In [ ]:
```
Siddhanth Kumar
Section 102
```

Workshop 2: Control Structures

Submit this notebook to bCourses to receive a grade for this Workshop.

Please complete workshop activities in code cells in this iPython notebook. The activities titled **Practice** are purely for you to explore Python, and no particular output is expected. Some of them have some code written, and you should try to modify it in different ways to understand how it works. Although no particular output is expected at submission time, it is *highly* recommended that you read and work through the practice activities before or alongside the exercises. However, the activities titled **Exercise** have specific tasks and specific outputs expected. Include comments in your code when necessary. Enter your name in the cell at the top of the notebook.

The workshop should be submitted on bCourses under the Assignments tab (both the .ipynb and .pdf files). Please label it by your student ID number (SIS ID)

Practice

Exercises start here

Loops and Lists

In Python, we can create a list of things (integers, floats, strings, more lists...) using the following syntax:

In [79]:
```
#Make sure the elements of your list are separated by commas -- and don't forget the bra
people = ['Rick', 'Morty', 'Beth', 'Summer', 'Jerry', 'Birdperson', 'Mr. Meeseeks']
some_numbers = [2, 3, 4.3, 9e3, 1/2]

print(people, some_numbers)
```

```
['Rick', 'Morty', 'Beth', 'Summer', 'Jerry', 'Birdperson', 'Mr. Meeseeks'] [2, 3, 4.3, 9
000.0, 0.5]
```

You can put a mix of whatever you want into a list, but we mostly only need lists of numbers in computational physics. If we want to go through each thing in a list (things in lists are called *elements*), we can use a `for` loop. Note that the first line with the word `for` has a colon at the end, and everything below it is indented.

In [80]:
```
'''When Python sees a for loop, it will go through each element in a list (in order) and
   commands are on the indented lines.  Below, we see that he name of the list (people)
   The variable name (person) between the "for" and "in" keywords is assigned to the val
   as the loop goes through the list.'''

for person in people:
    print(person)

'''Here, the variable name (number) is assigned to each element of the list (some_number
   In each iteration of the loop, the current value of number is squared and printed.'''
for number in some_numbers:
    print(number**2)
```

```
Rick
Morty
Beth
Summer
Jerry
Birdperson
Mr. Meeseeks
4
9
18.49
81000000.0
0.25
```

We can grab a specific element of the list by referencing its index (the first element has an index of 0, the second is 1, the third is 2, etc.).

In [81]:
```
print(people[0], people[3], some_numbers[1])
```

```
Rick Summer 3
```

There's another really common way of going through lists in Python. We'll use a `for` loop again, but this time we'll grab the elements by referencing indices. To do this, we'll use the pre-defined `range(N)` which gives us a list of numbers from 0 to N - 1.

In [82]:
```
#range(5) is equivalent to [0, 1, 2, 3, 4]
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

In [83]:
```
for i in range(5):
    print(some_numbers[i])
```

```
2
3
4.3
9000.0
0.5
```

If you want to go through every element of a list using this `for x in range(N)` syntax, it's useful to know the length of the list.

In [84]:
```
#len(list) returns the integer-valued length of the list
for i in range(len(people)):
    print(people[i])
```

```
Rick
Morty
Beth
Summer
Jerry
Birdperson
Mr. Meeseeks
```

Another useful thing you can do is check if an element is in a list using the `in` keyword. If the element is in the list, Python returns the Boolean value `True`. If not, it returns `False`.

In [85]:
```
'Squanchy' in people
```

Out[85]:
```
False
```

In [86]:
```
'Morty' in people
```

Out[86]:
```
True
```

In [87]:
```
'Squanchy' not in people
```

Out[87]:
```
True
```

Conditional statements

We can check if two things are equivalent using `==` (equal to) in Python.

In [88]:
```
5 == 5
```

Out[88]:
```
True
```

In [89]:
```
5 == 4
```

Out[89]:
```
False
```

If they are equivalent, the expression returns `True`, otherwise it returns `False`. We can then use these `True` and `False` expressions to write something called an `if` statement.

The syntax of an `if` statement is `if conditional:` where `conditional` is some expression that will either be `True` or `False`. The (indented) code below an `if` statement will only run if the **conditional expression** is `True`. For example, run the code below with two equal numbers and then two different numbers.

In [90]:
```
if 5 == 5:
    print("This is true")
```

```
This is true
```

We also have the option of adding an `else` statement *after* any `if` statement. The (indented) code below an `else` statement will only run if the conditional expression of the `if` statement is `False`.

In [91]:
```
if 5 == 4:
    print("This is true")
else:
    print("This is false")
```

```
This is false
```

There's also something called an `else if` statement, written as `elif` in Python code. But first, let's look at a few more ways we can compare things using `<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to), and `!=` (not equal to). All of these comparisons return either `True` or `False`.

In [92]:
```
5 < 4
```

Out[92]:
```
False
```

In [93]:
```
5 > 4
```

```
True
```

Out[93]:

In [94]:
```
5 >= 6
```

Out[94]:
```
False
```

In [95]:
```
5 <= 5
```

Out[95]:
```
True
```

In [96]:
```
5 != 6
```

Out[96]:
```
True
```

Alright, so suppose we want to compare the number `a` to multiples of the number `b`. I'm not sure why we'd want to do this, but it's a good way to demonstrate `elif` statements. If we just write a bunch of `if` statements, it's possible that all of them are `True`.

In [97]:
```
a = 5
b = 6

if a < b:
  print("{} is less than {}".format(a, b))

if a < 2*b:
  print("{} is less than 2*{}".format(a, b))

if a < 3*b:
  print("{} is less than 3*{}".format(a, b))
```

```
5 is less than 6
5 is less than 2*6
5 is less than 3*6
```

If we only want one thing to be printed, we can use `elif` statements after the first `if` statement -- you always need to start with an `if` statement. The syntax is `elif conditional:`, followed by some indented code that runs if the conditional expression is `True`. The difference: once an `elif` (or the first `if`) conditional expression is `True`, the following `elif` statements and their code are ignored.

In [98]:
```
a = 5
b = 4

if a < b:  #False, Python moves on to the next elif statement
  print("{} is less than {}".format(a, b))

elif a < 2*b: #True, Python runs the indented code below
  print("{} is less than 2*{}".format(a, b))

elif a < 3*b: #The above elif statement was True, so this is ignored
  print("{} is less than 3*{}".format(a, b))
```

```
5 is less than 2*4
```

You can put as many `elif` statements as you'd like after the first `if` statement. As always, you're free to throw on an `else` statement at the end.

In [99]:
```
a = 5
b = 1

if a < b:  #False, move on to the next one
```

```
  print("{} is less than {}".format(a, b))

elif a < 2*b:  #False, move on to the next one
  print("{} is less than 2*{}".format(a, b))

elif a < 3*b:  #False, move on to the next one
  print("{} is less than 3*{}".format(a, b))

else:  #If all the above if and elif conditionals are False, then we run the indented co
  print("Everything above the else statement is False")
```

```
Everything above the else statement is False
```

Combining Loops and Conditionals

Sometimes we want to go through the elements of a list and do something only if a certain condition is met. In this case, we'll want to combine `if` statements with `for` loops.

In [100…

```
people = ['Rick', 'Morty', 'Beth', 'Summer', 'Jerry', 'Birdperson', 'Mr. Meeseeks']

for name in people:
    if name == 'Mr. Meeseeks':
        print("I'm {}, look at me!!!".format(name))
    else:
        print("Hi, I'm {}".format(name))
```

```
Hi, I'm Rick
Hi, I'm Morty
Hi, I'm Beth
Hi, I'm Summer
Hi, I'm Jerry
Hi, I'm Birdperson
I'm Mr. Meeseeks, look at me!!!
```

For a more practical example, suppose we want to only print the even numbers in a list.

In [101…

```
numbers = [3, 4, 11, 2, 7]

for number in numbers:
    if (number % 2) == 0:  #The % (modulus) operator gives the remainder after integer d
        print(number)
```

```
4
2
```

Now suppose we want to create two new lists containing the even and odd numbers. We can use `.append()` to add things to an existing list.

In [102…

```
even_numbers = []  #Create an empty list
odd_numbers = []

for number in numbers:
    if (number % 2) == 0:            # If the number is even,
        even_numbers.append(number)  # put it in the even list
    else:                            # otherwise
        odd_numbers.append(number)   # put it in the odd list

print(even_numbers)
print(odd_numbers)
```

```
[4, 2]
[3, 11, 7]
```

Exercises

Write your solutions to these exercises in code cells in this notebook, and submit your finished notebook to bCourses.

In your file, separate and label each solution with a comment or a markdown cell marking the exercise.

Here are the exercises. Each one will require you to use conditionals, loops, and/or list comprehensions to solve a problem and print a solution. Many of these programs are short and don't require much if any commenting to explain them. For longer or trickier solutions, however, comments may be appropriate to make your code and thought process clear. Use your best judgment.

Short Conditional and Loop Exercises

Exercise 1 Write a program that prints multiples of 5 which are evenly divisible by 7 (zero remainder after division), between 1500 and 2700 (both included).

Hint: Try seeing what `range(10, 30)` and `range(10, 30, 4)` gives you; remember this `for i in range(...)` syntax.

In [103…
```
#The for loop below prints numbers between 10 (inclusive) and 30 (exclusive), skipping b
for i in range(10, 30, 4):
    print(i)
#Try using something similar, but use an 'if' statement to also check if the number is e

print('------------------')

for i in range(10, 30, 4):
    if i % 7 == 0:
        print(i)

#Your code here
```

```
10
14
18
22
26
------------------
14
```

Exercise 2 Write a Python program to construct the following pattern, using a `for` loop.

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
```

In [4]:
```
star = '* '

print(1*star)
print(2*star)
print(3*star)

print('----------------------')
```

```
for i in range(1, 7):
    print(i*star)


#Now try automating this printing process using a for loop
#Your code here
```

```
*
* *
* * *
----------------------
*
* *
* * *
* * * *
* * * * *
* * * * * *
```

Exercise 3 Use a `for` loop to reverse the string `'!scisyhP'` (don't just use `'!scisyhP'[::-1]`), and print the result. Remember you can think of a string as a list of characters.

In [12]:
```
string = '!scisyhP'

new_string = ''  #This is just an empty string
new_string = new_string + string[7] + string[6] + string[5]  #I can add characters to my
print(new_string)

#Now try automating this reversal process using a for loop
#Your code here

finalWord = []

for i in range(len(string)-1, -1, -1):
    finalWord.append(string[i])

print(finalWord)
```

```
Phy
['P', 'h', 'y', 's', 'i', 'c', 's', '!']
```

Exercise 4 Write a program to count the number of even and odd numbers in the list `numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]`. Expected Output :

```
Number of even numbers : 4
Number of odd numbers : 5
```

In [14]:
```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

#Use these variables to store the number of even and odd numbers
number_even = 0
number_odd  = 0

for number in numbers:
    print(number)
    if number % 2 == 0:
        number_even += 1
    else:
        number_odd += 1
    #Now using an if/else statement, check if each number is even or odd, and update the

print("Number of even numbers : {}".format(number_even))
print("Number of odd numbers : {}".format(number_odd))
```

```
1
```

```
2
3
4
5
6
7
8
9
Number of even numbers : 4
Number of odd numbers : 5
```

Exercise 5 Write a program that prints all the numbers from 0 to 6 except 3 and 6. Use the `continue` statement. When using a loop in Python, if the `continue` statement is reached, Python then skips to the next iteration of the loop.

In [15]:
```
#Example: only print odd numbers from 1 to 10
for number in range(1,11):
    if number % 2 == 0:  #if this is True (i.e. number is even),
        continue         #then skip to the next iteration of the loop

    print(number)        #the number is only printed when the continue statement is not
```

```
1
3
5
7
9
```

In [16]:
```
#Your code here
for number in range(0, 7):
    if number == 3 or number == 6:
        continue
    print(number)
```

```
0
1
2
4
5
```

Exercise 6: Fibonacci Sequence The Fibonacci sequence is a sequence of integers defined by the following recursion relation. The n-th integer a_n is defined in terms of previous integers of the sequence as

$$a_n = a_{n-1} + a_{n-2}$$

and $a_0 = 0$ and $a_1 = 1$. So the first few numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, ... Write a program to print the first 10 numbers in the Fibonacci series. Expected output:

```
0 1 1 2 3 5 8 13 21 34
```

Hint: Two (or more) variables can be assigned simultaneously. For instance, to swap the values of `a` and `b`, you can write `a, b = b, a`.

In [24]:
```
a0 = 0
a1 = 1
a2 = a0 + a1
a3 = a1 + a2
print(a0, a1, a2, a3)


#It's tedious typing out these summations; try automating this process using a for loop
#Your code here
```

```
sequence = [0, 1]

for i in range(2, 10):
    aNext = sequence[i - 2] + sequence[i - 1]
    sequence.append(aNext)

print(sequence)
```

```
0 1 1 2
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Exercise 7 Write a program which takes two digits m (row) and n (column) and generates a two dimensional array. The element value in the i-th row and j-th column of the array should be i*j. Print this array. (Note: i = 0,1.., m-1 and j = 0,1, n-1.) Start with this code for a 3x4 array:

Expected output:

```
    [[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6]]
```

Hint: You may use a nested `for` loops (a `for` loop inside another `for` loops) or look up "nested list comprehension".

In [30]:
```
m = 3 # number of rows
n = 4 # number of columns

my_array = []

#Your code here; try using a for loop over range(n) inside a for loop over range(m)

for row in range(0, m):
    my_array.append([])
    for column in range(0, n):
        my_array[row].append(row*column)


print(my_array)
```

```
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6]]
```

Exercise 8: Sinc Function

[Adapted from Ayaras, Problem 1-4] The mathematical function $\mathrm{sinc}(x)$ appears when deriving the theory for double-slit experiments.

$$\mathrm{sinc}(x) \equiv \frac{\sin x}{x}$$

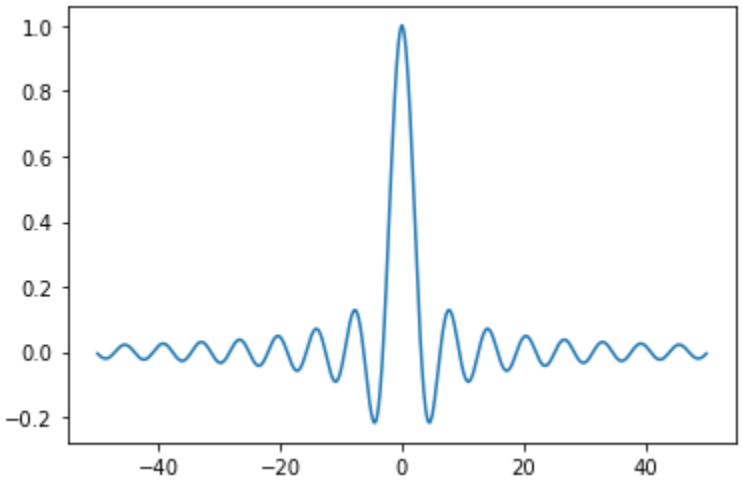
Write a python function for Make sure that your function handles $x = 0$ correctly. Type your code in a code cell in this iPython notebook. You can use the template below to get started.

Once you are happy with your function, write code to plot it. Use `numpy.linspace` to generate 1001 x values between -50 and 50 (inclusive), and use your `sinc` function to generate corresponding y values. Then plot the results on a new figure with appropriate limits and labels.

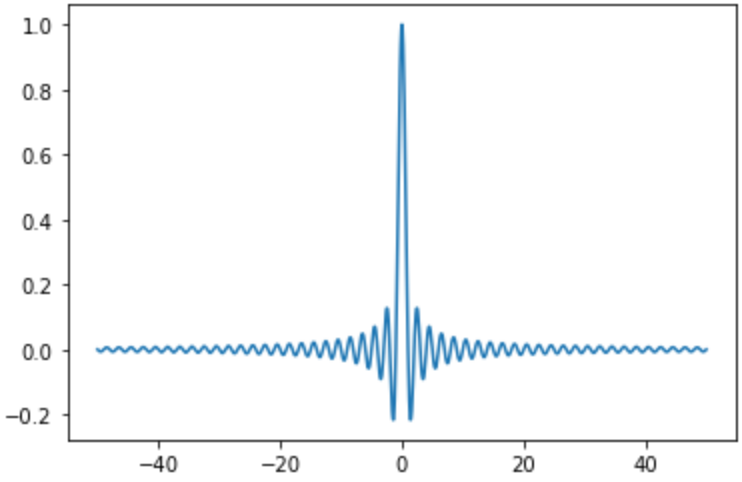
Hint

At $x = 0$, $\mathrm{sinc}(x)$ should give the value of $\lim_{x \to 0} \frac{\sin x}{x}$ so that the function is continuous. You'll probably used an if statement in your definition of $\mathrm{sinc}(x)$, so be careful when finding the y values to make the plot. Doing something like `y_values = sinc(x_values)` won't work in this case, so you'll need to evaluate each element of `x_values` individually.

In [39]:
```
import numpy as np
import matplotlib.pyplot as plt

def sinc(x):
    """ Takes a real number x and returns the continuous function
        sinc(x) = sin(x)/x. """

    # Your code here!
    if x != 0:
        return np.sin(x)/x

x_values = np.linspace(-50, 50, 1001)

'''y_values = sinc(x_values) would give an error; numpy arrays don't play nicely with co
   Instead, we have to fill a list of y_values one element at a time.'''

y_values = []

for x in x_values:
    y = sinc(x)
    y_values.append(y)

Ys_values = []

for x in x_values:
    y = np.sinc(x)
    Ys_values.append(y)

# Plotting the sinc function

plt.plot(x_values, y_values)
plt.show()
plt.plot(x_values, Ys_values)

# Your code here!
```



Out[39]:
```
[<matplotlib.lines.Line2D at 0x7fa2799b5910>]
```



Just for fun

The `numpy` library also includes a `sinc(x)` function. If you plot it, you'll find that it has the same general shape as yours, but different intercepts. Why is this? Type "numpy.sinc?" into your interpreter or Google something like "numpy sinc" to find out!

Exercise 9: Prime Numbers

[Adapted from Ayars, Problem 1-6] Write a function called `is_prime(n)` that determines whether a number `n` is prime or not, and returns either `True` or `False` accordingly. You can assume that the argument `n` passed to any of your functions will be an integer. Remember to include descriptive doc strings for each function your write!

Then, write a function called `list_primes(n)` that uses a `for` loop and returns a list of all primes below a given number. For fun, you can also try writing this function using a `while` loop and/or list comprehension.

(Note: If you accidentally find yourself in an infinite loop, press Control+C or use the Kernel->Interrupt to get out.)

Remember to try various test cases: What if the argument passed to `is_prime` or `list_primes` is ...

- 20
- 2
- 1
- 0
- negative

Hint

Use your `is_prime` function in your various `list_primes` functions so you don't have to rewrite it each time!

In [70]:
```
def is_prime(n):

    for i in range(2,n):
        if (n%i) == 0:
            return False
    return True
```

```
    """ Determines if n is prime or not. Takes an integer n.
    Returns True if n is prime, and False otherwise. """
    # Your code here!



print(is_prime(4))
print(is_prime(5))

primes = []

def list_primes(n):
    """ Takes an integer n. Returns a list of all primes
    less than n, using a for loop """
    # Your code here!

    for i in range(0, int(n) + 1):
        if is_prime(i) == True:
            primes.append(i)
    return primes



print(list_primes(1000))
```

```
False
True
[0, 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 7
9, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 17
3, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 37
9, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479,
487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 60
1, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709,
719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 82
9, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953,
967, 971, 977, 983, 991, 997]
```

Just for fun

If you're curious about (much!) more efficient ways to find primes, check out Newman problem 2.12. Try it out if you'd like!

Practice: Debugging (Optional)

[Partly adapted from Langtangen, Exercise 2.55, 2.56]

Working alone or with a partner, run each program exactly as you see it and note the errors you get. Figure out why these programs fail and correct the errors. Run your revised programs to make sure they work.

Note: Besides demonstrating errors, some of these programs also demonstrate generally bad coding practices. Try not to emulate these examples in your own code.

Program 1:

In [74]:
```
def f(x):
    return 1+x**2;
```

Program 2:

In [77]:
```
def f(x):

    term1 = 1
    term2 = x**2
    return term1 + term2
```

Program 3: Run the code below, then Google something like "python list copy" or check out this page to learn how to fix it.

In [78]:
```
old_list = [5, 2.0, 'hi', ('bye', 'bye')]

new_list = old_list
new_list[0] = 11

print(new_list)
print(old_list)  # but I didn't want the old list to change!
```

```
[11, 2.0, 'hi', ('bye', 'bye')]
[11, 2.0, 'hi', ('bye', 'bye')]
```

You're done!

Congratulations, you've finished this week's workshop! You're welcome to leave early or get started on this week's homework.

"Conditional and Loop Exercises" in this workshop adapted from w3resource.com