Group 16

CS420(G) Computer Networks
Python programming Assignment part 1 of 2

By Sushma Gajulapalli, Sathvika Kumbham, Aishwarya Khatpe, Meghana Kandala

---

Milestone 1 – Getting oriented to the Python programming environment

---

Download and install Python 3.3.

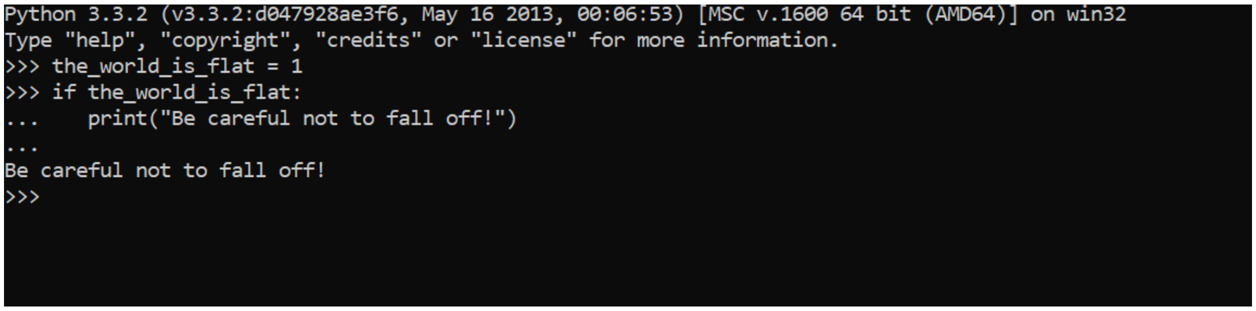
First download and install python 3.3 by following this link.

Video explanation: https://youtu.be/3zMmFz-0708 (https://youtu.be/3zMmFz-0708)

Execution of Initial Syntax

In Interactive mode, the Python interpreter prompts with '>>>' for primary and '...' for secondary. It prints version information before the first prompt. Continuation lines use '...'.

Below is a screenshot of an example.



```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
>>>
```

Python Interpreter environment

Error Handling:

- Interpreter prints error message and stack trace on error.
- In interactive mode, returns to the primary prompt; in file input mode, exit with a nonzero exit status.
- Some errors are fatal, causing nonzero exit (e.g., internal inconsistencies, memory exhaustion).
- KeyboardInterrupt raised on typing interrupt during execution.

Executable Python Scripts:

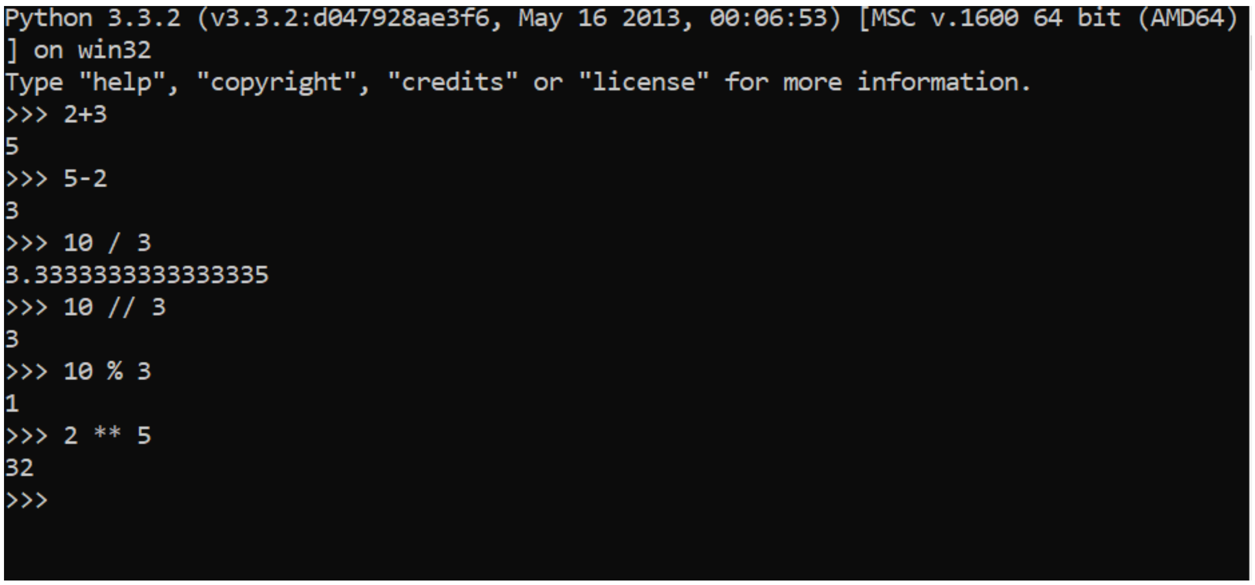
- Unix: Scripts made executable with `#! /usr/bin/env python3.3` and executable mode.
- Windows: .py files associated with python.exe; .pyw suppresses the console window.

Milestone 2 -- Demonstration

INTRODUCTION TO PYTHON

Arithmetic Operations:

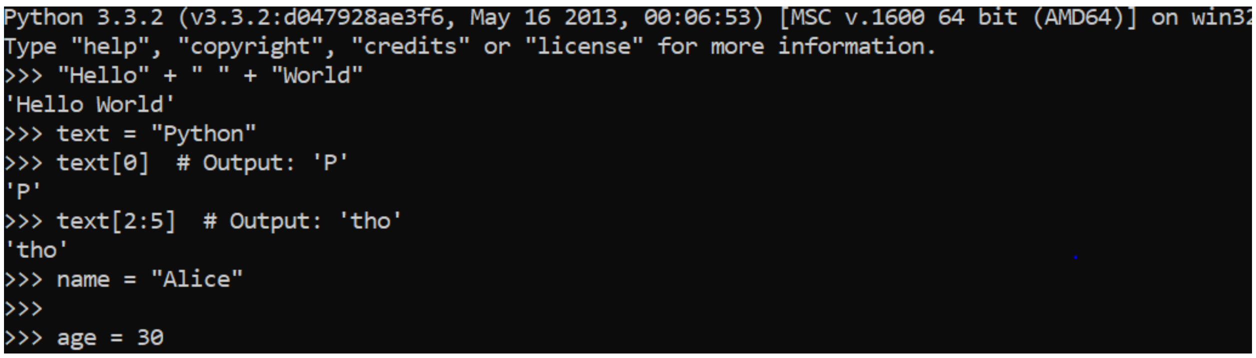
In Python, the interpreter can function as a calculator, allowing you to perform various arithmetic operations directly. When you run the provided examples in the Python interpreter, you're essentially using it as a calculator to perform calculations and observe the results. Here is a screenshot of the task I performed.



```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit (AMD64)
] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> 5-2
3
>>> 10 / 3
3.3333333333333335
>>> 10 // 3
3
>>> 10 % 3
1
>>> 2 ** 5
32
>>>
```

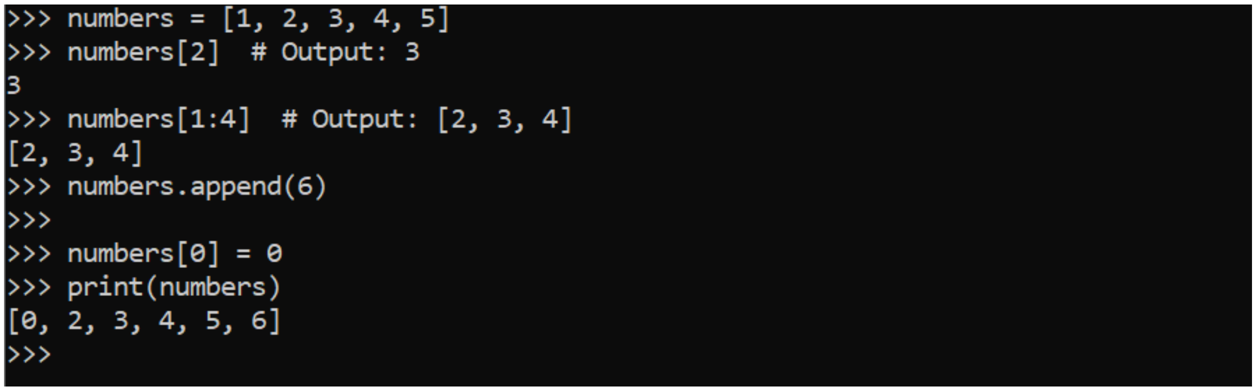
String Manipulation:

Through these examples tested in the Python interpreter(as given in the screenshot below), we explored fundamental string operations. We learned how to concatenate strings using the `+` operator, access individual characters with indexing, extract substrings through slicing, and employ f-strings for string interpolation. These operations showcase Python's versatility in handling string manipulation tasks efficiently and intuitively.



```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> "Hello" + " " + "World"
'Hello World'
>>> text = "Python"
>>> text[0]  # Output: 'P'
'P'
>>> text[2:5]  # Output: 'tho'
'tho'
>>> name = "Alice"
>>>
>>> age = 30
```

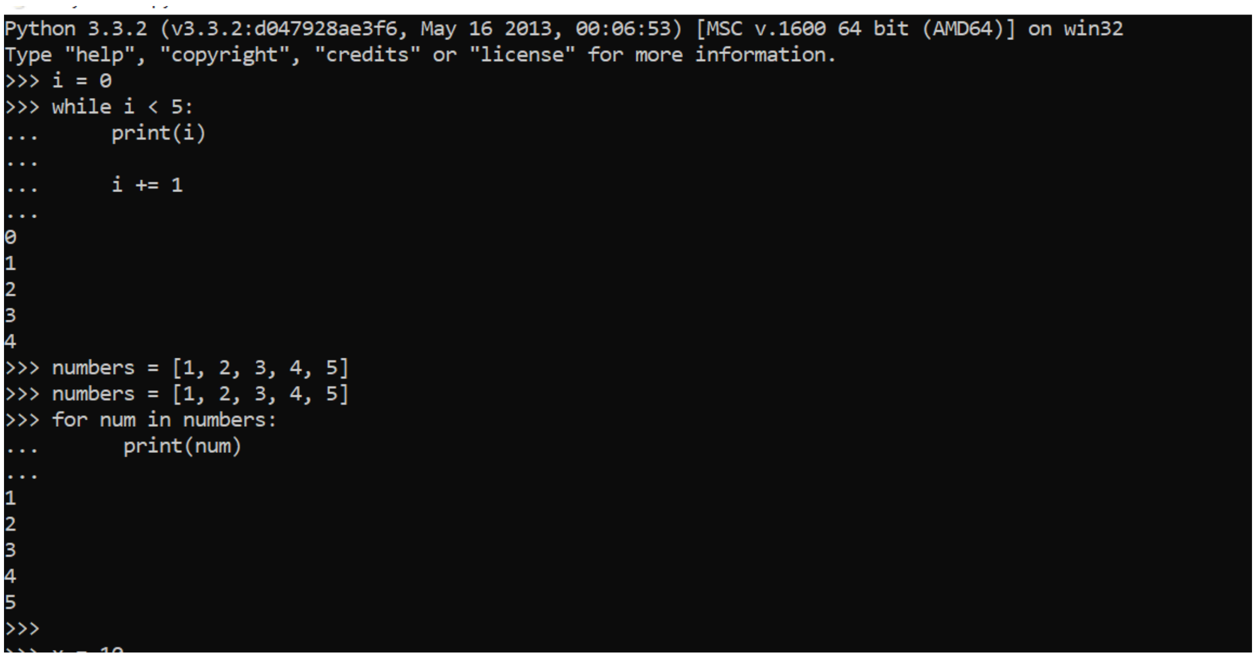
Lists:
In this exercise, we manipulated a list called numbers in Python. We accessed specific elements using indexing, retrieved a sublist using slicing, appended a new element to the list, and modified an existing element. Finally, we printed the modified list to observe the changes. This demonstrates Python's simplicity and flexibility in handling list operations, including accessing, modifying, and extending lists. The results are shown in the below screenshot.



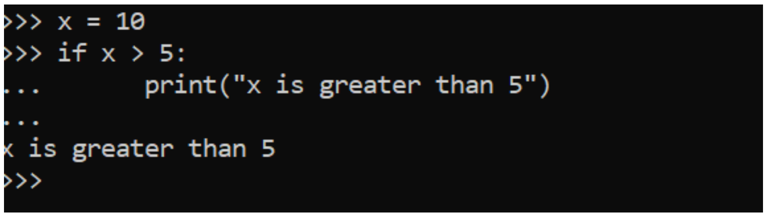
```
>>> numbers = [1, 2, 3, 4, 5]
>>> numbers[2]  # Output: 3
3
>>> numbers[1:4]  # Output: [2, 3, 4]
[2, 3, 4]
>>> numbers.append(6)
>>>
>>> numbers[0] = 0
>>> print(numbers)
[0, 2, 3, 4, 5, 6]
>>>
```

Loops and Conditional Statements:

In this Python interpreter session, we explored fundamental control flow structures. We used a while loop to iterate over a range of numbers, a for loop to iterate over elements in a list, and an if-else statement to conditionally execute code based on a comparison. These structures provide powerful ways to control program flow and make decisions based on specific conditions, enhancing the versatility and functionality of Python programs. A screenshot of the tests is given below.

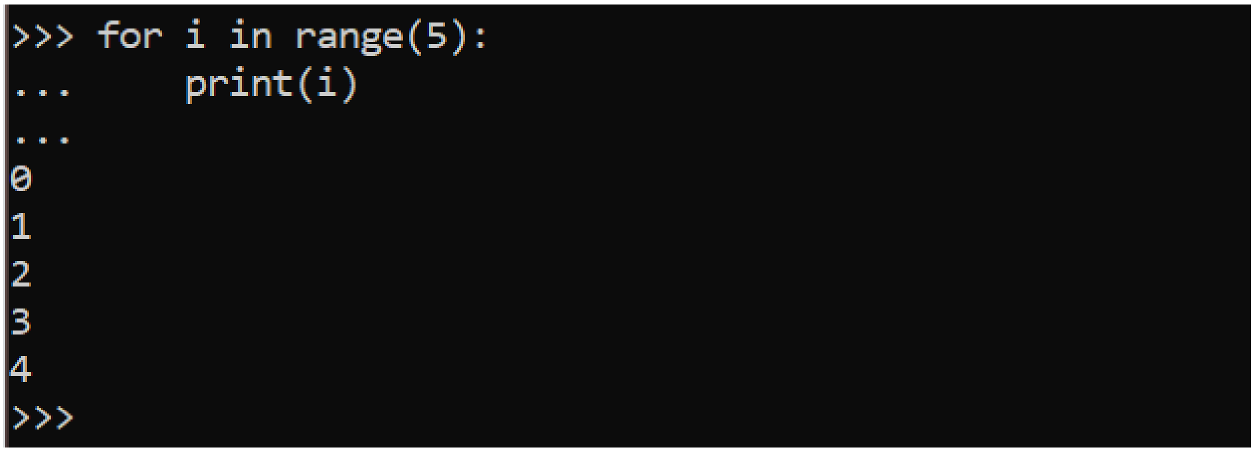


```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> i = 0
>>> while i < 5:
...     print(i)
...
...     i += 1
...
0
1
2
3
4
>>> numbers = [1, 2, 3, 4, 5]
>>> numbers = [1, 2, 3, 4, 5]
>>> for num in numbers:
...      print(num)
...
1
2
3
4
5
>>>
```



```
>>> x = 10
>>> if x > 5:
...       print("x is greater than 5")
...
x is greater than 5
>>>
```

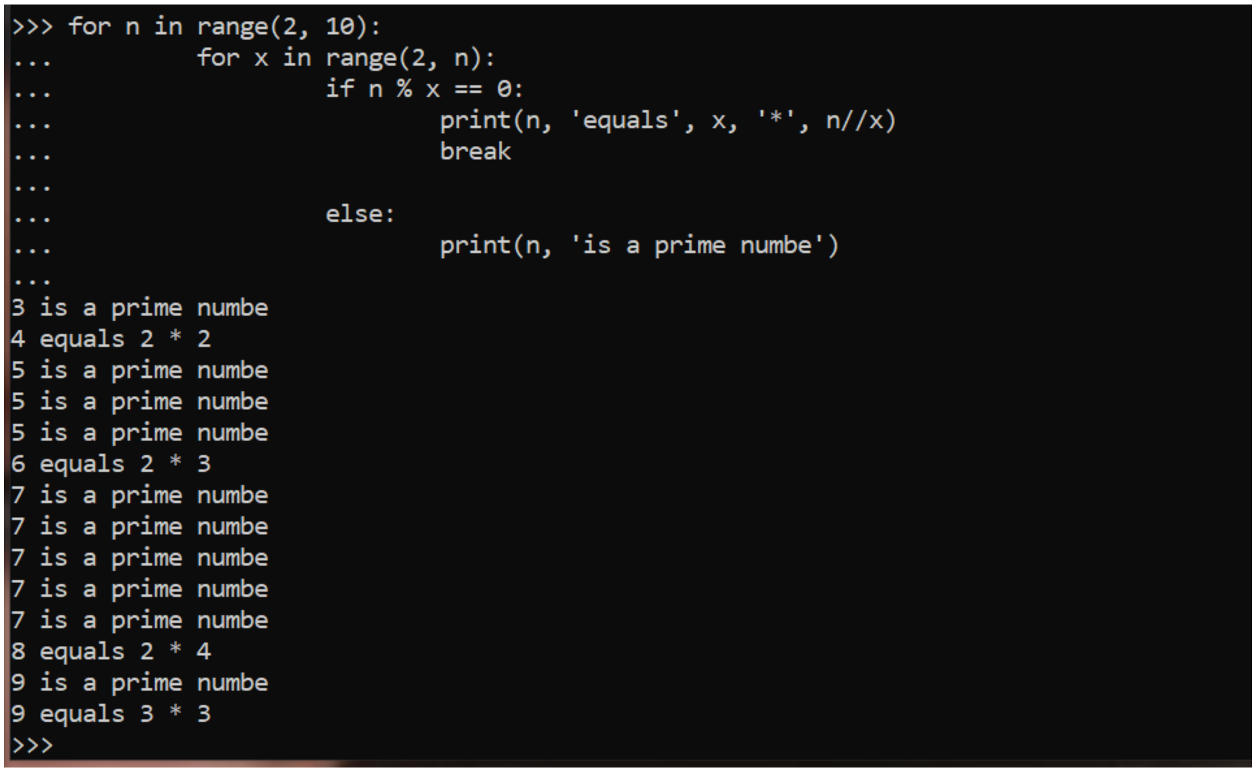
The range() Function:



```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

The **range()** function generates a sequence of numbers, which is useful for iterating a specific number of times in a loop.

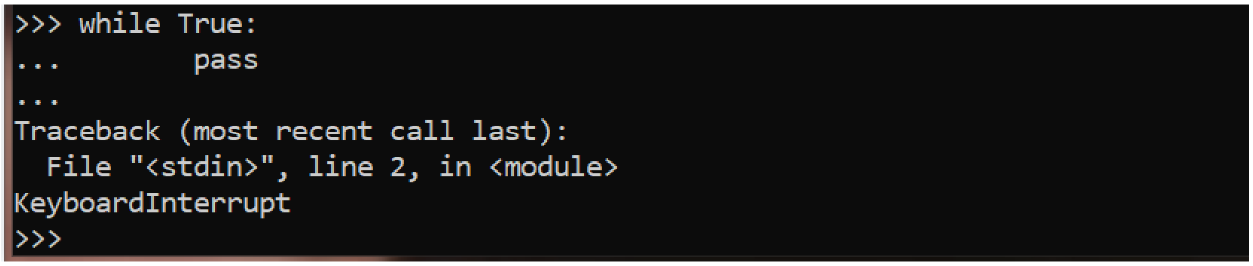
Break and continue Statements, and else Clauses on Loops:



```
>>> for n in range(2, 10):
...         for x in range(2, n):
...                 if n % x == 0:
...                         print(n, 'equals', x, '*', n//x)
...                         break
...
...                 else:
...                         print(n, 'is a prime numbe')
...
3 is a prime numbe
4 equals 2 * 2
5 is a prime numbe
5 is a prime numbe
5 is a prime numbe
6 equals 2 * 3
7 is a prime numbe
7 is a prime numbe
7 is a prime numbe
7 is a prime numbe
7 is a prime numbe
8 equals 2 * 4
9 is a prime numbe
9 equals 3 * 3
>>>
```

The **break** statement is used to exit the loop prematurely if a certain condition is met. In this example, it helps in finding prime numbers efficiently.

pass Statements:

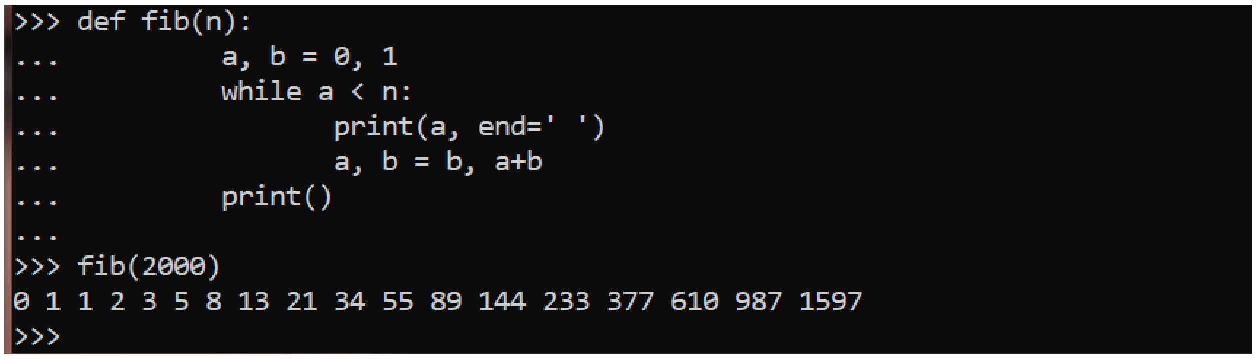


```
>>> while True:
...         pass
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
>>>
```

The **pass** statement is a no-operation placeholder, often used when syntactically a statement is required but no action is needed.

Defining Functions:

Functions allow us to encapsulate reusable pieces of code. Here, we define a function to print the Fibonacci series up to a given limit.

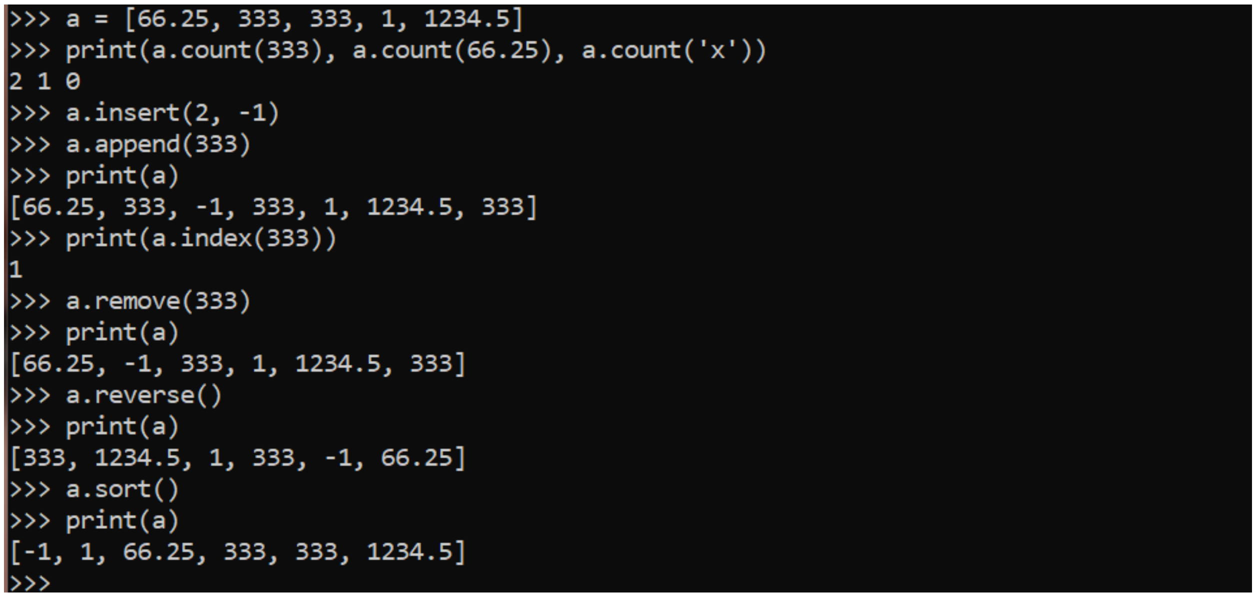


```
>>> def fib(n):
...          a, b = 0, 1
...          while a < n:
...                 print(a, end=' ')
...                 a, b = b, a+b
...          print()
...
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
>>>
```

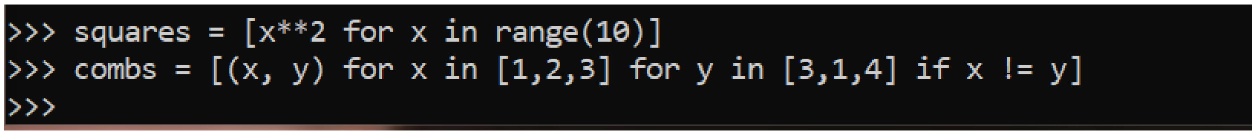
Data Structures in Python

Lists

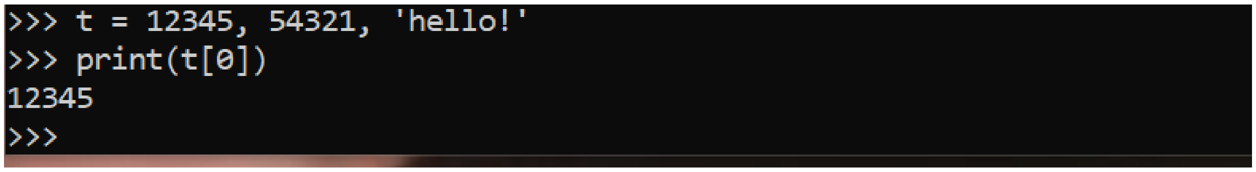
Python lists offer a variety of methods for manipulation, including appending, extending, inserting, removing, popping, indexing, counting, sorting, and reversing elements. They can be used as stacks or queues, but deque from the collection's module is preferred for efficient queue operations.


```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> print(a)
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> print(a.index(333))
1
>>> a.remove(333)
>>> print(a)
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> print(a)
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> print(a)
[-1, 1, 66.25, 333, 333, 1234.5]
>>>
```

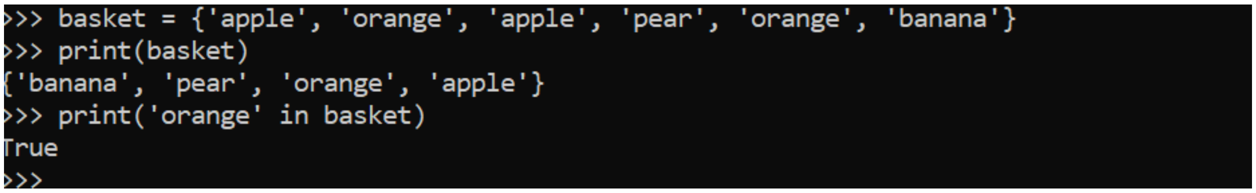
List Comprehensions
List comprehensions provide a concise way to create lists based on existing sequences, applying operations or filtering conditions.


```
>>> squares = [x**2 for x in range(10)]
>>> combs = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
>>>
```

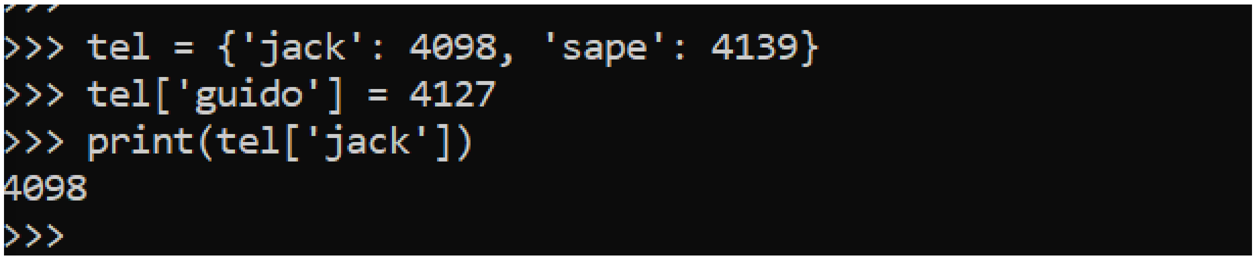
Tuples
Tuples are immutable sequences of values. They are often used for heterogeneous data and accessed via unpacking or indexing.


```
>>> t = 12345, 54321, 'hello!'
>>> print(t[0])
12345
>>>
```

Sets
Sets are unordered collections with no duplicate elements, useful for membership testing and eliminating duplicates.

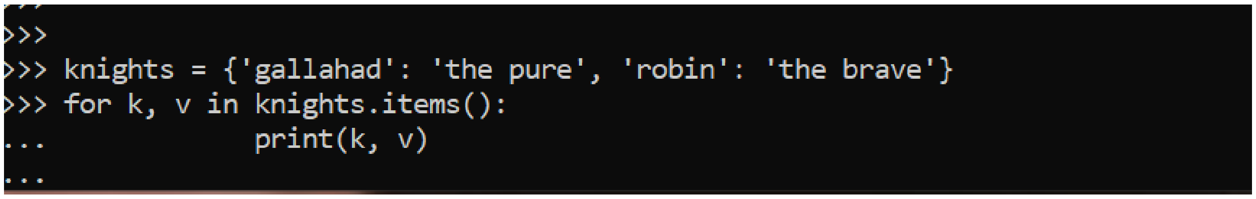

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)
{'banana', 'pear', 'orange', 'apple'}
>>> print('orange' in basket)
True
>>>
```

Dictionaries
Dictionaries are unordered collections of key-value pairs, commonly used for associative arrays. They support basic operations like storing, retrieving, and deleting elements.



```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> print(tel['jack'])
4098
>>>
```

Looping Techniques
Python provides various looping techniques including iterating over dictionaries, enumerating sequences, zipping sequences, and looping in reverse or sorted order.



```
>>>
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...             print(k, v)
...
```

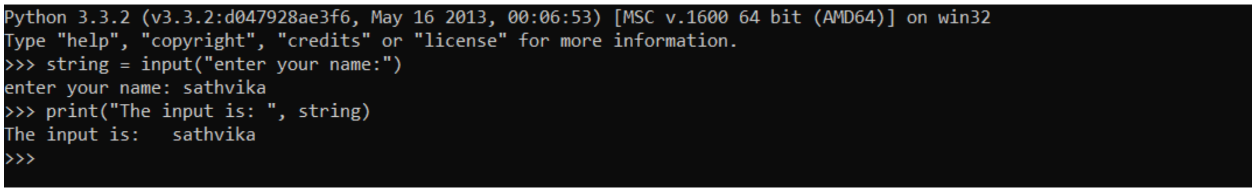
Milestone 3 -- Demonstration

Module: A module is simply a collection of related pieces of code stored in a.py file. One major work can be accomplished by defining functions, classes, variables, and other subtasks in a module.
As an example, putting all functions related to arithmetic operations into a module named "arithmetic.py" Importing: import arithmetic

Package: A package is a collection of related modules that fall under a larger category. A directory must contain the file init.py to be classified as a package. For the associated package, this file typically contains the initialization code.
For instance, Putting all operator modules, such as arithmetic, logical, etc., together into a package called "Operators," for instance

Library: A reusable code segment is referred to by the general word "library." A Python library typically consists of several linked modules and packages. Since packages can also contain modules and additional packages (sub-packages), the terms "Python package" and "package" are frequently used interchangeably.

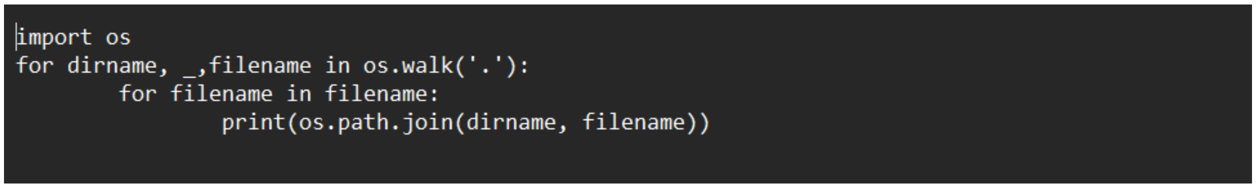
Input:
A line of text can be read from standard input in Python using two built-in functions. Standard input typically comes from the keyboard.



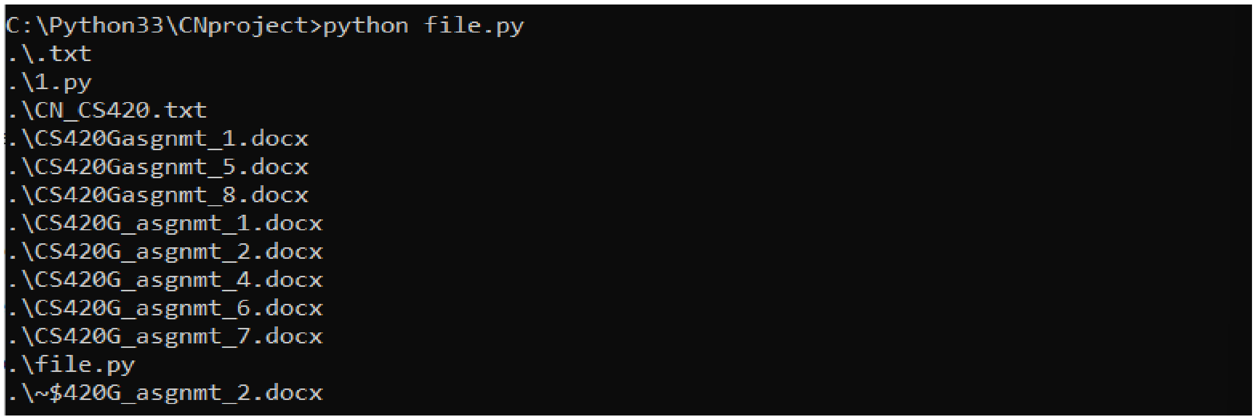
```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> string = input("enter your name:")
enter your name: sathvika
>>> print("The input is: ", string)
The input is:   sathvika
>>>
```

To print all the files present in the directory we have to save the file first and then compile using the command:
python filename.py.

Here, I saved the code with the filename as **file.py** and then executed it as you can see in the above screenshot:



```
import os
for dirname, _,filename in os.walk('.'):
        for filename in filename:
                print(os.path.join(dirname, filename))
```



```
C:\Python33\CNproject>python file.py
.\.txt
.\1.py
.\CN_CS420.txt
.\CS420Gasgnmt_1.docx
.\CS420Gasgnmt_5.docx
.\CS420Gasgnmt_8.docx
.\CS420G_asgnmt_1.docx
.\CS420G_asgnmt_2.docx
.\CS420G_asgnmt_4.docx
.\CS420G_asgnmt_6.docx
.\CS420G_asgnmt_7.docx
.\file.py
.\~$420G_asgnmt_2.docx
```

It will print all the files present in the directory.

The open function is used to open a file

file = open(file_name [, access_mode])

There are multiple types of Access_modes here like r, rb, w, wb, r+, w+

r → Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.

rb → Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.

w → Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, create a new file for writing.

wb → Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, create a new file for writing.

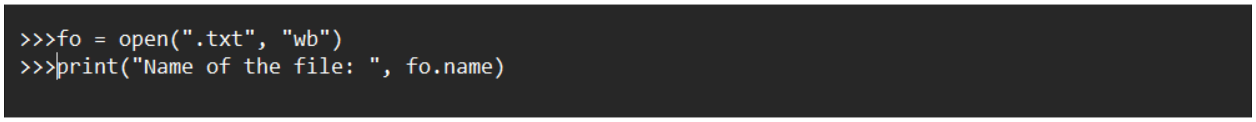
r+ → Opens a file for both reading and writing. The file pointer is placed at the beginning of the file.

w+ → Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, create a new file for reading and writing.

file.closed

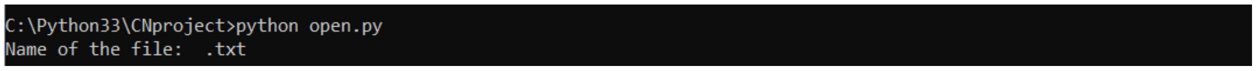
Returns true if the file is closed, false otherwise

To open a file, we need to save the code in **open.py** and then run it in the command prompt.



```
>>>fo = open(".txt", "wb")
>>>print("Name of the file: ", fo.name)
```

The output is:



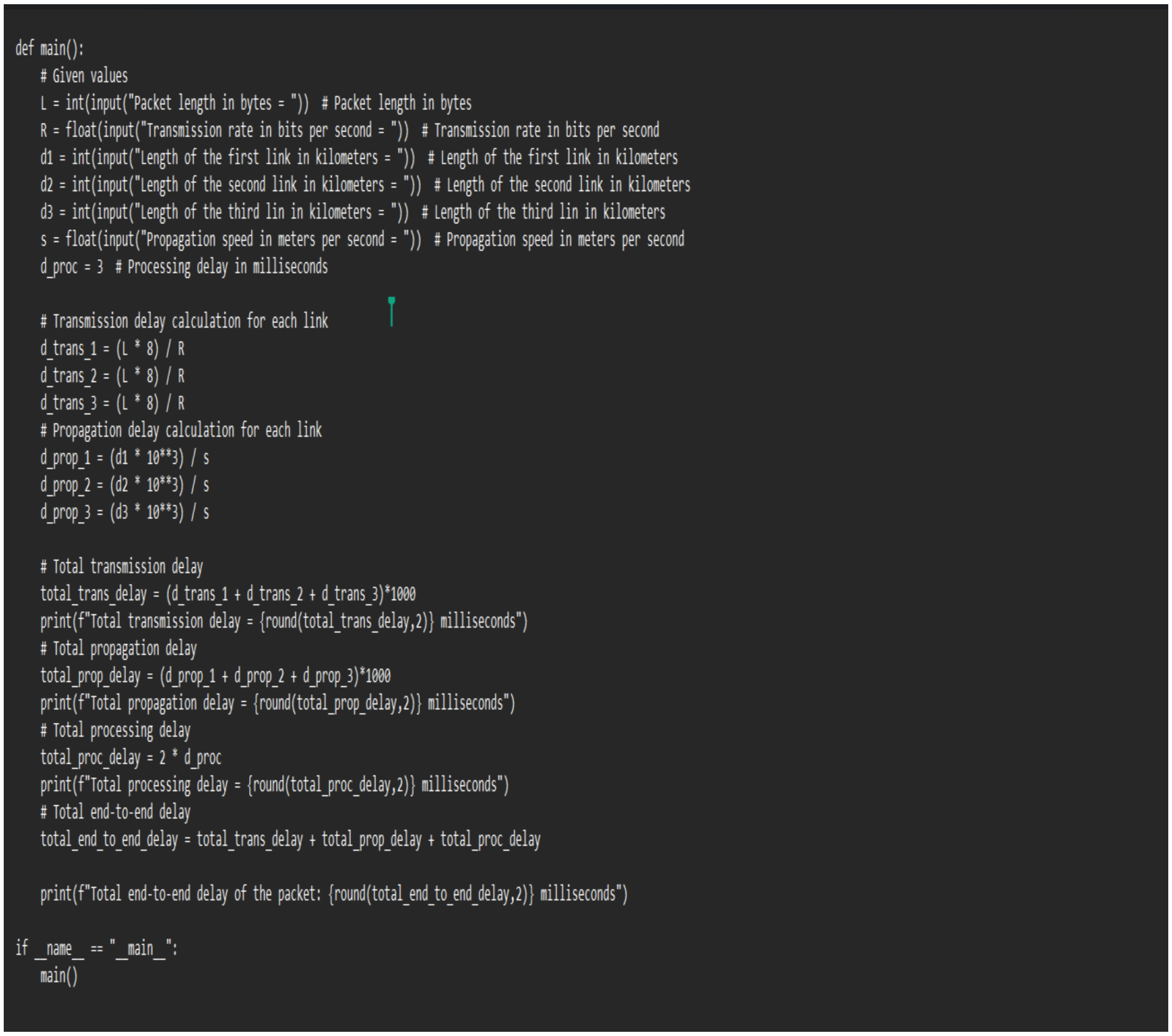
```
C:\Python33\CNproject>python open.py
Name of the file:  .txt
```

Milestone 4 – Assignment

write a program that (1) asks for or (2) accepts values to execute the dend-end formula, as you know from assignment 2.

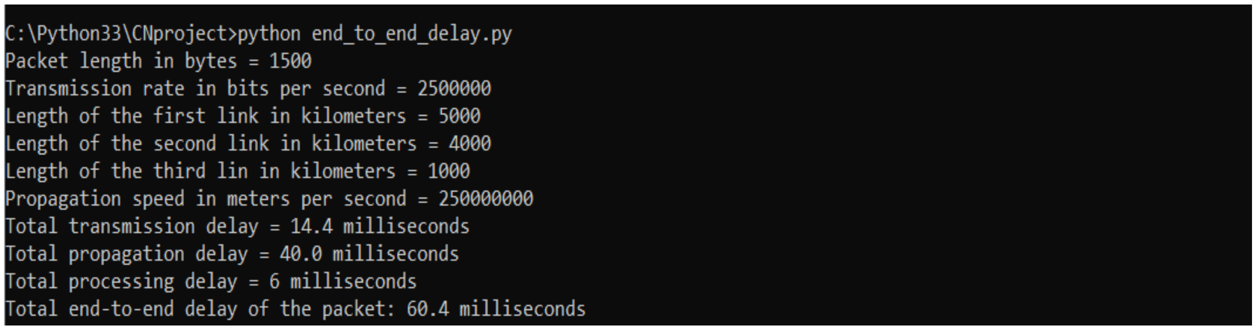
Solution:

1. we saved the program in a filename **end_to_end_delay.py** which we will run in the console**.**
2. In the command prompt, we have to compile the code using the command:
   python end_to_end_delay.py



```python
def main():
    # Given values
    L = int(input("Packet length in bytes = "))  # Packet length in bytes
    R = float(input("Transmission rate in bits per second = "))  # Transmission rate in bits per second
    d1 = int(input("Length of the first link in kilometers = "))  # Length of the first link in kilometers
    d2 = int(input("Length of the second link in kilometers = "))  # Length of the second link in kilometers
    d3 = int(input("Length of the third lin in kilometers = "))  # Length of the third lin in kilometers
    s = float(input("Propagation speed in meters per second = "))  # Propagation speed in meters per second
    d_proc = 3  # Processing delay in milliseconds

    # Transmission delay calculation for each link
    d_trans_1 = (L * 8) / R
    d_trans_2 = (L * 8) / R
    d_trans_3 = (L * 8) / R
    # Propagation delay calculation for each link
    d_prop_1 = (d1 * 10**3) / s
    d_prop_2 = (d2 * 10**3) / s
    d_prop_3 = (d3 * 10**3) / s

    # Total transmission delay
    total_trans_delay = (d_trans_1 + d_trans_2 + d_trans_3)*1000
    print(f"Total transmission delay = {round(total_trans_delay,2)} milliseconds")
    # Total propagation delay
    total_prop_delay = (d_prop_1 + d_prop_2 + d_prop_3)*1000
    print(f"Total propagation delay = {round(total_prop_delay,2)} milliseconds")
    # Total processing delay
    total_proc_delay = 2 * d_proc
    print(f"Total processing delay = {round(total_proc_delay,2)} milliseconds")
    # Total end-to-end delay
    total_end_to_end_delay = total_trans_delay + total_prop_delay + total_proc_delay

    print(f"Total end-to-end delay of the packet: {round(total_end_to_end_delay,2)} milliseconds")

if __name__ == "__main__":
    main()
```

Here is the output results screenshot when I run this code in the command prompt.



```
C:\Python33\CNproject>python end_to_end_delay.py
Packet length in bytes = 1500
Transmission rate in bits per second = 2500000
Length of the first link in kilometers = 5000
Length of the second link in kilometers = 4000
Length of the third lin in kilometers = 1000
Propagation speed in meters per second = 250000000
Total transmission delay = 14.4 milliseconds
Total propagation delay = 40.0 milliseconds
Total processing delay = 6 milliseconds
Total end-to-end delay of the packet: 60.4 milliseconds
```