own to play around with. We will do the latter and create a new practice notebook in which to work and test our code.

- Create a new notebook by selecting

  `File > New Python 3 notebook.`

  from the top menu in Google Colab.

Once you've created a new notebook you'll notice a little box to enter your code underneath the various menus. This is called a *cell*, and it is the basic building block of a Colab notebook. Any code that you want to execute in a notebook must be placed in a cell.

- In your notebook type the following into the cell:

```
1 print("Hello world!")
```

As you might expect, this is a command which tells the notebook to display the phrase `Hello world!` below the cell. In order to have Python actually perform this command, we must *execute* the cell, by holding down the *Shift* button and pressing *Return* or *Enter*.

- Execute this cell. You should see the following:

```
1 print("Hello world!")
```

```
Hello world!
```

Congratulations, you've just run your first Python program. After running your "Hello world!" command, you should notice that a new cell appears beneath the output of your last cell. You can enter new commands into this cell, and execute them the same way you did in the above cell. You can enter as many commands in a cell as you want, by putting each of them on a separate line. When you execute a cell with more than one command, the commands will be executed one at a time, from top to bottom.

- In the cell that was just created write and execute the following two commands:

```
print("Hello world!")
print("I'm hungry.")
```

```
Hello world!
I'm hungry.
```

After executing this cell you should see the two statements "Hello world!" and "I'm hungry." printed one after another.

After each cell you execute a new cell should be created at the bottom of the notebook. If at any time you want to create an additional new cell, you can push the + Code button at the top of the notebook. You can also move any cell up or down in the notebook, by selecting that cell and pushing the ↑ and ↓ buttons that appear in the top right corner of the cell. Likewise, you can delete any cell by selecting it first, and then clicking on the 🗑 icon that appears in the top right corner of the cell. At any time you can go back to a previous cell and edit it, and re-execute it as many times as you'd like. It is this ability to execute your code in small bite-sized chunks, and to go back and forth editing different parts, that makes Colab notebooks an easy way to quickly test out different ideas and pieces of code.

From this point on we'll be doing our work in a previously prepared notebook, called `Intro_to_Python_I.ipynb`.

- Go to the assignment window for this lab on Learning Suite (the page where you downloaded these instructions).
- Click on the second link, which will take you to a Colab notebook with the title

  `Lab 1 - Introduction to Python programming I`
- You may be presented with several apps to open the file with. Choose *Colaboratory*. *Note: In some cases the option to open the link in Colaboratory may not show up immediately. If this happens make sure you are logged into your Google account, and refresh the page if necessary.*
- Once you see the lab notebook, save a copy of it to your Google Drive folder by selecting

  `File > Save a copy in Drive...`

  You will not be able to edit the notebook until you save a copy in your Google Drive.

This will make an editable copy of the notebook, called `Copy of Intro_to_Python_I.ipynb`, which will be saved in a folder called "Colab Notebooks" that should automatically be created in your Google Drive folder.

- *Do not make any changes to the first cell!* This cell contains code which will be needed to help grade your lab.

- In the second cell enter your first and last names, Math 215 section number, and BYU NetID where indicated. For example, if my name is Carl Gauss, I'm registered in Math 215 section 11, and my BYU NetId is "GaussianElimination4", I would enter it in the second cell as follows:

```
# Enter your first and last names in between the quotation marks.

first_name="Carl"

last_name="Gauss"

# Enter your Math 215 section number in between the quotation marks.

section_number="11"

# Enter your BYU NetID in between the quotation marks.  NOT YOUR BYU ID

BYUNetID="GaussianElimination4"
```

It is important that you input your information correctly (with no extra spaces), as this is how we will associate your submitted lab to your name.

Now that you have your own copy saved along with your NetID you are ready to go! You should be able to edit the notebook, save your changes, and access them at any time through your Google Drive account.

3. Python basics

Now that you are working in your copy of the lab notebook, it's time to learn some Python. In what follows, feel free to play around and test out variations on the commands we discuss below. It will be best to use a separate notebook when doing this, however, so that you don't accidentally change any of the required functions or variables that will need to be graded in the `Copy of Intro_to_Python_I.ipynb` notebook.

- Create a new notebook to explore and test code in by selecting

  `File > New Python 3 notebook`

Anytime you want to test parts of your code that you are unsure of, or want to try new things, you can do so in the new notebook you've created. When you are sure that your code works, you can copy and paste it into the `Copy of Intro_to_Python_I.ipynb` notebook.

Comments. The first thing to notice is that you will often see the symbol `#` at various places in the notebooks. The symbol `#` and everything that follows it on the same line is called a *comment*. Python ignores everything on the line after the `#`, and hence it allows us to put instructions and explanations in our code to make it easier for others to read and understand.

```
print("Hello world!")  # Everything following the print statement is a
                       # comment, and is ignored by Python.
```

```
Hello world!
```

When writing code it is good practice to add explanatory comments about individual parts of your code. Not only does it help others read your code, it also helps you remember what each part of your code is supposed to do.

Arithmetic. Perhaps the most basic use of our notebook is as a fancy calculator, with all of the standard arithmetic operations defined: `+`, `-`, `*`, and `/`. Python follows the usual order of mathematical operations, including the use of parentheses. For example, to compute $15 \times 3 - 81 \div 9$, we would enter

```
15*3-81/9  # This will output 36.
```

```
36.0
```

We can also compute exponentiation using the `**` operator. For example, we can compute 2^5 by typing the following.

```
2**5  # This will output 32.
```

```
32
```

- Compute the values of $(13 - 17) \times 6$ and $2^3 + 21$ in your practice notebook.

Output and print statements. As mentioned above, we can include as many statements as we want in a single 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
```

```
5.0
```

If we'd like to see the output of multiple commands we can use the `print` command to make sure that those commands are included in the output display.

- Enter the following commands in a cell, and execute them. What output do you see?

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

Variables. 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    # We are assigning the value 2 to the variable a.
```

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   # The variable a currently holds the value 2.
```

```
17
```

We can also redefine the value of `a` at any time in our notebook, and we can even use the current value of `a` when we redefine it.

```
a=2
print(a)  # This statement will print the number 2 when executed.
a=3*a     # Now we've redefined a to be 3 times the original value of a.
print(a)  # This statement will print 6, which is the new value of a.
a=3*(5-2) # Now a is 9.
print(a)
```

```
2
6
9
```

Sometimes it is useful to swap the values of given variables. To do this, we can introduce a "placeholder" variable as follows:

```
x=2          # x has value 2.
y=5          # y has value 5.
print(x,y)   # This will print both the value of x and the value of y.
z=x          # z is a placeholder, holding the value of x.
x=y          # x is changed to hold the value of y, which is 5.
y=z          # y is changed to hold the value of z, which was the
             # original value of x, which is 2.
print(x,y)
```

```
2 5
5 2
```

- Enter the following commands into a cell, and then execute it. What do you expect the output will be?

```
b=5
print(b)
b=b+7
print(b)
b=3*(5-b)
print(b)
```

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           # Assigning 5 to the variable a.
print(7<=a)   # This will print False, as 7 is not less than or equal to 5.
print(a==5)   # This will print True, as a is equal to 5.
print(a<10)   # This will print True, as a is less than 10.
```

```
False
True
True
```

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` and testing if it equals the number 5.

- What will the output of the following cell be?

```
c=-5
c=c+3
print(c==-5)
print(c>=1)
print(c==-2)
```

Finally, notice that variables in Python can represent a variety of objects, not just numbers. For example, variables can represent *strings*, which are sequences of characters, or *Booleans* which are variables that are either *True* or *False*.

```
c="aString"    # Assigning a string to the variable c.
b=7>-2         # Since 7 > -2, the value True will be saved in b.
print(b)       # This prints the Boolean value True.
print(c)       # This prints the string "aString".
```

```
True
aString
```

Problem 1. In the `Copy of Intro_to_Python_I` lab notebook enter the expression

$$\frac{118 + 11 \times 2}{9 - 2}$$

and store it as a variable called `my_first_var`. (Remember to use parentheses to ensure the order of operations is correct.) Don't just save the numerical value of this expression, which is 20. Save the actual expression with the addition, multiplication, division, subtraction, and parentheses as the variable.

For example, if we asked you to save the expression $3 \times 2 + 5$ *as a variable called* `var`*, we would write*

```
var=3*2+5  # Like this.
```

instead of writing

```
var=11  # Not like this.
```

4. Functions

In computer programming, like in mathematics, a function is an object which accepts as input values from some set, and produces output which depends both on the input values and some given rules. In Python we illustrate how to define simple functions with the following example.

- Type the following into a cell, and execute it.

```
def reciprocal(i):
    # A function that takes the reciprocal of the input i.
    return 1/i
```

Here we have defined a function called `reciprocal`, which has a single input parameter `i`. The first line of the function definition begins with `def`, followed by the name of the function, the parameters it accepts in parentheses, and ends with a colon. Each line in the remainder of the function must be indented (which Colab will do for you automatically), and the function definition ends with a `return` statement that defines what the output of the function will be. It is customary to put a docstring directly below the `def` statement, which is a comment designed to describe the purpose of the function to the user. In the case of our function `reciprocal`, it accepts a single value `i` as its input, and it returns the value `1/i` as its output. To evaluate our function on a given input, we write it much as one might expect:

```
reciprocal(13)
```

```
0.07692307692307693
```

```
a=2          # We can also pass variables to functions.
reciprocal(a)
```

```
0.5
```

- Evaluate the following. What do you expect the output to be?

```
print(reciprocal(-1))
print(reciprocal(0.5))
print(reciprocal(0))
```

You should have received an ugly error message when you tried to evaluate `reciprocal(0)`, as a result of trying to divide by zero. Python will produce an error message anytime you try to execute code that violates one of its rules. Learning to interpret error messages is an important part of becoming a good programmer, and more details can be found in the "Common Errors and Error Messages" document on Learning Suite. Be warned though, just because you don't get an error message when you execute your code doesn't mean that your code is doing what you want it to be doing. This is why we will always test our code with various input values.

Our functions can also include more lines of code inside of them, which dictate which steps to perform before returning the output of the function. We 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.

- Define the following function in your practice notebook. *Remember to indent all of the lines in the function definition from the second line on! Proper use of indentation and whitespace is very important in Python.*

```
def arithmetic(i):
    j=i+2
    k=3*j
    w=k-5
    return w
```

- What output do the following commands produce?

```
print(arithmetic(3))
print(arithmetic(-10))
```

Problem 2. Define a function called `arithmetic2(i)` which does exactly the same thing as the function `arithmetic(i)` 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 `arithmetic(i)`, but which fits in only two lines of code.

You can test that your code is correct by evaluating the commands `arithmetic2(3)` and `arithmetic2(-10)`. Your function should output `10` and `-29` respectively.

One item to note in the above example is how variables are treated by Python when they are defined inside of a function (like the variables `j`, `k`, and `w` above). They are examples of *local* variables, which are defined and can only be accessed from within the function itself. For example, when calling the function `arithmetic(3)`, the intermediate variable `k` is assigned the value of `15` as part of the evaluation. However, as soon as the function finishes evaluating, the variable `k` and its value are immediately discarded, and can no longer be accessed. Trying to access it will result in an error message, indicating that we did something wrong:

```
arithmetic(3)
k
```

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

We can also define functions that accept multiple arguments 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.

multiply(3,7)
```

```
21
```

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

sumdiff(3,7)
```

```
(10,-4)
```

```
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)
```

```
24
```

Problem 3. (1) Define a function called `triple(y)` which takes a value `y` as input, and outputs 3 times `y`.

(2) 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 $\frac{a+b}{2}$.

(3) Define a function called `combine(x,y)` which takes a pair of input values `x` and `y`, and finds the mean of `x` and 3 times `y`. The function `combine(x,y)` should call both of your functions `triple(y)` and `avg(x,y)` in its definition.

You can test that your code is correct by evaluating the commands `triple(10)`, `avg(5,25)`, and `combine(6,5)`. These commands should output `30`, `15.0`, and `10.5` respectively.

When defining functions it is always a good idea to test your function with a few different input values to make sure it is behaving as you expect. We've given you some input values to test above, but it shouldn't be difficult to come up with other input values and their expected outputs to further test your functions.

- Check the functions you've defined above with a few additional input values to make sure you haven't made an unexpected error. This is a very important habit to get in the practice of doing. We encourage you to do it after all of the functions you define in these labs.

5. 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. We've also seen examples of *strings*, which are sequences of characters, like the string `"Hello world!"` we printed at the beginning of this lab.

Another very important data type in Python is the `list` data type. A list is an ordered collection of objects (which can be numbers, strings, or even other lists), which we specify by enclosing them in square brackets `[]`.

- Define a list called `my_list` by executing the following.

```
my_list=["Hello",91.7,"world",15,100,-10.2]
```

Here the list `my_list` contains two strings, two floats (decimal values), and two integers. We can access any of the elements in a given list, or any subset of the elements by *indexing* and *slicing*. 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 `my_list` is `"Hello"` which has index 0, the second element is `91.7` 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.

```
my_list=["Hello",91.7,"world",15,100,-10.2]

print(my_list[0])   # This will print the first element of the list,'Hello'.
print(my_list[4])   # This will print the fifth element of the list, 100.
```

```
Hello
100
```

An important thing to remember is that Python begins indexing elements of a list *starting at 0*. This may seem unusual at first, since humans typically start counting objects with the number 1.

We can also access elements from the end of a list by using negative numbers.

- What values are displayed when we execute the following?

```
print(my_list[-1])
print(my_list[-2])
print(my_list[-3])
```

- What will the output be when you execute `my_list[-6]`?

If we would like to access a range of characters in a list, we can *slice* the list `list_var` using the notation `list_var[start:stop]`, where `start` and `stop` are both integer index values. Using this command will return all of the objects in `list_var` that are between the positions `start` and `stop`.

- Define the list

```
list_var=[0,1,2,3,4,5,6,7,8,9,10]
```

- What is the output of the following command? Notice the difference between how Python treats the starting index and the stopping index in both cases.

```
print(list_var[3:6])
print(list_var[0:7])
```

- Will `list_var[0:10]` print the entire list, or will it miss any entries? Evaluate `list_var[0:10]` to confirm your answer.

By not specifying a starting or stopping index, Python returns the elements starting at the beginning of the list, or stopping at the end.

```
print(list_var[3:8])   # Prints elements with index 3 through 7.

print(list_var[:4])    # When the starting index is omitted the slice
                       # will start at the beginning of the list.

```

```
print(list_var[6:])    # When the stopping index is omitted the slice
                       # will stop at the end of the list.
```

```
[3, 4, 5, 6, 7]
[0, 1, 2, 3]
[6, 7, 8, 9, 10]
```

There is something you will need to be careful about when using lists in Python, and in particular when you are trying to copy a list. Suppose I create a list, called `list_a` with the values `[1,2,3]`. Suppose I then create a second list `list_b`, and assign it the value of `list_a`. As expected, when we print the values of `list_b` Python returns the list `[1,2,3]`.

```
list_a=[1,2,3]
list_b=list_a
print(list_a)
print(list_b)
```

```
[1,2,3]
[1,2,3]
```

You might expect that what we've done above is to create two separate lists, `list_a` and `list_b`, both of which happen to have the same values. However, *we have actually only created a single list, but given it two different names* `list_a` *and* `list_b` *to reference it by!* For example, if we change one of the entries in `list_b`, we will also be changing the list `list_a`.

```
list_b[0]=100  # This changes the first entry in both list_b and list_a!
print(list_a)
print(list_b)
```

```
[100,2,3]
[100,2,3]
```

There are several ways to create a new copy of a list, which will avoid this behavior. One is by using the command `list_a.copy()`, which we illustrate below.

```
list_a=[1,2,3]

list_b=list_a.copy()  # Here we create a separate copy of list_a,
                      # and assign it to list_b.
print(list_a)
print(list_b)

list_b[0]=100         # Now this only changes list_b.

print(list_a)
print(list_b)
```

```
[1,2,3]
[1,2,3]
[1,2,3]
[100,2,3]
```

As you might imagine, you can create a copy of any list by putting `.copy()` after the name of the list to be copied.

Problem 4. (1) Create a function `first(c)` which accepts as input any list `c`, and outputs the first element in the list `c`.

(2) 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).

(3) 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.

You can test that your code is correct by defining the list `w=[1,2,3,4,5]`, and evaluating the commands `first(w)`, `first_last(w)`, and `middle(w)`. These commands should output `1`, `(1, 5)`, and `[2, 3, 4]` respectively.

- Test each of your three functions above on a few lists of different lengths to see if you get what you expect.
- Test each of your three functions above on a list with only one element, like `c=[1]`. What do you expect the output will be for each of the functions? What actually happens?
- Test each of your three functions above on the empty list `c=[]`. What do you expect the output will be for each of the functions? What actually happens?

A good idea after defining a function is to test it on some unusual cases to see if it behaves as you expect, like we did above. Don't be too worried if the functions you defined give you an error when you input the lists `c=[1]` or `c=[]`. There are ways to handle exceptions like these, though we won't discuss them. For now though it's a good idea to always try to understand the limitations of your functions and what inputs might yield undesirable results.

Finally, we 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.

```
1 my_list=[1,2,3,4]
2
3 print(my_list[2])   # This will print a 3.
4
5 my_list[2]=-15      # This changes the element at position 2 to -15.
6
7 print(my_list)      # The list now has -15 where the 3 was originally.
```

```
3
[1, 2, -15, 4]
```

Problem 5. Define a function `swap(c)` which accepts a list `c` with two or more elements, and returns another list which is the same as `c` except that the first and last elements are switched.

Note: In your function code we have included a statement `original_list=c.copy()`. As described above, this creates a copy of the input list `c`, called `original_list`. Your code should only use the copied list `original_list`, instead of the list `c` that is input into the function. This way your function will return a new list, instead of just editing the original input list `c`.

Hint: In order to swap the value of two variables, you'll need to create a separate third variable to hold one of the values temporarily.

You can test that your code is correct by evaluating the command `swap([0,1,2,3,4,5])`, which should output `[5, 1, 2, 3, 4, 0]`.

6. IF STATEMENTS

So far we have enough tools to create functions which perform arithmetic operations and rearrange lists, but not much else. To define more complicated functions we will need a few more building blocks. We will end this lab by talking about *if statements*, and will learn about *for loops* in the next lab. We introduce `if` statements with the following example.

```
1 if 1<7:
2     print("1 is less than 7")
3 else:
4     print("1 is not less than 7")
```

All `if` statements start with a condition, or question, whose answer may be either `True` or `False`. In our case, this question is asking whether the number 1 is less than 7. When Python executes the `if` statement it first checks to determine whether the condition is `True` or `False`. If the condition is `True` then Python will continue and execute the code which is contained immediately below the `if` statement line. If, on the other hand, the condition is `False`, then Python will jump immediately to the `else` line and execute the block of code below it, skipping over any commands in between.

In our case, because 1 is indeed less than 7, Python will execute the line after the `if` statement, and will print the following output.

```
1 is less than 7
```

Note that `if` statements do not need to be followed by `else` statements. If an `if` statement is not followed by an `else` statement, and the condition contained in the `if` statement is `False`, then the code won't do anything:

```
if 1>7:
   print("1 is less than 7")     # This won't execute since 1>7 is False.
```

- Execute the following code.

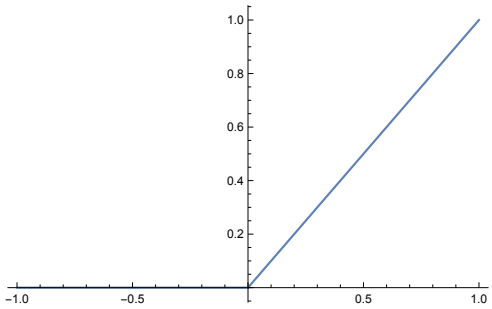
```
a=-5

if a==7:
   a=4
else:
   a=7
```

  What is the value of `a` now? (Recall that the double equal sign in `a==7` is a question that asks whether `a` is equal to `7`, while the single equal sign, say in `a=4`, is setting the value of the variable `a` equal to `4`.)

- Check to see if you are correct by executing `print(a)`.

Suppose now that we want to define a function `f` in Python whose graph is displayed below.



In other words, if `x` is less than `0`, then we have `f(x)=0`, while if `x>=0` then `f(x)=x`. We can define such a function using an `if` statement quite simply, by including two separate `return` statements in the `if` statement.

```
def f(x):
   if x<0:
      return 0
   else:
      return x

print(f(7))
print(f(-100))
```

```
7
0
```

Notice that every time we call the function `f(x)` only one of the two `return` statements is being executed, while the other is simply skipped over depending on whether the `if` evaluates the condition to be `True` or `False`.

Problem 6. Define a function, called `absolute_value(x)` which accepts as input a single number `x`, and returns the absolute value of `x`.

Hint: What should `absolute_value(x)` *be if* `x` *is negative? What should* `absolute_value(x)` *be if* `x` *is positive?*

You can test that your code is correct by evaluating the commands `absolute_value(10)` and `absolute_value(-10)`, both of which should output `10`.

To test more complicated conditions it is useful to use the `and` and `or` operators. The statement `P and Q` will return `True` only if both `P` and `Q` are `True`. If either one of, or both of, `P` and `Q` are `False`, then the statement `P and Q` will return `False`.

```
(10<11) and (-3>=-12)    # This will return True because both (10<11)
                         # and (-3>=-12) are True.

(10<11) and (-3==-12)    # This will return False because one of
                         # the statements is False.

(10==11) and (-3==-12)   # This will also return False because both of
                         # the statements are False.
```

The statement `P or Q`, on the other hand, will return `True` if at least one of, or both of, `P` and `Q` are true. The only situation in which `P or Q` will return `False` is if both `P` and `Q` are `False`.

```
(10<11) or (-3>=-12)     # This will return True because at least
                         # one of the statements is True.

(10<11) or (-3==-12)     # This will return True because at least
                         # one of the statements is True.

(10==11) or (-3==-12)    # This will return False because both
                         # of the statements are False.
```

Problem 7. Define a function, called `indicator(lower,upper,n)` which accepts as input three numbers `lower`, `upper`, and `n`, with `lower <= upper`, and returns `1` if the number `n` satisfies `lower <= n <= upper`, and returns `0` otherwise.

For example, we should have `indicator(3,7,2)=0` because `2` is not between `3` and `7`, and `indicator(-3,9,8)=1`, because `8` is between `-3` and `9`. Test your function using these values, and other values of your choosing.

Problem 8. Define a function, called `trunc_max(x,y)` which accepts as input two numbers `x`, `y`, and returns the larger of the two numbers if at least one of them is positive, and returns `0` otherwise.

For example, we should have `trunc_max(3,-5)=3`, and `trunc_max(2,7)=7`, while `trunc_max(-173,-21)=0`. Test your function using these values, and other values of your choosing.

Hint: You may need to use multiple `if` *statements, possibly nested inside each other. Remember that every time you call an* `if` *statement, you need to indent the code inside the* `if` *statement.*

7. Submitting your lab

Once you have written code which answers each of the problems listed above, you will need to submit your work for grading. You will do these steps at the end of each of the labs in the coming weeks.

- Verify that all of the functions and variables you created are copied into the prepared lab notebook.
- Make sure that you haven't changed anything in the first cell, or the names of any of the functions or variables in the lab notebook.
- Check that your name and BYU NetID have been entered in the lab notebook where indicated, inside the quotation marks, and with no extra spaces.
- *IMPORTANT STEP:* In order to have your lab graded by the grading software, you will need to make sure that every cell in your notebook can be executed without producing an error message. Go to the "Runtime" menu at the top of this notebook, and select "Restart and run all". If any of the cells produce error messages, you will either need to fix the error(s) or delete the code that is causing the error(s) (see the document "Common Errors and Error Messages" on Learning Suite for help interpreting error messages). Then use "Restart and run all" again to see if there are any new errors. Repeat this until no new error messages show up. *If you leave a cell that produces an error in your notebook then your work may not be graded properly and you may receive no credit for the lab!*
- Download the prepared lab notebook `Copy of Intro_to_Python_I` with inputted work (*not your practice notebook!*) to your computer as a file in the `.py` format by selecting

  `File > Download .py`

  in the menu at the top of your lab notebook. You should now have a file named `Copy of Intro_to_Python_I.py` saved somewhere on your computer.

  Important note: Your file must be submitted as a `.py` *file, and not a* `.ipynb` *file. The website cannot process* `.ipynb` *files.*

- In an internet browser navigate to the webpage

  `http://www.math.byu.edu:30000`

- Upload the file `Copy of Intro_to_Python_I.py` where indicated and click the *Submit* button.
- You should see your grade on the lab appear, as well as any functions which did not produce correct output under the "Feedback" section of the webpage. In rare cases you may also see more complicated error messages.
- If desired, you may go back to fix any errors and repeat the above steps to resubmit your lab. This may be done as many times as you'd like before the deadline.