```
print("Hello world")
```

Great! Now let's do a quick refresher on Phython 3

VARIABLES

Variables are created in Python when they are assigned a value. Basic variable types are int, float, string, and Boolean. Here are some examples. Run the cells to execute the Python to create and assign values to variables.

```
a = 2
b = 3.142
s = "Say Howdy"
bool_val = True
```

Now create a new code cell following this cell to create an integer with a name ANint and assign it a value of 1953. In the same cell use the python print command to print the value of ANint

```
ANint = 1953
print(ANint)
```

```
1953
```

We can always check the type of a created variable with Python print command:

```
print(type(a))
print(type(b))
print(type(s))
print(type(bool_val))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

Insert a code cell following this cell to verify the type of ANint

```
print(type(ANint))
```

```
<class 'int'>
```

Now, a quick overview of built in functions

```
print("a =", a, "and b =" , b)
print(a + b)      # addition
print(a * b)      # multiplication
print(a - b)      # subtraction
print(a / b)      # division
print(b ** a)     # exponentiation
print(a % b)      # modulus - returns the remainder
```

```
a = 2 and b = 3.142
5.1419999999999995
6.284
-1.142
0.6365372374283896
9.872164
2.0
```

Insert a code cell to assign a new variable aquad with a value from the product of (b-a) and (b+a). Print the result

```
aquad = (b-a) * (b+a)
print(aquad)
```

```
5.872163999999999
```

Some Boolean built in functions

```
print('hello' == 'HELLO')
print('hello' is 'hello')
print(3 != 77)
print(1 < 2)
print('four' > 'three')
```

```
False
True
True
True
False
<>:2: SyntaxWarning: "is" with 'str' literal. Did you mean "=="?
C:\Users\there\AppData\Local\Temp\ipykernel_4168\3606750855.py:2: SyntaxWarning: "is" with 'str' literal. Did you mea
  print('hello' is 'hello')
```

We can change types implicitly by value assignment or explicitly with casting

```
a = 3.142
print(type(a))
a = 3
print(type(a))
a = a*1.0
print(type(a))
a = int(a)
print(type(a))
a = 3.142
a = int(a)
print(type(a))
print(a)
a = "3.142"
print(type(a))
a = float(a)
print(type(a))
```

```
<class 'float'>
<class 'int'>
<class 'float'>
<class 'int'>
<class 'int'>
3
<class 'str'>
<class 'float'>
```

LISTS

A list is a set of values, of any type separated by commas and delimited by '[' and ']'

```
list1 = [6, 54, 89 ]
print(list1)
print(type(list1))

list2 = [3.142, 2.71828, 9.8 ]
print(list2)
```

```
print(type(list2))

myname = "Peter"
list3 = ["Hello", 'to', myname ]
print(list3)
myname = "Fred"
print(list3)
list3 = ["Hello", 'to', myname ]
print(list3)
print(type(list3))

list4 = [6, 5.4, "numbers", True ]
print(list4)
print(type(list4))
```

```
[6, 54, 89]
<class 'list'>
[3.142, 2.71828, 9.8]
<class 'list'>
['Hello', 'to', 'Peter']
['Hello', 'to', 'Peter']
['Hello', 'to', 'Fred']
<class 'list'>
[6, 5.4, 'numbers', True]
<class 'list'>
```

Insert a cell after this one to create a list with 5 elements. Use python range to initialize the values of the list. You may look up from the python site how to use range(). range takes a starting value, stop value, and an optional step value. Find a way to make a list, list1, with elements [7, 8, 9, 10, 11] then create a second list, list2, with 5 elements where this list is [2, 5, 8, 11, 14]. Look up range values on python.org site if necessary.

```
list1 = []
for i in range(7,12):
    list1.append(i)

list2 = []
for i in range(2,17,3):
    list2.append(i)

#VERIFICATION
if True:
    print(f"list1: {list1}")
    print(f"list2: {list2}")
```

```
list1: [7, 8, 9, 10, 11]
list2: [2, 5, 8, 11, 14]
```

We select an entry from a list or "Index" by indicating the entries or range that we want selected within square brackets:

```
num_list = [4,5,6,11]

last_num_in_list = num_list[-1]
print(last_num_in_list)

odd_from_list = [num_list[1], num_list[3]]
print(odd_from_list)
print(num_list[0])
print(num_list[2])
print(num_list[1:2])
print(num_list[0:])         #from index 0 to the end
print(num_list[:2])         #from the start to the one BEFORE index 2
```

```
11
[5, 11]
4
6
```

```
[5]
[4, 5, 6, 11]
[4, 5]
```

Insert a cell after this cell that prints out the last entry in the list list2

```
print(list2[-1])
```

```
14
```

Indexing characters in strings is handled the same way for string type variables.

Branching

```
# if ... else ...

value = 4
threshold = 5
print("value = ", value, "and threshold = ", threshold)

if value > threshold :
    print("above threshold")
else :
    print("below threshold")
```

```
value =  4 and threshold =  5
below threshold
```

```
# while loop
n = 10
cur_sum = 0
# sum of n  numbers
i = 1
while  i <= n :
    cur_sum = cur_sum + i
    i = i + 1
print("The sum of the numbers from 1 to", n, "is ", cur_sum)
```

```
The sum of the numbers from 1 to 10 is  55
```

```
# for loops

print("\nExample 1\n")
for i in [1,2,3] :
    print(i)

print("\nExample 2\n")
for name in ["Tom", "Dick", "Harry"] :
    print(name)

print("\nExample 3\n")
for name in ["Tom", 42, 3.142] :
    print(name)

print("\nExample 4\n")
for i in range(3) :
    print(i)

print("\nExample 5\n")
for i in range(1,4) :
    print(i)

print("\nExample 6\n")
for i in range(2, 11, 2) :
```

```
    print(i)

print("\nExample 7\n")
for i in "ABCDE" :
    print(i)

print("\nExample 8\n")
longString = "The quick brown fox jumped over the lazy sleeping dog"
for word in longString.split() :
    print(word)
```

```
Example 1

1
2
3

Example 2

Tom
Dick
Harry

Example 3

Tom
42
3.142

Example 4

0
1
2

Example 5

1
2
3

Example 6

2
4
6
8
10

Example 7

A
B
C
D
E

Example 8

The
quick
brown
fox
jumped
over
the
lazy
sleeping
```

Functions

```
def get_item_count(items_str,sep):
    '''
    This function, named get_item_count, takes a string with a list of items and the character that they're separate
    '''
    items_list = items_str.split(sep)
    num_items = len(items_list)
    return num_items

items_owned = "bicycle;television;solar_panel;table"
print(get_item_count(items_owned,';'))
```

```
4
```

Libraries

Libraries or packages in python are collections of functions that can be installed using a python package manager (eg. pip) and subsequently repeatedly called

The following cell uses the $ notation to run a program. In this case the program is the package manager pip. In this example pip is told to install the desired package. You need to specify the package name. Some packages that we will use regularly in this course are: numpy, matplotlib, tensorflow, keras, pandas, scipy. The name of the package would replace below

```
$ pip install <package name>
```

Often we use shorthand names for packages to keep python lines short. See pd, np, and plt below.

```
# some example commom library imports
import csv
import json
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

FILE I/O

We will cover basic file I/O in Python next. Python file I/O is simple but unique. Most often in this class we will use more powerful file I/O capabilities of numpy and pandas. For completeness, here are Python File I/O basic instructions:

Python has two types of files: text files and binary files.

Text files are structured as a sequence of lines, where each line includes a sequence of characters.

Each line is terminated with a special character, called the EOL or End of Line character. There are several types, but the most common is the comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun.

A backslash character can also be used, and it tells the interpreter that the next character – following the slash – should be treated as a new line. This character is useful when you don't want to start a new line in the text itself but in the code.

A binary file is any type of file that is not a text file. Because of their nature, binary files can only be processed by an application that understands the file's structure. In other words, they must be applications that can read and interpret binary.

Reading from File

In Python, files are read using the open() method. This is one of Python's built-in methods, made for opening files.

The open() function takes two arguments: a filename and a file opening mode. The filename points to the path of the file on your computer, while the file opening mode is used to tell the open() function how we plan to interact with the file.

By default, the file opening mode is set to read-only, meaning we'll only have permission to open and examine the contents of the file. If we want to do anything other than just read a file, we'll need to manually tell Python what we intend to do with it:

- 'r' – Read Mode: This is the default mode for open(). The file is opened and a pointer is positioned at the beginning of the file's content.
- 'w' – Write Mode: Using this mode will overwrite any existing content in a file. If the given file does not exist, a new one will be created.
- 'r+' – Read/Write Mode: Use this mode if you need to simultaneously read and write to a file.
- 'a' – Append Mode: With this mode the user can append the data without overwriting any already existing data in the file.
- 'a+' – Append and Read Mode: In this mode you can read and append the data without overwriting the original file.
- 'x' – Exclusive Creating Mode: This mode is for the sole purpose of creating new files. Use this mode if you know the file to be written doesn't exist beforehand.

Once file is opened we can read from it (or write to it, etc)

Filenames can refer to files in the current working directory (CWD) or can have file path information. In python such as infile.txt, logfile.txt, foo.csv, bar.jpg do not have file path and thus are assumed to exist or are created in the current working directory. File names such as C:/Users/bob/Desktop/itsafile.tx (windows) or /Users/bob/Desktop/itsafile.txt (linux/mac) contain full path information for python to locate the file. Note python prefers the forward slash path nomenclature rather than the windows back slash nomenclature.

Writing to a File

Before we can write to a file in Python, it must first be opened in write mode.

In Python, write to file using the open() method. You'll need to pass both a filename and a special character that tells Python we intend to write to the file.

Closing a File

After opening a file in Python, once you are done reading from it or writing to it, it's important to close the file. Closing a file ensures that the program can no longer access its contents.

Since we will be using Colab operating on Google Drive files, we will practice out File IO in Colab. We first need to learn the essentials of Colab storage

Jupyter in Colab

Now, let's do a simple tour of Jupyter in Colab. We will start with addressing files in Colab. By default, files in Colab are located in /content. First let's use the Python call to linux command ls to show the directory contents for /content

```
%ls /content
```

```
sample_data/
```

Double-click (or enter) to edit

```
# open the file in write mode
myfile = open("sample.txt",'w')

myfile.write("Hello! Hello!")
myfile.write("Notice that you need to have a newline character to get the next item on a new line \n")
myfile.writelines("writelines writes whole strings of lines\nHello\nHello\nHello\n")
myfile.write("Howdy!")

# close the file
myfile.close()
```

```
%ls /content
```

```
sample_data/  sample.txt
```

```
# read.py
# loading a file with open()
myfile = open("sample.txt")

# reading each line of the file and printing to the console
for line in myfile:
    print(line)
```

```
Hello! Hello!Notice that you need to have a newline character to get the next item on a new line

writelines writes whole strings of lines

Hello

Hello

Hello

Howdy!
```

Now, that you now know how to load a Jupyter notebook in Colab and you know how to do file I/O in python in Jupiter, let'd do some datafile movement and manipulation in Jupyter.

Let's start with how you will load data into your Jupyter Notebook in Colab. You will copy datafiles from Canvas to your Google Drive in MyDrive. Since you have already done that for today's datafile, let's go get the data.

The easiest way to get data from your Google Drive into Colab is to mount your Google Drive for use in the Jupyter notebook: Run the following cell. If prompted for an authorization code, follow the URL link, log into your Google account, copy the authorization code, and paste that code into the box below and press return. You should get a response "Mounted at /content/drive"

```
from google.colab import drive

# This may prompt for authorization.
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Now use unix ls command to make sure that the file we will load is on MyDrive. Run the following cell to verify. If the file is not listed, go back and verify the upload of the file to your google drive and rerun the cells up to this point ...

```
!ls /content/drive/MyDrive/ECEN250-F25-Lab1-Data.csv
```

```
/content/drive/MyDrive/ECEN250-F25-Lab1-Data.csv
```

We could use Python reader to read the csv data and writer to output or we could read with Python dictionary, but instead we will use the Python utility NumPy. NumPy is a library for mathmatical manipulations of arrays and matrices. First we import numpy into our notebook, then we use genfromtext() to read the CSV file. We can then use the implied print command to look at the result.

```
import numpy as np
gps=np.genfromtxt('/content/drive/MyDrive/ECEN250-F25-Lab1-Data.csv',delimiter=',')
gps
```

```
array([[ 30.6098961 , -96.34038089],
       [ 29.7392039 , -95.4644667 ],
       [ 30.27470226, -97.7403505 ],
       [ 32.77537027, -96.80895863],
       [ 29.42573914, -98.48632939]])
```

This data should be reconizable as GPS coordinates. Let's compute distances between these locations. Lets extract the first element of the array:

```
kylefield=gps[0]
```

Now we are going to install a new package to help us compute distances between GPS locations. Haversine computes distances between two points on a sphere. This utility is not pre-installed in the python packaged used on Google Colab so we will need to use the Python installer pip to install this utility into the current Colab python for this notebook. Once installed we import it into the notebook with hs as shorthand name:

```
!pip install haversine
import haversine as hs
```

```
Collecting haversine
  Downloading haversine-2.9.0-py2.py3-none-any.whl.metadata (5.8 kB)
Downloading haversine-2.9.0-py2.py3-none-any.whl (7.7 kB)
Installing collected packages: haversine
Successfully installed haversine-2.9.0
```

Now we can use haversine to compute the distance between two locations. First, compute the distance between Kyle Field and Kyle Field. The second line is a python implied print.

```
kyle2kyle = hs.haversine(kylefield, kylefield)
kyle2kyle
```

```
0.0
```

The second gps coordinate is for the Houson Galleria. Complete the following cell to extract the second element of the gps array using the first line of the cell and then use haversine to calculate the distance between the Galleria and Kyle Field. Finally add an implied print for galleria2kyle:

```
galleria= gps[1]
galleria2kyle = hs.haversine(galleria, kylefield)
galleria2kyle
```

```
128.30733648051216
```

Now, add a code cell after this cell and in that cell insert code to extract the third element of the array and compute the distance from that location to Kyle field. You add a cell by positioning your curser at the bottom of this cell and when the +code and +text buttons appear, press select the code button. Alternatively you can select + Code on the secondary taskbar at the top of the notebook or you can do Insert, Code Cell from the primary taskbar at the top of the notebook.

```
thirdloc= gps[2]
```

```
thirdloc2kyle = hs.haversine(thirdloc, kylefield)
thirdloc2kyle
```

```
139.28724832593284
```

Now, repeat the same process for the two remaining gps coordinates.

```
fourthloc= gps[3]
fourthloc2kyle = hs.haversine(fourthloc, kylefield)
fourthloc2kyle
```

```
244.8363537308462
```

```
fifthloc= gps[4]
fifthloc2kyle = hs.haversine(fifthloc, kylefield)
fifthloc2kyle
```

```
244.994369200396
```

As we have seen, NumPy operates on numerical arrays. We can use the following to create an array 2-dimensional array with two rows (each having 4 elements) We can use shape to give the dimensional characteristics of the numpy array arr. Shape will be very useful to us as we try to convert and combine arrays of different dimensional characteristics.

Notice how we used explicit print in python instead of the implicit print we used above.

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

print("arr has shape of ",arr.shape)
```

```
arr has shape of  (2, 4)
```

Now enter create a code cell below this cell with the python to print the first element of the last row of arr

```
print(arr[1][0])
```

```
5
```

Next, let's modify the next cell to create a python list with 4 entries for the distances from Kyle Field for the 4 other gps locations that you already computed:

```
dist2kyle = [galleria2kyle, thirdloc2kyle, fourthloc2kyle, fifthloc2kyle]
dist2kyle
```

```
[128.30733648051216, 139.28724832593284, 244.8363537308462, 244.994369200396]
```

Finally let's compute the sum of these distances. Modify the following code cell to compute and print the sum.

```
Sum = sum(dist2kyle)
print(Sum)
```

```
757.4253077376873
```

After running your last cell, prepare to submit your notebook to canvas. Since it is essential that you have all your cells executed and results printed. If you have some cells run out of order or other issues, you can do Edit > Clear all outputs then return to the top of the notebook and rerun each cell in order.

When you are happy with your results, create a PDF of your notebook by doing File > Print and specify print to pdf. Finally upload that pdf to canvas for your Lab 1 submission