Python Learner / How-to Guide By

Farah Al Yasari

Introduction

I have been wanting to learn Python for the past couple years. It is a flexible language used for countless purposes. I am particularly interested in learning more Python for my capstone project. Finally, many students in my classes already know Python, so I think it is necessary for me to keep my skills up to date and stay competitive with my peers.

Learning Approach

I learn by taking notes, then doing programming examples. My approach to learning Python is to study the language from Online sources. My goal is to have an overview of built-in functions, data-structures, and learning the language semantics. This manual will serve as my guide, in which I will keep my study notes and programming examples. There will be examples scattered throughout the manual, written in either plain text or shown inside screenshots. I will wrap up with a reflection on the important concepts.

Installation process

1. Download Python: https://www.python.org/
2. Download Atom: https://atom.io/
3. Install the following package for Atom: platformio-ide-terminal
    - This package is used to run the window terminal inside the Atom Editor
    - Next time you open the atom editor, you will see a small "+" at the bottom left. Click the "+" to open the terminal.

⇒ To check that your installation is correct, try to run a python file with the following command: `python file.py` or `py file.py` from now and forward command lines will be written in blue for readability.

Python Introduction

⇒ What is Python?
- Popular programming language
- Started in 1991
- Used in Data Science, Software Development, server-side development on the web, mathematics, and general scripts purposes.
- Fun fact: second most popular statistical language in addition to the R language

⇒ What can Python do?
- Server-side scripts
- Workflows for software
- Connect database systems
- Read and modify files

    - Handle big data
    - Perform complex math operations
    - Rapid prototyping
- ⇒ Why Python?
    - Works on different platforms (Windows, Mac, Linux, Raspberry Pi, Arduino, etc.). ○ Simple syntax
    - Less lines with more meanings
        - Allows developers to write less lines of code than some other programming language
    - Runs on an interpreter system
        - Code can be executed as soon as it is written
        - Prototyping can be very quick
    - Cool thing: Python can be treated in a procedural way, object-oriented way, or functional way! Here is a review of the three programming approaches:

Functional	Object-Oriented	Procedural
Treats computation as the evaluation of mathematical functions	Based on the concept of "objects" which may contain data. Ex: fields in a structure using methods.	Derived from structured programming, based on the concept of "procedure call" aka routines, subroutines, or functions
Avoids changing the state and data	Comfortably performs operations on fields if desired	Linear computational steps

Python Syntax

- ⇒ Quick and important table before we jump into coding:

	Python	Other languages
Main design	Readability with mathematical influence	Various
Complete a command	New line	Semicolon or Parenthesis
Scope	Indentation using whitespace	Curly-brackets

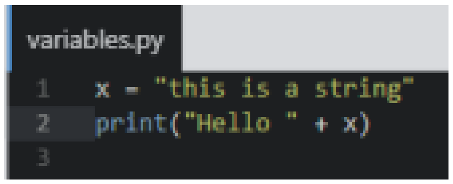
Variable declaration	No command to declare a variable	Usually it's data type followed by variable name then an initial value

Variables in Python

⇒ Overview
- ○ A variable is generated the moment it is assigned a value for the first time ○ A variable does not need to be declared with any type
- ○ A variable can change type after it has been set

⇒ Rules for Variable Names
- ○ Must start with a letter or underscore character
- ○ Cannot start with a number
- ○ Can only contain alpha-numeric characters and underscores (A-z, 0-9, _)
  - ▪ Hence, no special characters can be used in variable names
- ○ Case sensitive

⇒ + character
- ○ Combine both text and a variable in a print statement
  - ▪ Note: Like in other languages, you cannot combine an int and a string using + oeprator



variables.py

```
x = "this is a string"
print("Hello " + x)
```

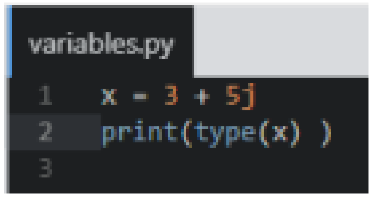
- ○ Add a variable to another variable for math operations
- ○ concatenate a variable to another variable for the case of a string

Python Numbers

⇒ Three numeric types in Python
- ○ Int
  - ▪ Whole number, positive number, negative
  - ▪ Without decimals
  - ▪ Unlimited length
- ○ Float
  - ▪ Positive or negative
  - ▪ Containing one or more decimals
  - ▪ Can be a scientific number with an "e" to indicate the power of 10
- ○ Complex
  - ▪ Written with "j" as the imaginary part

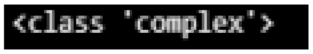
⇒ type () function

- verify the type of any object in python
- Example:



```
variables.py
x = 3 + 5j
print(type(x) )
```

And the output is:



```
<class 'complex'>
```

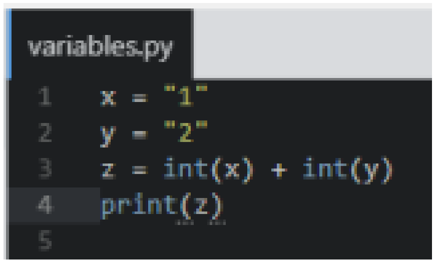
Specify a Variable Type

⇒ Specifying a type on to a variable is done with casting

⇒ Casing is done using constructor functions

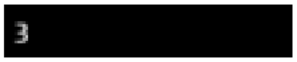
- Int() can construct:
  - An integer number from an integer literal
  - An integer number from a float literal (by applying a ceiling on the previous whole number)
  - An integer number from a string literal (in the case that the string provided represents a whole number)
- Float() can construct
  - A float number from an integer literal
  - A float number from a float literal
  - A float number from a string literal (providing the string represents a float or an integer)
- str() can construct
  - A string from a wide variety of data types
  - A string from a string literal
  - A string from an integer literal
  - A string from a float literal

⇒ Example of casting



```
variables.py
x = "1"
y = "2"
z = int(x) + int(y)
print(z)
```

the output is



```
3
```

String Literals

- ⇒ Surrounded by either single or double quotation marks
    - ‘hello’ is the same as “hello”
- ⇒ Strings in python are arrays of bytes representing Unicode characters
- ⇒ Python does not have a character data type
- ⇒ A single character is simply a string with a length of 1
- ⇒ Square brackets can be used to access elements of the string
- ⇒ Get the character at position n in string a
    - Example:

      a = “sample word”
      a[0] would return s
- ⇒ Get a substring, meaning
    - Gives the characters from position n to position m and m is not included ○ Example:

      b = “Hello, World!”
      b[2:5] would return llo
- ⇒ strip() function
    - removes any whitespace from the or the end or both
    - Example:

      a = “ this begins with a white space”
      print(a.string())
      returns “this begins with a white space”

- ⇒ replace() function
    - replaces a string with another string
    - Analogy in C/C++: replace a character by another character for a string
    - Example

      s = “Friday”
      print(s.replace(“a”, “@”))
      this would print “Frid@y”
- ⇒ Useful functions for string operations

Functions on strings s	Purpose
s.strip()	Removes whitespace from the beginning of the end or both
len(s)	Returns length of a string
s.lower()	Returns the string in lower case
s.upper()	Returns the string in upper case
s.replace(“e1”, “e2”)	In the string s, replace e1 with e2

s.split("separator")	Splits the string into substrings if it finds instance of the separator. Common separators: "," "-"

- ⇒ Command-line String Input
  - o input() method prompts the user to give an input
  - o Example

```
print("Enter your name:")
x = input()
print("Hello," + x)
```

Python Operators

- ⇒ Operators are used to perform operations on variables and values
- ⇒ Group of python operators
  - o Arithmetic
    - ▪ Numeric values to perform common mathematical operations

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation
//	Floor division: divides then cuts off the numbers after period by performing floor operation

  - o Assignment
    - ▪ Assignment operators are used to assign values to variables
    - ▪ Let # represent a dummy assignment operator,
      Var #= value is equivalent to Var = Var # value

        - Assignment can be done on Arithmetic operation and the following: &, |, ^, >> , and <<
    - Comparison
        - Used to compare two values

Operator	Name
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

    - Logical
        - Used to combine conditional statements

Operator	Description
and	Returns True if both statements are true
or	Returns True if one of the statements is true
not	Reverse the result, returns False if the result is true

    - Identity
        - To compare the objects are the same objects, with the same memory location.
        - This is different from comparing equality

Operator	Description
is	Returns true if both variables are the same objects
Is not	Returns true if both variables are not the same object

- Membership
  - Used to test if a sequence is presented in an object

Operator	Description
in	Returns True if a sequence with the specified value is present in the object
not in	Returns True if a sequence with the specified value is not present in the object

- Bitwise

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of the two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Python Lists

⇒ Four collection data types

1. Lists
   - Ordered, changeable, and allows duplicate members.
2. Tuple
   - Ordered, unchangeable, and allows duplicate members
3. Set

        ▪ Unordered, unchangeable, and disallows duplicate members

4. Dictionary
    ▪ Unordered, changeable, and disallows duplicate members. And it is indexed!

⇒ Lists

- Generate a list using square brackets
- Access item by referring to the index number
- Change item value by referring to the index number
- Example

```
thislist = [ “red” , “yellow”, “green” ]
print(thislist)
print(thislist[1])
thislist[2] = “white”
print(thislist[2])
```

- Loop through a list using a for loop
- Example

```
thislist = [ “red” , “yellow”, “green” ]
for x in thislist:
        print(x)
```

- Check if item exists in a list using the “in” keywork
- Useful methods and their purpose

Methods on list L	Description
Len(L)	Returns how many items are in the list
L.append(“item”)	Append an item to the end of the list
L.insert(index, “item”)	Add an item at a specific index in list L
L.remove(“item”)	Removes the specified item
L.pop(index)	If index is specified, it will remove the specified index. Otherwise, it removes the last item from list L.
del L[index]	Removes the specified index from the list
L.clear()	Empties the list

Other methods:

Method	Description
copy()	Returns a copy of the list

count()	Returns number of elements with the specified value
extend()	Add the element of a list (or an iterable) to the end of the current list
index()	Returns the index of the first element with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

- The list() constructor
  - ▪ The list() constructor can be used to make a list
  - ▪ Example:

  thislist = list((“red”, “yellow”, “green”))

  # note the double round-brackets

⇒ Tuple Length

- To generate a tuple, use round brackets
- Access tuple items by referring to the index number, inside square brackets • Tuples are unchangeable
  - ▪ if you try to change an item nothing will happens
  - ▪ if you try to add an item, you will raise an error
  - ▪ You cannot delete an item
  - ▪ You can delete the entire tuple using the del keyword
- The tuple() constructor
  - ▪ The tuple() constructor can be used to make a tuple
  - ▪ Example:

    thistuple = tuple((“red”, “green”, “blue”))

    #note the double round-brackets

- Built-in methods used on tuples

Method	Description
count()	Returns the number of times the specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

⇒ Sets

- To generate a set, use curly brackets
- Unordered and unindexed
- You cannot access items in a set by referring to an index, since sets are unordered • You can still loop through the set with a for loop
- You cannot change the items
- You can add new items
- Built-in methods you can use on sets

Method	Description
add()	Add an element to the set
update()	Add multiple elements to the set
copy()	Returns a copy of the set
difference()	Takes two or more sets and returns a new set which contains the difference of the two sets
difference_update()	Removes the items in this set that also in the other set
discard()	Removes a specified item
intersection()	Takes two or more sets and returns a new set with the intersection of the given sets
intersection_update()	Removes the items in this set that are not in the other set
isdisjoint()	Returns whether two sets have an intersection or not
issubset()	Returns whether this set has the other set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_differenence()	Returns a set with the symmetric difference of two sets
symmetric_difference_ update()	Inserts the symmetric difference from this set to another set

union()	Returns a set containing the union of two or more sets
update()	Update the set with the union of this set

⇒ Dictionary

- To generate a dictionary, use curly braces and separate the fields and the values of the fields with :
- Example

```
my_dictionary = {

        “cat_name” : “Tom”

        “age” : “1”

        “year” : 2007

}
```

- Items of the dictionary can be accessed by referring to the key name, inside square brackets.
- Example
  ```
  X = my_dictionary[“cat_name”]
  ```
- The get() method can also be used to get the same results
- Example
  ```
  X = my_dictionary.get(“cat_name”)
  ```
- A value inside the dictionary can be changed by referring to its key name
- Example
  ```
  #change the age to 2
  my_dictionary[“age”] = 2
  ```
- Looping through a dictionary
  - Get the keys of the dictionary, one by one
    - Example
      ```
      for x in my_dictionary:
          print(x)
      ```
    - Example
      ```
      For x in my_dictionary:
          Print(my_dictionary[x]
      ```
  - Get the values of the dictionary, one by one
    - Example
      ```
      For x in my_dictionary.values():
      ```

print(x)

- Use items() function to loop though both keys and values
  - For x, y in my_dictionary.items():
    Print(x,y)
- The in keyword can be used to check if a specified key is in a dictionary
- The len() method can be used to determine how many items are in the dictionary
- How to add an item to an existing dictionary?
  - Use a new index key and assign a value to it
  - Example
    my_dictionary[“color”] = “orange”
- Ways to remove items from a dictionary:

pop()	Removes the item with the specified key name
popitem()	Removes the last inserted key-value pair
clear()	Removes all fields in the dictionary

Conditional statements & Loops

⇒ If-else
- The conditions end with “:”
- Words for Operators: “and”, “or”
- The keyword “elif” refers to the else-if behavior
- We can have one line if else statement, with multiple conditions
  - Example
    Print(“a”) if a > b else print(“=”) if a == b else print(“b”)
    Is similar to
    If(a > b) print(“a”)
    Else if(a==b) print (“=”)
    Else print(“b”)

⇒ While Loop
- Continue statement
  - Stops the current iteration, and continues to the next iteration before executing the reminder lines of code
- The while() end with “:”

⇒ For Loop
- Works similarly as an iterator, that’s found in OOP languages
- Strings are considered iterable objects because they contain a sequence of characters
- Continue statement
  - Stops the current iteration, and continues to the next iteration before

executing the reminder lines of code

- "Else" keyword after a "For" keyword
    - Specifies a block of code to be executes when the For Loop is done
- range() function
    - Loops through a set of code a specified number of times
        - Default behavior:
            - Increments by 1
            - Starts a t 0
            - Ends at the (specified value – 1) iteration
        - Can be customized
            - Increments by a customizable value
            - Starts at a given initial value
            - Ends at the (final value – 1) iteration
    - Example of default behavior

```
for x in range(10)
        y = x % 2
```

    - Example of customizable behavior

```
for x in range(4, 20)
        y = x % 2
```

    - Example of another customizable behavior

```
for x in range(2, 20, 2)
        y = x + z
```

- 3 maximum parameters: range(initial value, final value, increment value)

Functions in Python

⇒ Start a function using the "def" keyword

⇒ Call a function using the function name followed by parenthesis

⇒ Parameters in python functions work just like in other languages

- We can give default parameters to functions
    - If the function expects a parameter, but no parameter is used, then the default value will be used.
    - Example of setting a default parameter

```
def my_fun (Name = "Tom"):
        print("Nice to meet you" + Name)
```

⇒ Use "return" to set the return value

Lambda Functions in Python

⇒ An anonymous function

- A function definition that is not bound to an identifier
- Used for constructing results of other functions
- In functional programming languages, anonymous functions server as the function type like how literals serve data types.

- ⇒ Can take any number of arguments
- ⇒ Can have only one expression
- ⇒ Syntax
  - o function name = lambda argument(s): expression
- ⇒ Example

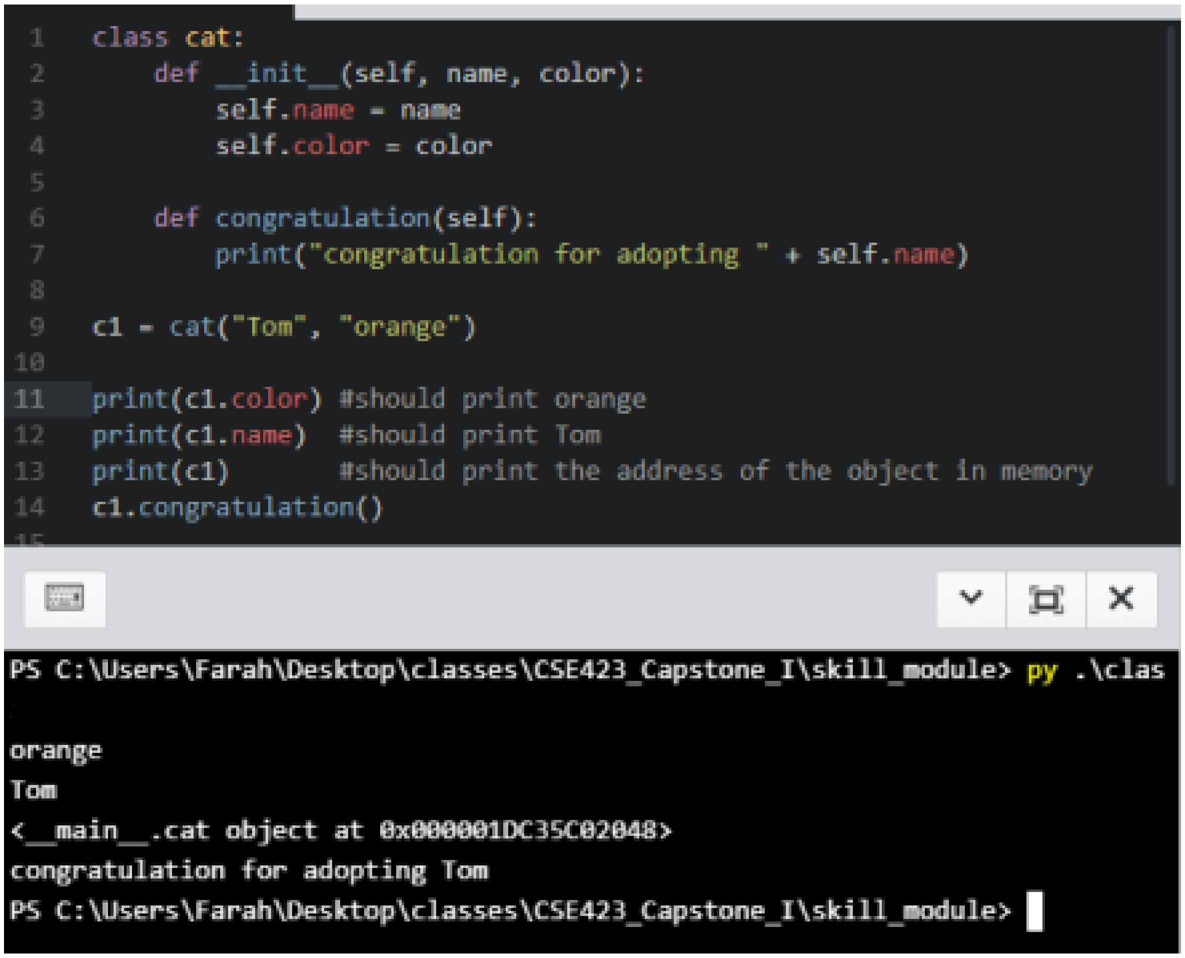
```
x = lambda a, b = a + b
print(x(2,1))
#expected to return 3
#notice that the order of arguments in the function call corresponds to the order
of  #arguments in the function definition
```

- ⇒ Lambda functions can be used in the body of other named functions
  - o Example

```
def powerful_func(n)
        x = lambda a : a * n
        return x
#this function can be used to as a doubler, tripler, etc
```

Classes and Objects in Python

- ⇒ Use the keyword “class” to start a class
- ⇒ Use a dot “.” to access a property of the class
- ⇒ The __init__() function
  - o Called automatically every time the class is used, or a new object is made o Assigns values to the object properties
- ⇒ The self-Parameter
  - o Reference to the class itself
  - o Used to access variables inside the class
  - o Does not have to be named self, it can be named anything
  - o It must be the first parameter of any function in the class
- ⇒ Example



```python
class cat:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def congratulation(self):
        print("congratulation for adopting " + self.name)

c1 = cat("Tom", "orange")

print(c1.color) #should print orange
print(c1.name)  #should print Tom
print(c1)       #should print the address of the object in memory
c1.congratulation()
```

```
PS C:\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module> py .\clas

orange
Tom
<__main__.cat object at 0x000001DC35C02048>
congratulation for adopting Tom
PS C:\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module>
```

Python Iterators

⇒ An iterator object which implements the iterator protocol
  - Consists of the methods __iter__() and __next__()

⇒ Used to traverse through a countable number of values

⇒ Iterators vs Iterable
  - List, tuples, dictionaries, sets, and arrays are examples of iterable objects ○ Iterable objects can have an iterator to traverse through their elements ○ Iterable objects have an iter() method to give them an iterator
  - Examples
    #pass the tuple to the iter() method which returns an iterator
    mytuple = (“green”, “yellow”, “orange”)
    myit = iter(mytuple)

⇒ Iterator methods can be implemented for classes

Python Modules

⇒ Must have the file extension .py

- ⇒ A module in python is like a library in other languages
- ⇒ To use a module, use the "import" keyword followed by the module name
- ⇒ To access an element inside the module, use the "." operator
- ⇒ To easily rename a module in the future, you can make an alias for a module and use the alias name rather than the module name
  - o Use the "as" keyword to give an alias to a module
  - o Example:

    Import communication_protocol as com
- ⇒ Python has built-in modules and open source modules available in the community
  - o urx python library is the open source library we are using in the capstone class to control the UR robot arm
- ⇒ To import only one element from the module you can use the "from" and "import" keywords
  - o Example

    From communication_protocol import TCP
  - o When you import only one element from the module, do not use the"." operator – simply refer to the element by it's name
- ⇒ The dir() Function
  - o A built-in function
  - o Lists all the functions names (or variables) in a module
  - o The argument is the specified module
  - o The returned value is a comma separated lists of variables and functions inside the module, where each variable or function is inside single quotations

Python PIP – Package Manager

- ⇒ Included by default for Python version 3.4 or later
- ⇒ To download a package, ask PIP to install the package
  - o Navigate to your command line
  - o Navigate to Python's script directory
  - o Type "install" followed by space and the name of the module/package to install o A package is ready to use after it is installed
  - o To uninstall a package, use the "uninstall" command
- ⇒ List command: Shows all the packages installed on your system

Exception Handling

- ⇒ Like Exception error handling in java
- ⇒ Format of try and except

```
try:

        #run these blocks of code

except:
```

```
                #run these blocks of code if there is an error in running the try block of
code ⇒ Format of try, except, and else
    try:
                #run these blocks of code
    except:
                #run these blocks of code if there is an error in running the try block of
    code Else:
                #run this code if not exception happened
```

⇒ Format of try, except, and finally

```
    try:
                #run these blocks of code
    except:
                #run these blocks of code if there is an error in running the try block of
    code Else:
                #run these blocks of code regardless if try block gives an error or not
```

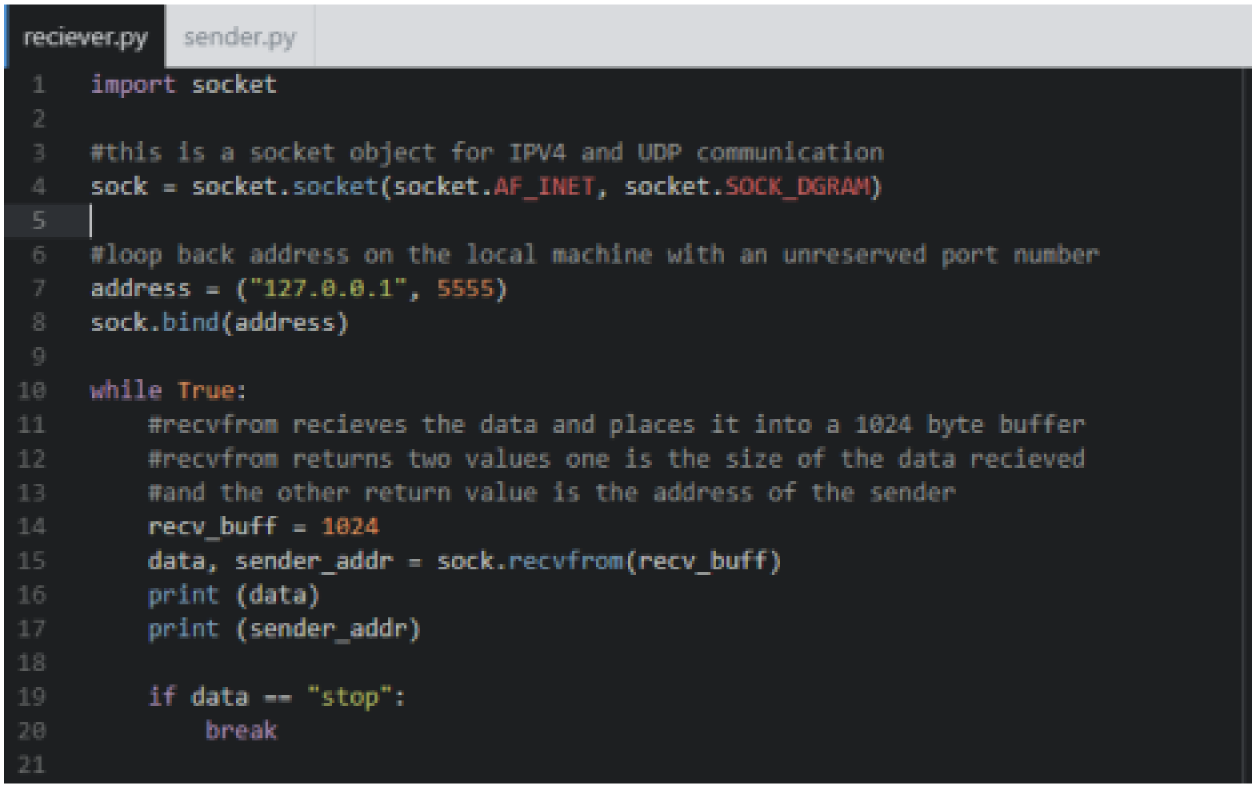
⇒ Example: Writing to a file without write permission granted.

```
    try:
                f = open (“sample.text”)
                f.write(“hello”)
    except
                print(“you don’t have permission to write”)
    finally:
                f.close
```

Simple UDP sender and receiver

⇒ Familiarize yourself with the socket API in python and write a UDP sender and receiver ○ I learned that in python the recvfrom function can have two return values. This is unique, and it is the first time I use a function that can return several values. In C/C++ it is not possible to have more than one return value, however, one could allocate a slap of memory to store a return value. Then, a pointer can be used to access the returned values.

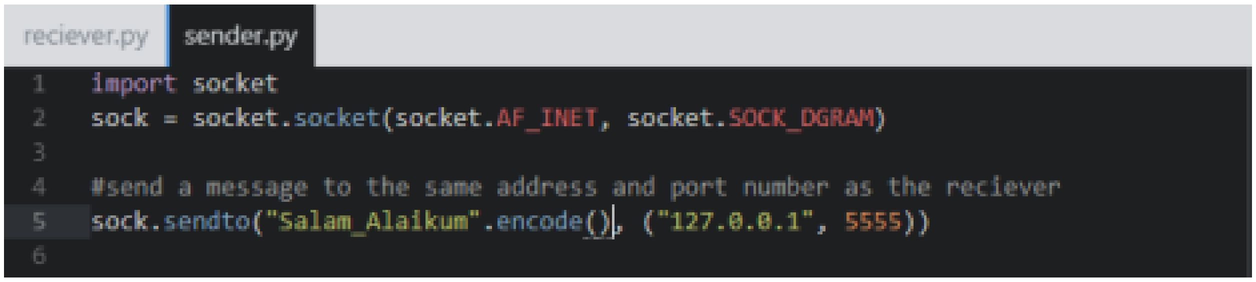
- ○ I learned that in python there is a “print” and a “print()”. They are the same, but one is for an old version and one for version 3 and beyond.
- ○ Here is a screenshot of the receiver code:



```python
import socket

#this is a socket object for IPV4 and UDP communication
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

#loop back address on the local machine with an unreserved port number
address = ("127.0.0.1", 5555)
sock.bind(address)

while True:
    #recvfrom recieves the data and places it into a 1024 byte buffer
    #recvfrom returns two values one is the size of the data recieved
    #and the other return value is the address of the sender
    recv_buff = 1024
    data, sender_addr = sock.recvfrom(recv_buff)
    print (data)
    print (sender_addr)

    if data == "stop":
        break

```

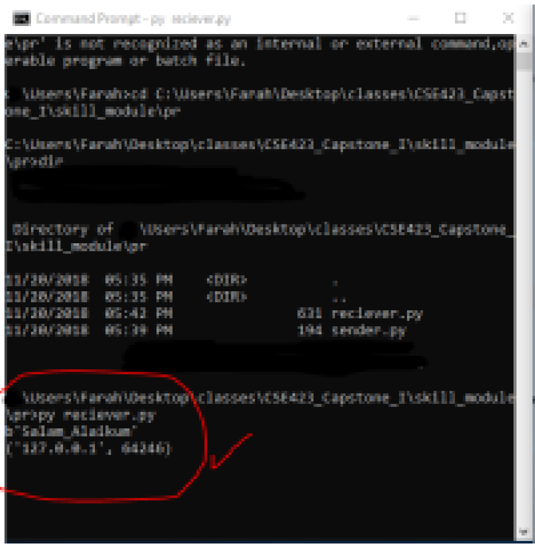
⇒ Sender

- I learned about the encode() method which is used to encode a string into a byte like object.
- Here is the sender code:

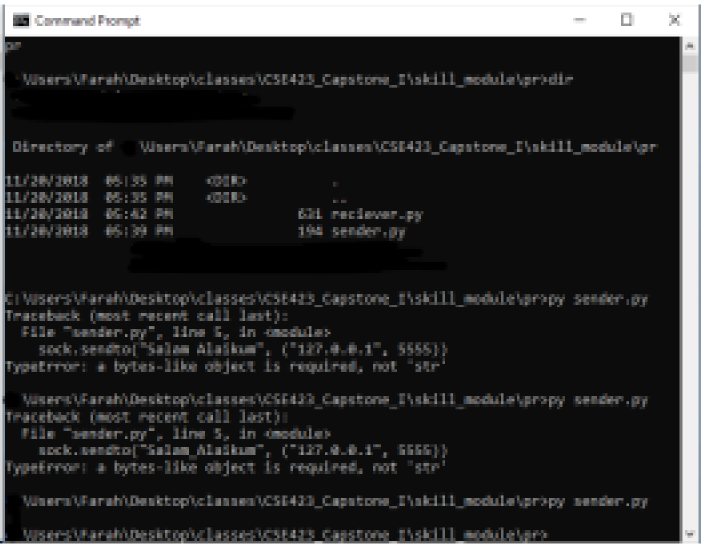


```python
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

#send a message to the same address and port number as the reciever
sock.sendto("Salam_Alaikum".encode(), ("127.0.0.1", 5555))

```

⇒ The sender and the receiver can successfully communicate together!



```
Command Prompt - py reciever.py
e\pr' is not recognized as an internal or external command,op
erable program or batch file.

[redacted]\Users\Farah>cd C:\Users\Farah\Desktop\classes\CSE423_Capst
one_I\skill_module\pr

C:\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module
\pr>dir
[redacted]

 Directory of [redacted]\Users\Farah\Desktop\classes\CSE423_Capstone_
I\skill_module\pr

11/20/2018  05:35 PM    <DIR>          .
11/20/2018  05:35 PM    <DIR>          ..
11/20/2018  05:42 PM               631 reciever.py
11/20/2018  05:39 PM               194 sender.py
               [redacted]

[redacted]\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module
\pr>py reciever.py
b'Salam_Alaikum'
('127.0.0.1', 64246)
```

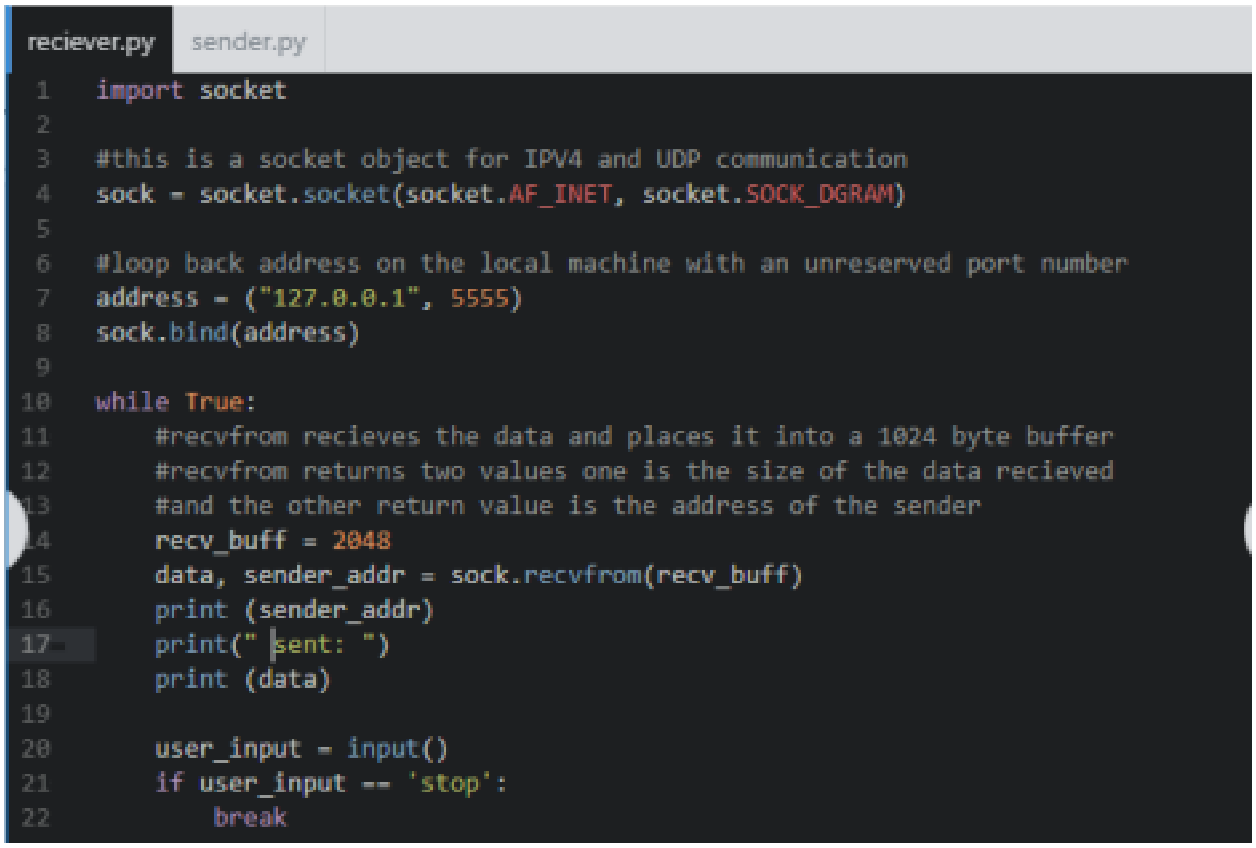


```
Command Prompt
pr

[redacted]\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module\pr>dir
[redacted]

 Directory of [redacted]\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module\pr

11/20/2018  05:35 PM    <DIR>          .
11/20/2018  05:35 PM    <DIR>          ..
11/20/2018  05:42 PM               631 reciever.py
11/20/2018  05:39 PM               194 sender.py
               [redacted]

C:\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module\pr>py sender.py
Traceback (most recent call last):
  File "sender.py", line 5, in <module>
    sock.sendto("Salam Alaikum", ("127.0.0.1", 5555))
TypeError: a bytes-like object is required, not 'str'

[redacted]\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module\pr>py sender.py
Traceback (most recent call last):
  File "sender.py", line 5, in <module>
    sock.sendto("Salam_Alaikum", ("127.0.0.1", 5555))
TypeError: a bytes-like object is required, not 'str'

[redacted]\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module\pr>py sender.py

[redacted]\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module\pr>
```

Improved UDP sender and receiver

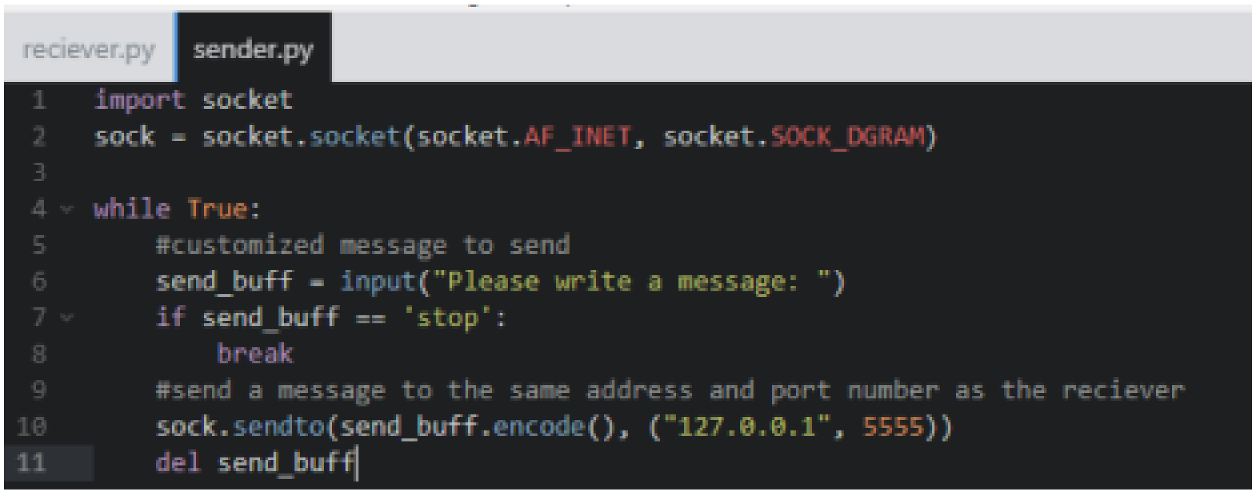
⇒ In the previous exercise, the message to send is hardcoded. Improvement is made to prompt the user to send a customized message, if either the sender or the receiver type "stop" the communication will stop.

⇒ New receiver



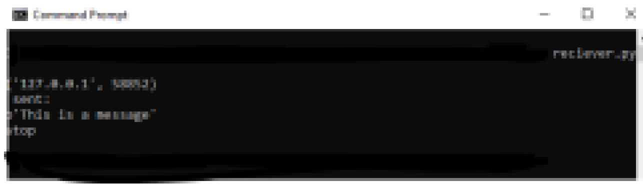
```python
import socket

#this is a socket object for IPV4 and UDP communication
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

#loop back address on the local machine with an unreserved port number
address = ("127.0.0.1", 5555)
sock.bind(address)

while True:
    #recvfrom recieves the data and places it into a 1024 byte buffer
    #recvfrom returns two values one is the size of the data recieved
    #and the other return value is the address of the sender
    recv_buff = 2048
    data, sender_addr = sock.recvfrom(recv_buff)
    print (sender_addr)
    print(" sent: ")
    print (data)

    user_input = input()
    if user_input == 'stop':
        break
```

⇒ New Sender

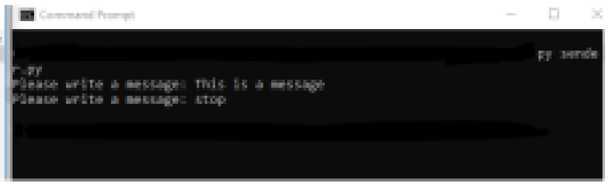


```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

while True:
    #customized message to send
    send_buff = input("Please write a message: ")
    if send_buff == 'stop':
        break
    #send a message to the same address and port number as the reciever
    sock.sendto(send_buff.encode(), ("127.0.0.1", 5555))
    del send_buff
```

⇒ Sender and receiver:



```
[redacted] reciever.py
('127.0.0.1', 58852)
sent:
b'This is a message'
stop
[redacted]
```



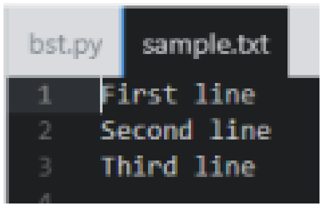
```
[redacted] py sende
r.py
Please write a message: This is a message
Please write a message: stop
[redacted]
```

⇒ Comment

- I spent a very long time trying to figure out how to import the socket package. I assumed that I needed to download it separately using pip, so I searched online to download the python socket library, and I was close to going into a loop of downloading the headers inside the socket library. Then, I decided to pause and check if the socket package already exists as a standard library. Fortunately, it does, and I did not have to go through the hassle of installing it. I spent a long time on this because I kept getting an error when running my code that imported the socket package. I later learned that the error had nothing with the socket package, and I was accidently trying to run the file outside of its home directory.

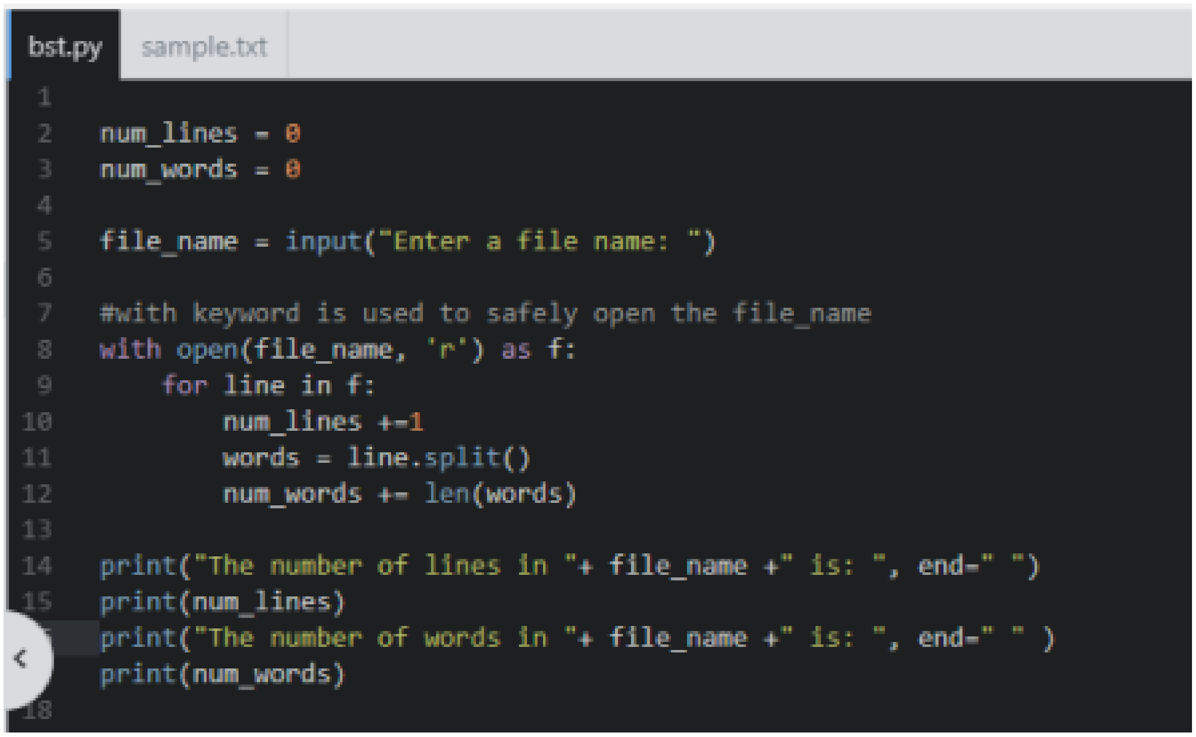
Count number of lines and words in a file

⇒ Test File



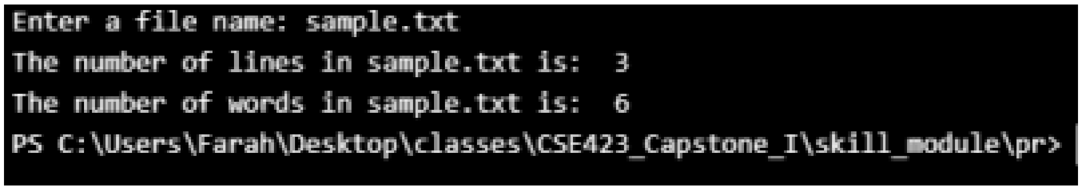
```
First line
Second line
Third line
```

⇒ Code that counts the number of words and lines in a given file



```python

num_lines = 0
num_words = 0

file_name = input("Enter a file name: ")

#with keyword is used to safely open the file_name
with open(file_name, 'r') as f:
    for line in f:
        num_lines +=1
        words = line.split()
        num_words += len(words)

print("The number of lines in "+ file_name +" is: ", end=" ")
print(num_lines)
print("The number of words in "+ file_name +" is: ", end=" " )
print(num_words)

```

⇒ Output



```
Enter a file name: sample.txt
The number of lines in sample.txt is:  3
The number of words in sample.txt is:  6
PS C:\Users\Farah\Desktop\classes\CSE423_Capstone_I\skill_module\pr>
```

⇒ Comment

- I learned that in order to print an integer and a string on the same line, the print method must receive an argument setting the end flag to “ “. This indicates that no newline should be added at the end. In python, a newline is added by default after every print. Writing this code in python was much more convenient than in C/C++ because in python it is possible to say, “for line in f”. In C/C++, a line must be recognized by a special end of line character. Notice that using the “with” keyboard allows the file to be closed automatically at the end.

Reflection

My goal for this skill module has been achieved. I feel very comfortable with the Python programming language semantics, built-in methods, and built-in data structures. I think that I best learn programming by studying the language, and taking notes, along with examples. I will attempt this approach again to learn other programming languages. I initially had some challenges setting up the environment to run Python. When I first started programming, I kept using tokens from other languages like parenthesis and semicolons. These Tokens are not valid in Python. I eventually learned to adjust my coding habits. In conclusion, I enjoyed studying

Python, and I hope someone else finds the guidance in this paper beneficial too.