PROGRAMMING WITH PYTHON

By Randal Root

Module 04

In this module, you learn about **creating scripts that work with collections of data in the form of Tuples and Lists**. We also take a closer look at some of the many functions you use working with Strings.

Collections (Sequences)

Programming languages allow you **to manage collections of data using a single variable** or constant. In Python, these are known as "**Sequences," but in other languages, they are called "Collections."**

You create a Sequence variable using a **different syntax based on the type of sequence you want.** Here is an example of **the ones you use most often**, note how **each uses different operators**.

```
strData = "123" # a string uses quotes
rngData = range(1,4) # a range uses a function
tplData = (1,2,3) # a tuple uses parentheses
lstData = [1,2,3] # a list uses brackets
```

Listing 1

You can **access individual values in the collection using a "Subscript."** A subscript **usually is a number** that represents the location of the value you want within the sequence. The **number almost always starts with zero** and is **known as an "Index."**

```
strData = "123"
print(strData[0]) # index zero prints value 1
print(strData[1]) # index one prints value 2
```

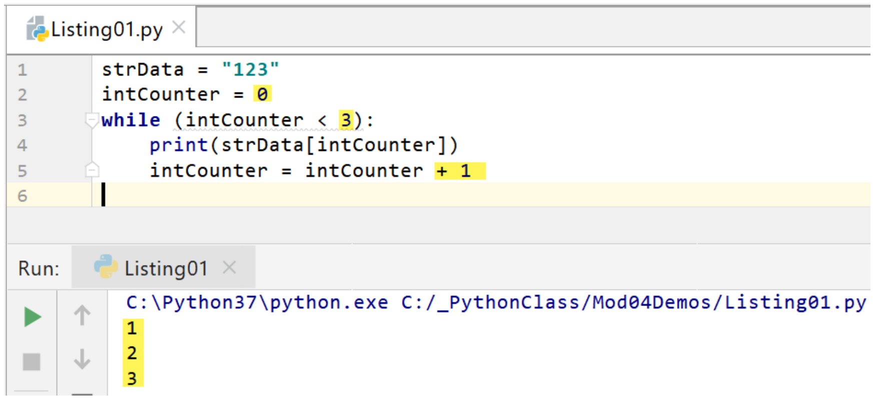
Listing 2

Looping through collections

You can **loop through all the values in a collection** using a loop. Here is an example using a While loop. Note that the **counter is used as an index**.

```
strData = "123"
intCounter = 0
while (intCounter < 3):
    print(strData[intCounter])
    intCounter = intCounter + 1
```

Listing 3


Listing01.py

```
strData = "123"
intCounter = 0
while (intCounter < 3):
    print(strData[intCounter])
    intCounter = intCounter + 1
```

Run: Listing01

```
C:\Python37\python.exe C:/_PythonClass/Mod04Demos/Listing01.py
1
2
3
```

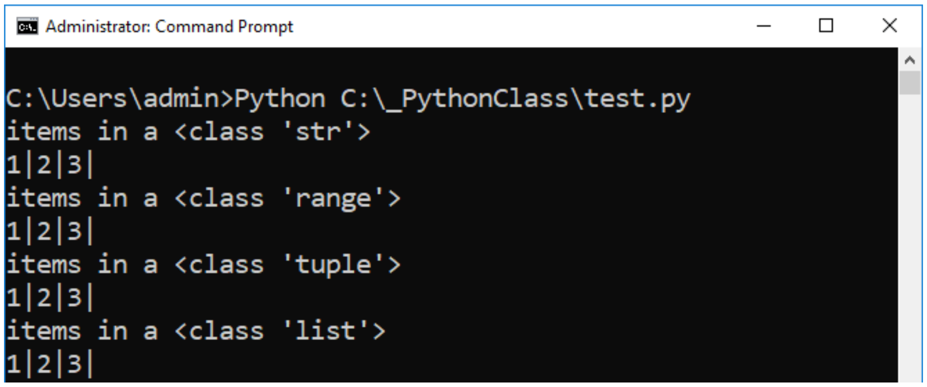
Figure 1. The results of Listing 3 in PyCharm

The for loop

Instead of using a While loop, you can simply use a "for loop" to loop through all items in the collection automatically. **For loops** do not need a counter variable since they **automatically stop** looping after they process the last value in the collection. Here is an example:

```
strData = "123"
rngData = range(1, 4)
tplData = (1, 2, 3)
lstData = [1, 2, 3]

print("items in a", type(strData))
for item in strData:
    print(item, end='|')

print("\n", "items in a", type(rngData))
for item in rngData:
    print(item, end='|')

print("\n", "items in a", type(tplData))
for item in tplData: print(item, end='|')

print("\n", "items in a", type(lstData))
for item in lstData: print(item, end='|')
```

Listing 4


Administrator: Command Prompt

```
C:\Users\admin>Python C:\_PythonClass\test.py
items in a <class 'str'>
1|2|3|
items in a <class 'range'>
1|2|3|
items in a <class 'tuple'>
1|2|3|
items in a <class 'list'>
1|2|3|
```

Figure 2. The results of Listing 4

Strings

Strings are a collection of individual characters. Since they are a collection, they are programmed to **act like other sequences**. However, they **also have additional methods** designed specifically for working with character data.

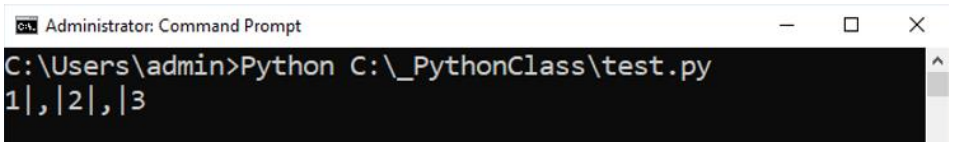
You use either the **single-quotes or double-quotes operator** to indicate which values you wish to be part of the String's collection. In **Python, the choice of single or double quotes does not matter, but in other languages it may**, so always test your code!

String indexes

You can **identify an individual character** in a string **with an index subscript.**

```
strData = '1,2,3'
print(strData[0], strData[1], strData[2], strData[3], strData[4],
sep='|')
```

Listing 5



```
C:\Users\admin>Python C:\_PythonClass\test.py
1|,|2|,|3
```

Figure 3. The results of Listing 5

NOTE: Remember, the numbering system is **zero**-based!

String Methods

There are a lot of methods in Python's String class, but there are only **a few you will use all the time. Here are some notes from the Python help document on ones I find useful**:

"

str.lower()
Return a copy of the string converted to lowercase.

str.upper()
Return a copy of the string converted to uppercase.

str.strip([chars])
Return a copy of the string with the leading and trailing characters removed.

str.replace(old, new[, count])
Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

str.isalpha()
Return true if all characters in the string are alphabetic and there is at least one character (in the string), false otherwise.

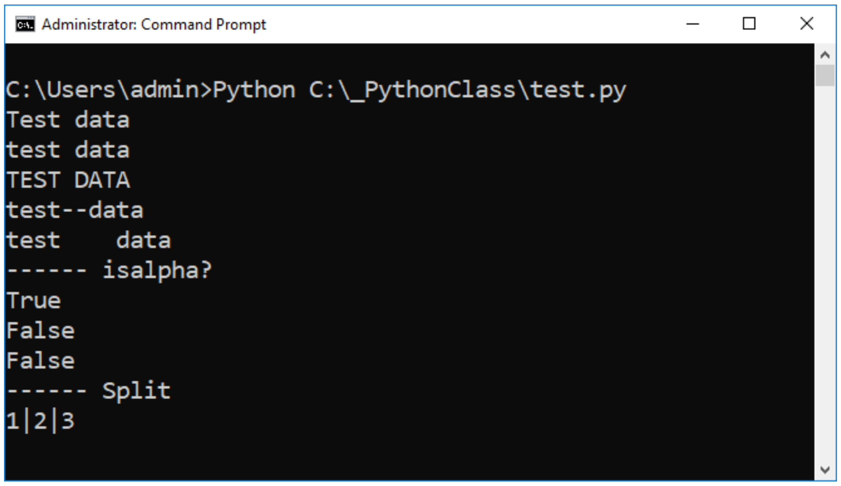
str.split([sep[, maxsplit]])
Return a list of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified, then there is no limit on the number of splits (all possible splits are made)."

(Python Help Files, www.python.org, 2019)

Here are **some examples** of how these functions are used:

```
print(strData.lower())
print(strData.upper())
print(strData.replace(" ", "--"))

strData = "  test    data   "
print(strData.strip())

print("------ isalpha?")
strData = "abc"
print(strData.isalpha())#true
strData = "123"
print(strData.isalpha())#false
strData = "abc123"
print(strData.isalpha())#false

print("------ Split ")
strData = '1,2,3'
lstData = strData.split(',')
print(lstData[0], lstData[1], lstData[2], sep='|')
```

Listing 6



```
C:\Users\admin>Python C:\_PythonClass\test.py
Test data
test data
TEST DATA
test--data
test    data
------ isalpha?
True
False
False
------ Split
1|2|3
```

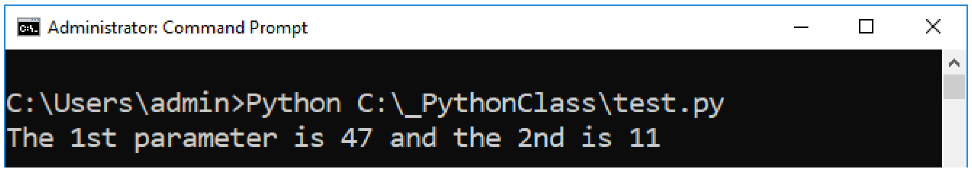
Figure 4. The results of Listing 6

Substitution Parameters

Python allows the use of **Substitution Parameters in a string by using the format() method**. These can be convenient if you to concatenate a lot of data into add to a single String, and most languages offer something similar. Here is an example using **positional parameters**:

```
print("The 1st parameter is {0} and the 2nd is {1}".format(47,11))
```

Listing 7


Administrator: Command Prompt

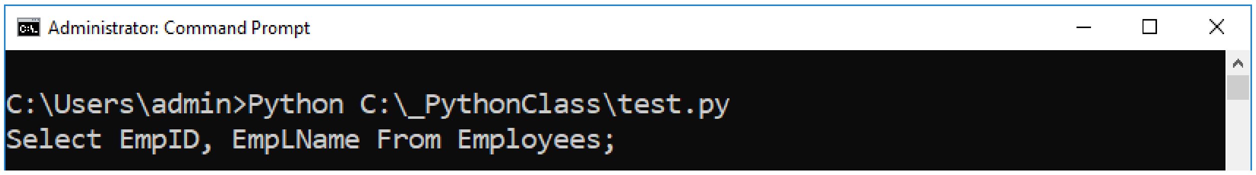
```
C:\Users\admin>Python C:\_PythonClass\test.py
The 1st parameter is 47 and the 2nd is 11
```

Figure 6. The results of Listing 7

Here is another, but this one uses **named parameters** instead of positional ones:

```
print("Select {Col1}, {Col2} From
{Table};".format(Col1="EmpID",Col2="EmpLName", Table="Employees"))
```

Listing 8


Administrator: Command Prompt

```
C:\Users\admin>Python C:\_PythonClass\test.py
Select EmpID, EmpLName From Employees;
```

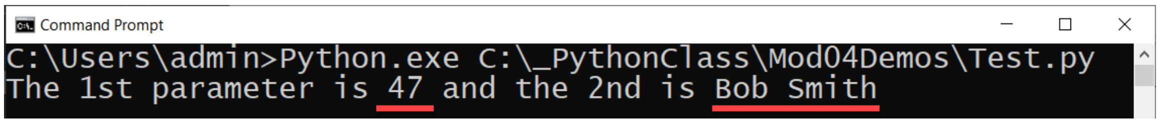
Figure x. The results of Listing 8

The %s and %d Operators

Many languages also allow the use of a "%s" operator to indicate a string parameter and a "%d" operator for a decimal parameter. These are usually used as part of a Print function, as shown in the following code:

```
print("The 1st parameter is %d and the 2nd is %s" % (47, "Bob Smith"))
```

Listing 9


Command Prompt

```
C:\Users\admin>Python.exe C:\_PythonClass\Mod04Demos\Test.py
The 1st parameter is 47 and the 2nd is Bob Smith
```

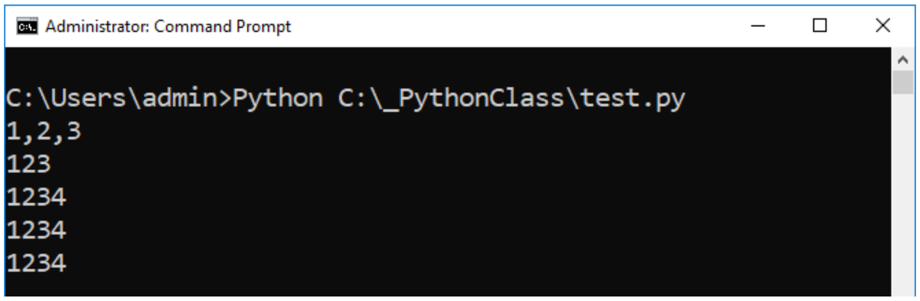
Figure x. The results of Listing 9

The Immutable String

Once strings are made, they **cannot change their values, <u>even if it looks like they can</u>**.

```
strData = '1,2,3'
print(strData)

strData = '123'
print(strData) #Actually a new string!

strData = '123' + '4'
print(strData) #Actually another new string!
```

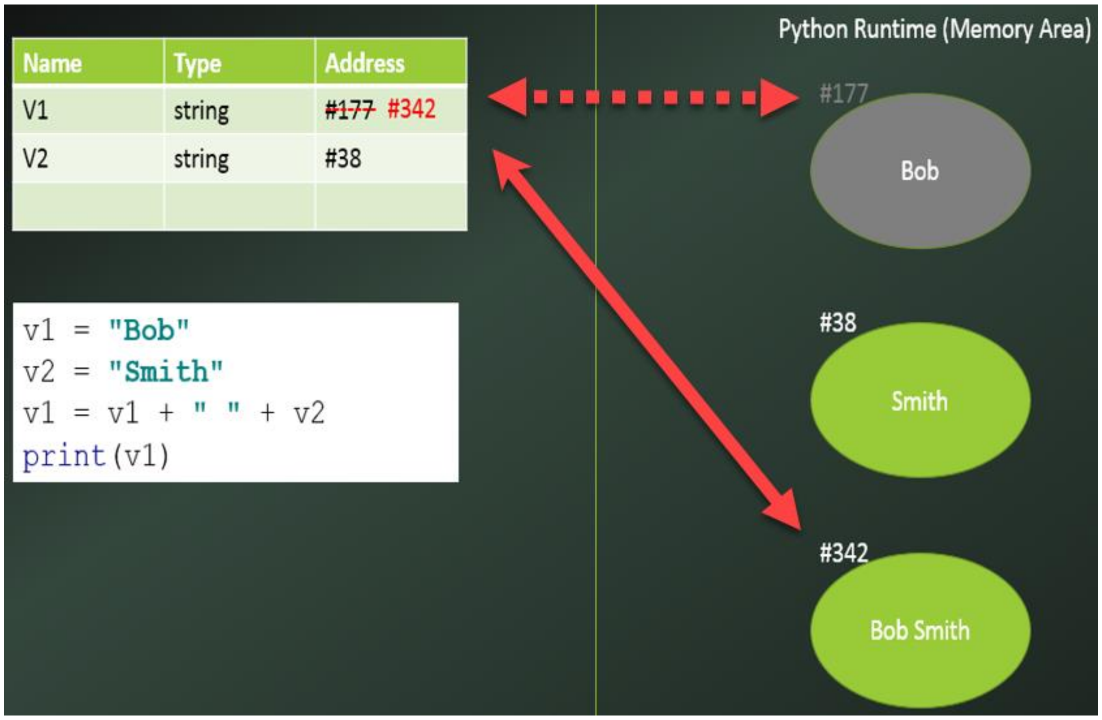
```
#strData[2] = 5 #You cannot assign a value after a string is made
print(strData)

#strData[5] = 5 #You cannot add more elements either
print(strData)
```
Listing 10



```
Administrator: Command Prompt

C:\Users\admin>Python C:\_PythonClass\test.py
1,2,3
123
1234
1234
1234
```

Figure x. The results of Listing 10

Figure x shows a way to visualize what is happening. Note how the **String variable V1** address is replaced by a **different address whenever the variable's value changes**. The previous value will still be in memory, but the variable will no longer point to it. **Eventually, the Python runtime will automatically remove it from memory.**



Name	Type	Address
V1	string	~~#177~~ #342
V2	string	#38

```
v1 = "Bob"
v2 = "Smith"
v1 = v1 + " " + v2
print(v1)
```

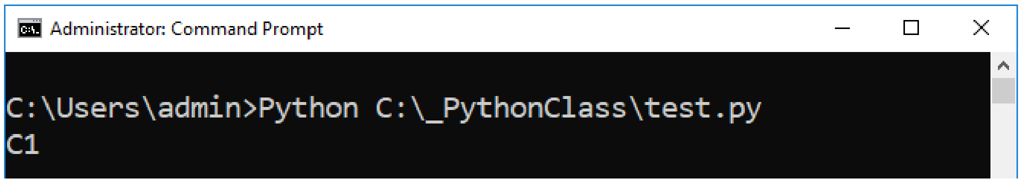
Figure x. A string variable's address changes when its value changes

Slicing strings

You can slice out portions of a string with the range operator. Note how only value (or "Element") 2 and 3 are returned. **The fourth value of the slice is not included in Python, but is included in other languages! Always double-check when dealing with a new language!**

```
strData = 'ABC123'
# Elements 012345
print(strData[2:4])
```

Listing 11



```
C:\Users\admin>Python C:\_PythonClass\test.py
C1
```

Figure x. The results of Listing 11

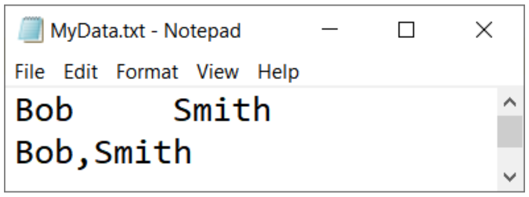
Writing Strings to a File

One of the most common tasks you perform with Strings is writing data to a file. When you do so, you **often separate the string by an invisible "tab" character or a visible comma character**. The visible "," has become the most common choice.

```
strFName = "Bob"
strLName = "Smith"
objF = open("MyData.txt", "w")
objF.write(strFName + '\t' + strLName + '\n')
objF.write(strFName + ',' + strLName + '\n')
objF.close()
```

Listing 12

Note: When you want to include a new-line for each line of text data, **you may have to use either "\n", "\r" or "\n\r"** depending on the operations system and version of Python. **Always double-check your output!**



```
Bob	Smith
Bob,Smith
```

Figure x. The results of Listing 12

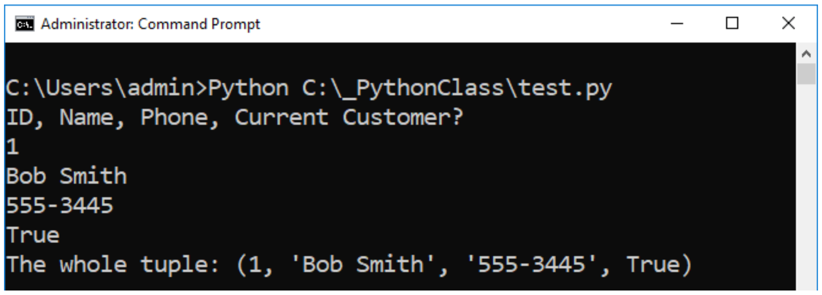
Tuples (Arrays)

Unlike Strings, which only hold character data, the **Tuple collection can hold many types of data**. In many languages, a tuple is **called an "array."**

You **use the parenthesis to indicate which values** you wish to be part of the Tuple collection.

```
print("ID, Name, Phone, Current Customer?")
tplData = (1,"Bob Smith","555-3445", True)
print(tplData[0])
print(tplData[1])
print(tplData[2])
print(tplData[3])
print("The whole tuple:", tplData)
```

Listing 13



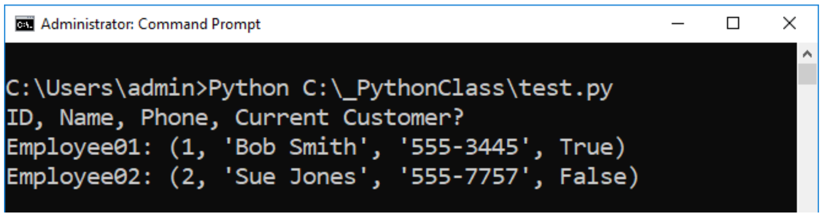
```
C:\Users\admin>Python C:\_PythonClass\test.py
ID, Name, Phone, Current Customer?
1
Bob Smith
555-3445
True
The whole tuple: (1, 'Bob Smith', '555-3445', True)
```

Figure x. The results of Listing 13

Actually, **in Python, you do not need to use parenthesis to create a tuple**, but it is a good idea since it identifies this visually in your code. **Creating a tuple without the parenthesis is called tuple packing.**

```
tplData01 = (1,"Bob Smith","555-3445", True)
tplData02 =  2,"Sue Jones","555-7757", False
print("ID, Name, Phone, Current Customer?")
print("Employee01:", tplData01)
print("Employee02:", tplData02)
```

Listing 14



```
C:\Users\admin>Python C:\_PythonClass\test.py
ID, Name, Phone, Current Customer?
Employee01: (1, 'Bob Smith', '555-3445', True)
Employee02: (2, 'Sue Jones', '555-7757', False)
```

Figure x. The results of Listing 14

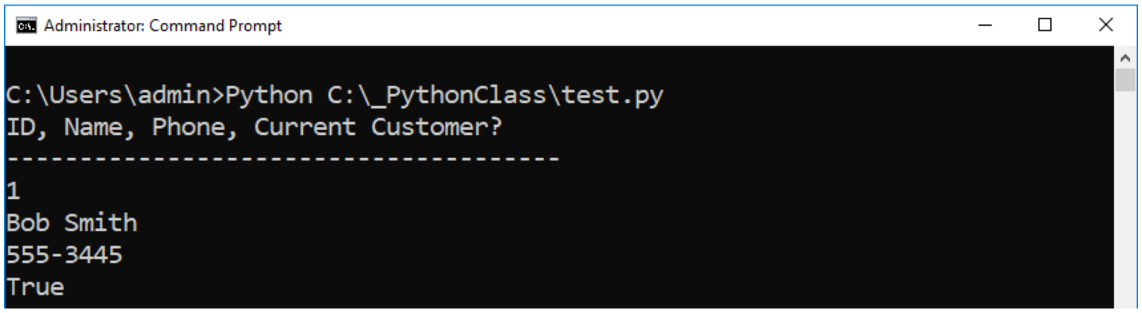
It is useful to **think of a tuple as a "row" of data** like those found in a spreadsheet or database. In fact, many of these applications use tuples (arrays) to hold row data in memory!

Tuples and loops

You can use **loops to access each element from a tuple**. When using a While loop, you may find the length function, **len(), useful since returns the number of values in the tuple**. Here is an example:

```
tplData = (1,"Bob Smith","555-3445", True)
print("ID, Name, Phone, Current Customer?")
print("--------------------------------------")
counter = 0
while counter < len(tplData):
    print(tplData[counter])
    counter += 1
```

Listing 15



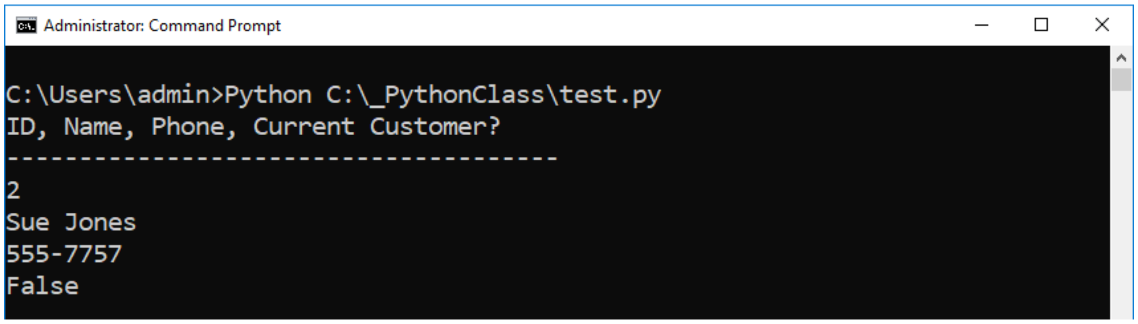
```
C:\Users\admin>Python C:\_PythonClass\test.py
ID, Name, Phone, Current Customer?
--------------------------------------
1
Bob Smith
555-3445
True
```

Figure x. The results of Listing 15

You can use the "**For" loops can extract data from Tuples**. Here is an example:

```
tplData =  2,"Sue Jones","555-7757", False
print("ID, Name, Phone, Current Customer?")
print("--------------------------------------")
for item in tplData:
    print(item) #Output the values
```

Listing 16



```
C:\Users\admin>Python C:\_PythonClass\test.py
ID, Name, Phone, Current Customer?
--------------------------------------
2
Sue Jones
555-7757
False
```

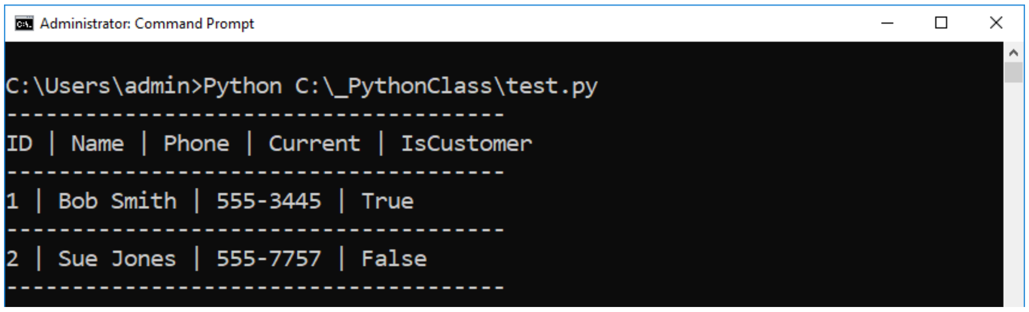
Figure x. The results of Listing 16

Un-packing tuples

Instead of using an Index subscript to extract individual items from a Tuple, or **you can "un-pack" a tuple into individual variables** with the following code.

```
tplData01 = (1,"Bob Smith","555-3445", True)
tplData02 =  2,"Sue Jones","555-7757", False

#Unpack tuple into singleton variables
print("--------------------------------------")
print("ID | Name | Phone | Current | IsCustomer")
print("--------------------------------------")
print(tplData01[0], tplData01[1], tplData01[2], tplData01[3], sep=' |
')
print("--------------------------------------")
#results in the same output as:
intCustId, strName, strPhone, blnCurrent = tplData02 #Unpack tuple
print( intCustId, strName, strPhone, blnCurrent, sep=' | ')
print("--------------------------------------")
```

Listing 17



```
C:\Users\admin>Python C:\_PythonClass\test.py
--------------------------------------
ID | Name | Phone | Current | IsCustomer
--------------------------------------
1 | Bob Smith | 555-3445 | True
--------------------------------------
2 | Sue Jones | 555-7757 | False
--------------------------------------
```

Figure x. The results of Listing 17

The advantage of this option is that it **may make your code easier to read** and work within later parts of your code.

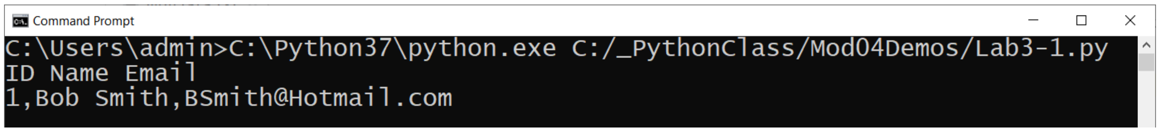
Lab 4-1: Working with Tuples

In this lab, you will program a simple example of a tuple.

1. Create a script that uses a tuple to hold the following data:

```
1   "Bob Smith"   "BSmith@Hotmail.com"
```

2. Add code to print out the row of the data as shown here:



```
C:\Users\admin>C:\Python37\python.exe C:/_PythonClass/Mod04Demos/Lab3-1.py
ID Name Email
1,Bob Smith,BSmith@Hotmail.com
```

Figure x. The results of step 2

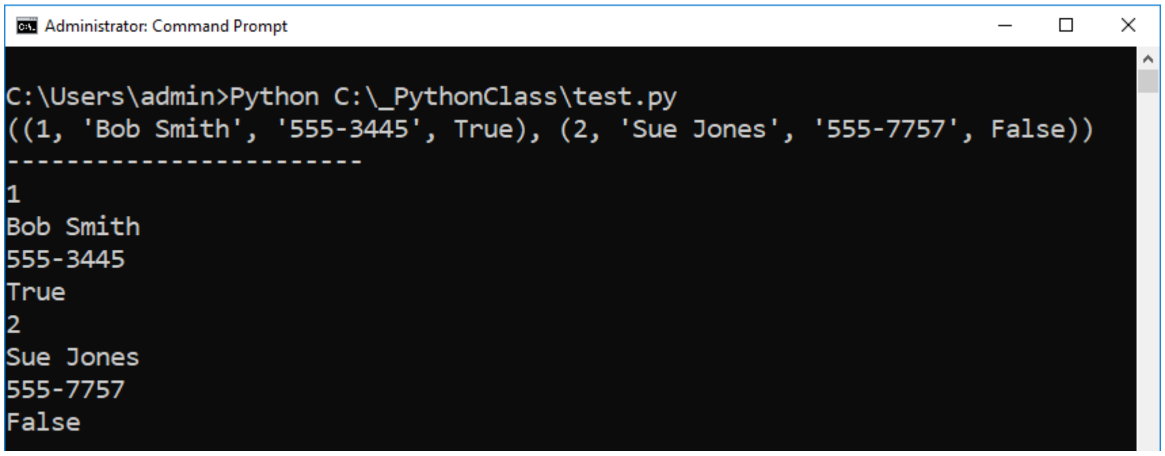
3. Write down how your code works.

Tuples with Multiple Dimensions

Tuples can contain other **"nested" tuples**. This means that there is an **outer tuple and an inner tuple**. Here is an example:

```
tplDataRow01 = (1,"Bob Smith","555-3445", True)
tplDataRow02 =  2,"Sue Jones","555-7757", False
tplDataTableA = (tplDataRow01, tplDataRow02)
print(tplDataTableA)
print('------------------------')
for row in tplDataTableA:
    for col in row:
        print(col)
```

Listing 18



```
C:\Users\admin>Python C:\_PythonClass\test.py
((1, 'Bob Smith', '555-3445', True), (2, 'Sue Jones', '555-7757', False))
------------------------
1
Bob Smith
555-3445
True
2
Sue Jones
555-7757
False
```

Figure x. The results of Listing 18

You can think of this as creating a table of rows.

1	'Bob Smith'	'555-3445'	True
2	'Sue Jones'	'555-7757'	False

The **inner tuple (set of columns)** is considered a first *Dimension* (or *Rank*), while the **outer tuple (set of rows)** is considered the second *Dimension* (or *Rank*).

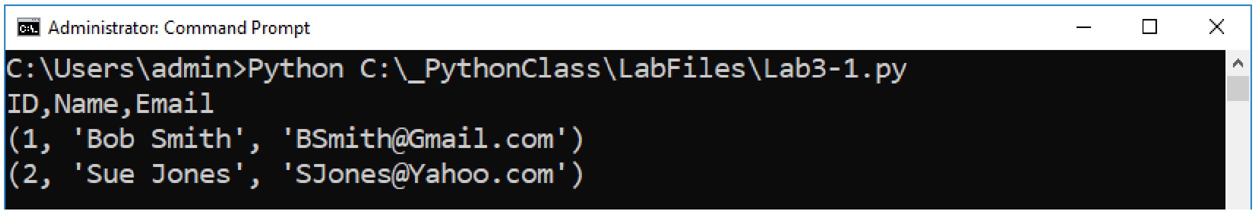
Lab 4-2: Working with Tuples

In this lab, you will program a simple example of a two-dimensional tuple.

1. Create a script that uses a two-dimensional tuple to hold the following data:

```
Id Name       Email
1  Bob Smith  BSmith@Hotmail.com
2  Sue Jones  SueJ@Yahoo.com
```

2. Create a for loop that prints out each row of the data as shown here:

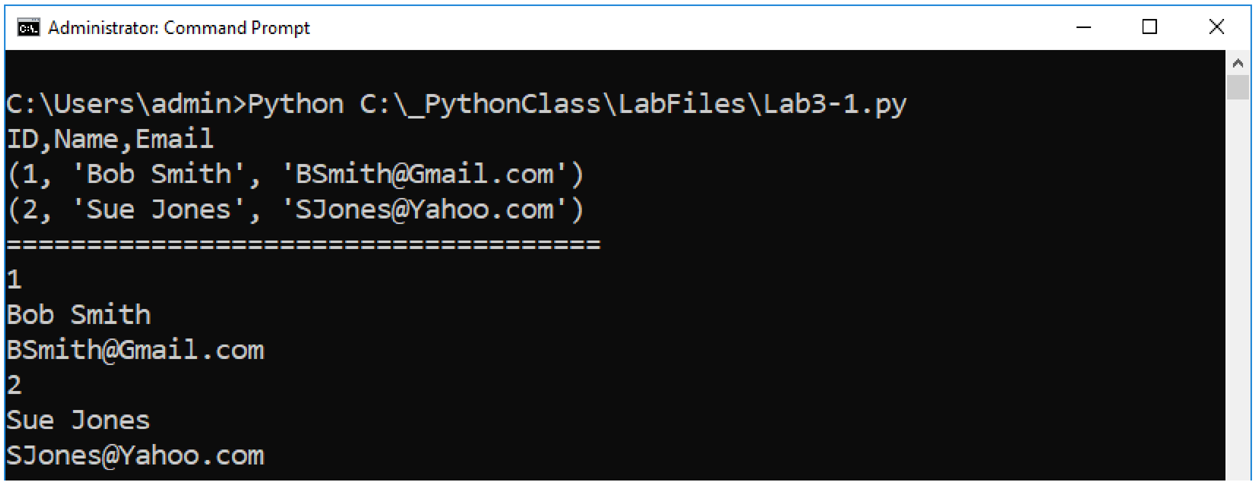


Administrator: Command Prompt

```
C:\Users\admin>Python C:\_PythonClass\LabFiles\Lab3-1.py
ID,Name,Email
(1, 'Bob Smith', 'BSmith@Gmail.com')
(2, 'Sue Jones', 'SJones@Yahoo.com')
```

Figure x. The results of set 2

3. Now added a nested for-loop to extract the individual elements (columns) data and "pivot" the display of data as shown here:



Administrator: Command Prompt

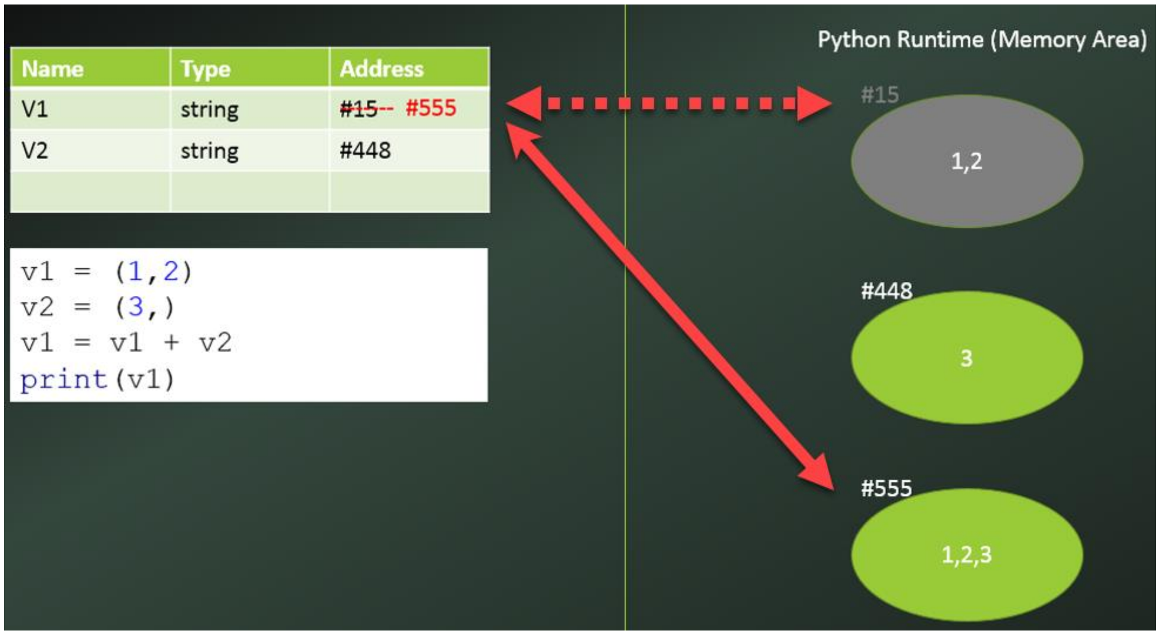
```
C:\Users\admin>Python C:\_PythonClass\LabFiles\Lab3-1.py
ID,Name,Email
(1, 'Bob Smith', 'BSmith@Gmail.com')
(2, 'Sue Jones', 'SJones@Yahoo.com')
=====================================
1
Bob Smith
BSmith@Gmail.com
2
Sue Jones
SJones@Yahoo.com
```

Figure x. The results of step 3

4. Write down how your code works.

Tuples are Immutable

Once a tuple is created, **it seems that you can change the content, but** you actually cannot. Like Strings, a new tuple will be created when you "modify" its collection of values!



Name	Type	Address
V1	string	~~#15~~-- #555
V2	string	#448

```
v1 = (1,2)
v2 = (3,)
v1 = v1 + v2
print(v1)
```

Figure x. Modifying a tuple creates a new tuple in memory

Note: While this makes changing a Tuples costly, **the process is invisible to the user or the developer!**

Adding to a Tuple

To **add data to a tuple,** you use the rather **odd syntax** shown here:

```
tplData = ("1","2","3")
print(tplData)
tplMoreData = ("4"), #Note that the comma at the end makes this a
tuple!!!
tplData += tplMoreData # One tuple added to another works fine!
print(tplData) #Actually another new tuple(just like strings!)

# tplData = tplData + '4' # Adding a string to a tuple does NOT work!
# tplData[2] = 4 # Modifying value in a tuple does NOT work!
```

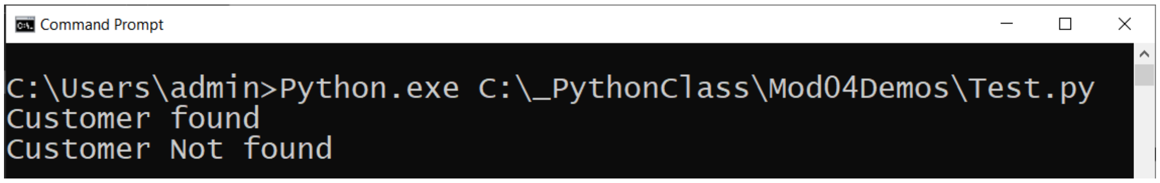
Listing 19

The In operator

The "*in"* operator **searches though a sequence and return a Boolean if found**. Here is an example:

```
tplDataRow01 = (1, "Bob Smith", "555-3445", True)
tplDataRow02 = (2, "Sue Jones", "555-7757", False)
```

```
if "Bob Smith" in tplDataRow01: print("Customer found")
else: print("Customer Not found")

if "Bob Smith" in tplDataRow02: print("Customer found")
else: print("Customer Not found")
```
Listing 20



```
C:\Users\admin>Python.exe C:\_PythonClass\Mod04Demos\Test.py
Customer found
Customer Not found
```
Figure x. The results of Listing 20

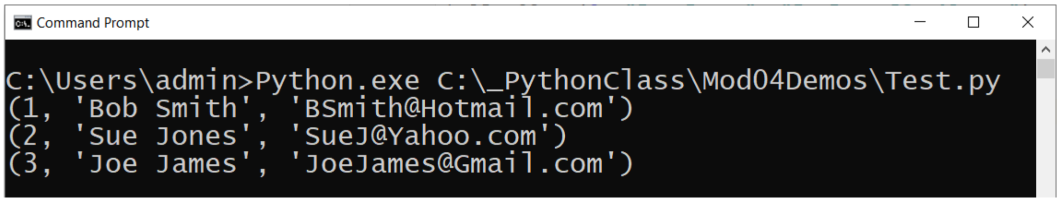
Lab 4-3: The "In" Operator

In this lab, you practice searching for data in a two-dimensional tuple (table).

1. Create a script that uses a two-dimensional tuple to hold the following data: (this is the same data as the last lab's)

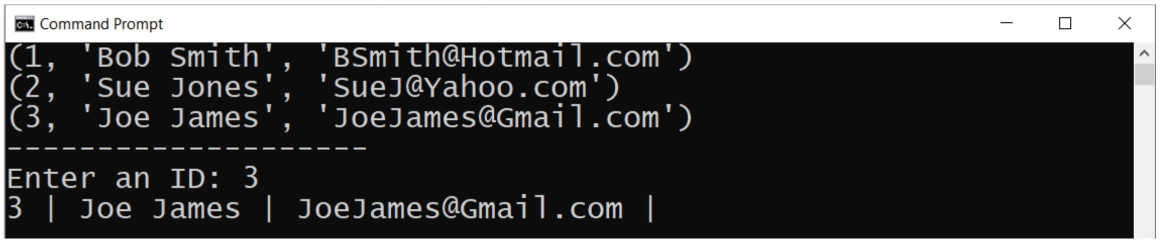
Id	Name	Email
1	Bob Smith	BSmith@Hotmail.com
2	Sue Jones	SueJ@Yahoo.com
3	Joe James	JoeJames@Gmail.com

2. Create a for-loop that prints out each row of the data.



```
C:\Users\admin>Python.exe C:\_PythonClass\Mod04Demos\Test.py
(1, 'Bob Smith', 'BSmith@Hotmail.com')
(2, 'Sue Jones', 'SueJ@Yahoo.com')
(3, 'Joe James', 'JoeJames@Gmail.com')
```
Figure x. The results of step 2

3. Add code that allows a user to display only a single row based on that row's Id value. Format the row using a vertical bar between columns of data.



```
(1, 'Bob Smith', 'BSmith@Hotmail.com')
(2, 'Sue Jones', 'SueJ@Yahoo.com')
(3, 'Joe James', 'JoeJames@Gmail.com')
--------------------
Enter an ID: 3
3 | Joe James | JoeJames@Gmail.com |
```
Figure x. The results of step 3

Tip: don't forget to convert the user's input into an integer list this:

```
int(input("Enter an ID:) "))
```

4. Write down how your code works.

Lists

Lists are very **similar to Tuples but are more flexible**. They include more functions and make changing the List's data more manageable. **We will look at Lists in again in module 05, but since they are highly useful, let's preview the subject now!**

The syntax for a list is almost the same as for a tuple, but **instead of parentheses, you use brackets.** Also, you **do not need to use a "," ** to indicate a list has **only one object** in it.

```
v1 = [1,2]
v2 = [3] #Note that no comma is needed this time
v1 = v1 + v2
print(v1)
```

Listing 21

Another major difference is that **List objects have many more built-in functions** than those Tuples. Here are several outlined in the python help files:

"

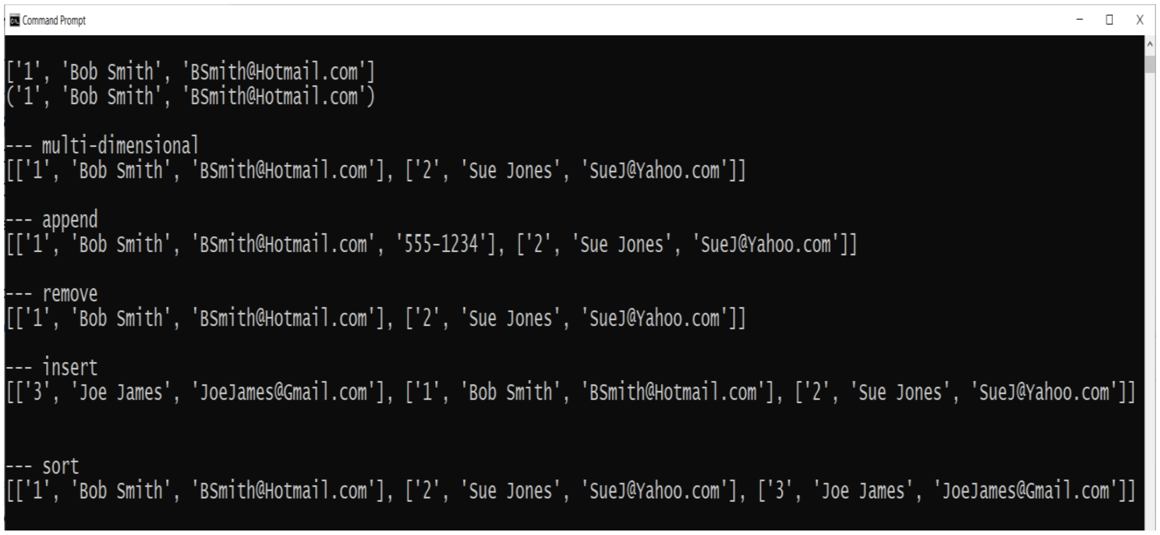
```
        The list data type has some more methods. Here are all of the methods of list objects:”
        list.append(x)
        Add an item to the end of the list; equivalent to a[len(a):] = [x].

        list.extend(L)
        Extend the list by appending all the items in the given list; equivalent to a[len(a):] =
        L.

        list.insert(i, x)
        Insert an item at a given position. The first argument is the index of the element before
        which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a),
        x) is equivalent to a.append(x).

        list.remove(x)
        Remove the first item from the list whose value is x. It is an error if there is no such
        item.

        list.pop([i])
        Remove the item at the given position in the list, and return it. If no index is
        specified, a.pop() removes and returns the last item in the list. (The square brackets
        around the i in the method signature denote that the parameter is optional, not that you
        should type square brackets at that position. You will see this notation frequently in the
        Python Library Reference.)

        list.index(x)
        Return the index in the list of the first item whose value is x. It is an error if there
        is no such item.

        list.count(x)
        Return the number of times x appears in the list.

        list.sort()
        Sort the items of the list, in place.

        list.reverse()
        Reverse the elements of the list, in place.
```

"(https://docs.python.org/3.7/tutorial/datastructures.html#more-on-lists, 2019)

Here are **some examples of list methods and features I find useful**:

```
#1) A list and a tuple are very similar
lstRow1 = ["1","Bob Smith", "BSmith@Hotmail.com"]
tplRow1 = ("1","Bob Smith", "BSmith@Hotmail.com")
print(lstRow1)
print(tplRow1)

#2) Just like tuples your list can be nested (multi-dimensional)
print("\n--- multi-dimensional")
lstRow2 = ["2","Sue Jones", "SueJ@Yahoo.com"]
lstTable = [lstRow1, lstRow2]
print(lstTable)

#3) Lists have a number of extra functions and properties
#3a) You can Append to a list without it recreating a new one
print("\n--- append")
lstRow1.append("555-1234")
print(lstTable)


#3b) You can Remove an item
print("\n--- remove")
lstRow1.remove("555-1234")
print(lstTable)

#3c) You can Insert data into a given spot
print("\n--- insert")
lstRow3 = ["3", "Joe James", "JoeJames@Gmail.com"]
lstTable.insert(0, lstRow3)
print(lstTable)

#3d) You can Sort the data
print("\n--- sort")
lstTable.sort()
print(lstTable)
```

Listing 22



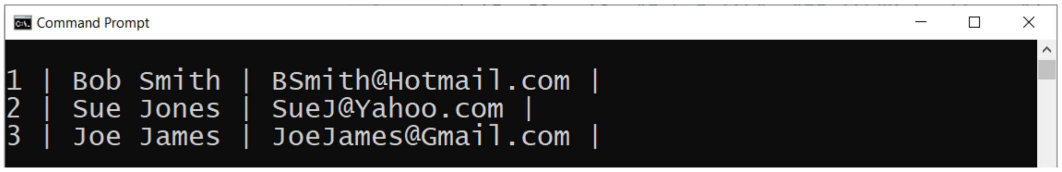
```
['1', 'Bob Smith', 'BSmith@Hotmail.com']
('1', 'Bob Smith', 'BSmith@Hotmail.com')

--- multi-dimensional
[['1', 'Bob Smith', 'BSmith@Hotmail.com'], ['2', 'Sue Jones', 'SueJ@Yahoo.com']]

--- append
[['1', 'Bob Smith', 'BSmith@Hotmail.com', '555-1234'], ['2', 'Sue Jones', 'SueJ@Yahoo.com']]

--- remove
[['1', 'Bob Smith', 'BSmith@Hotmail.com'], ['2', 'Sue Jones', 'SueJ@Yahoo.com']]

--- insert
[['3', 'Joe James', 'JoeJames@Gmail.com'], ['1', 'Bob Smith', 'BSmith@Hotmail.com'], ['2', 'Sue Jones', 'SueJ@Yahoo.com']]


--- sort
[['1', 'Bob Smith', 'BSmith@Hotmail.com'], ['2', 'Sue Jones', 'SueJ@Yahoo.com'], ['3', 'Joe James', 'JoeJames@Gmail.com']]
```

Figure x. The results of listing 22

Lab 4-4: Working with Lists

In this lab, you allow use List instead of a tuple for performing the functionality in Lab 4-2.

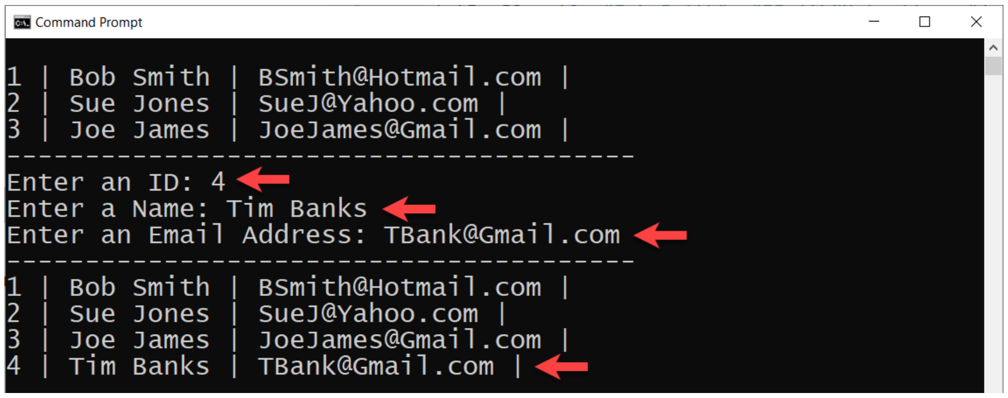
1. Modify your code in Lab 4-2 to use Lists instead of a tuple. Format the rows using a vertical bar between columns of data as shown here:



```
1 | Bob Smith | BSmith@Hotmail.com |
2 | Sue Jones | SueJ@Yahoo.com |
3 | Joe James | JoeJames@Gmail.com |
```

Figure x. The results of step 1

2. Add code that allows a user to add a new row to the table, and then display the updated table of values.



```
1 | Bob Smith | BSmith@Hotmail.com |
2 | Sue Jones | SueJ@Yahoo.com |
3 | Joe James | JoeJames@Gmail.com |
--------------------------------------
Enter an ID: 4
Enter a Name: Tim Banks
Enter an Email Address: TBank@Gmail.com
--------------------------------------
1 | Bob Smith | BSmith@Hotmail.com |
2 | Sue Jones | SueJ@Yahoo.com |
3 | Joe James | JoeJames@Gmail.com |
4 | Tim Banks | TBank@Gmail.com |
```

Figure x. The results of step 2

3. Write down how your code works.

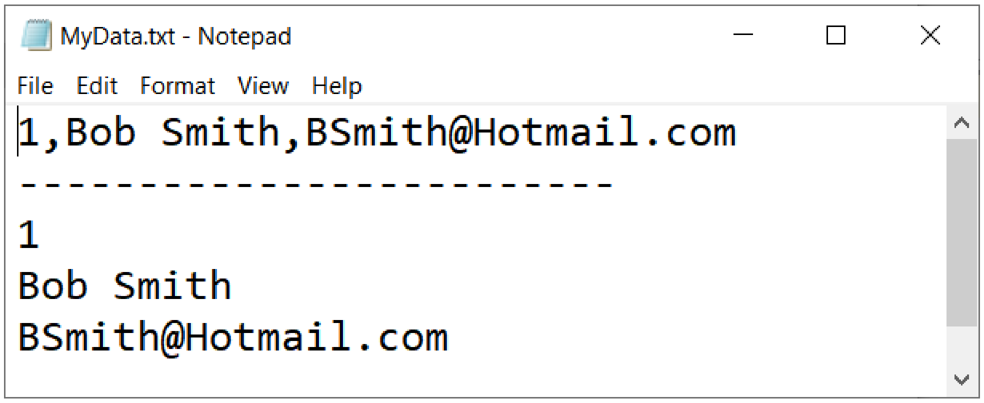
Writing Lists to a File

Like Strings, one of the most common tasks you perform with a list is writing data to a file. When you do so, you **often separate the string by an invisible "tab" character or a visible comma character**. The visible "," has become the most common choice.

```
lstRow = ["1","Bob Smith", "BSmith@Hotmail.com"]

objF = open("MyData.txt", "w")

objF.write(lstRow[0] + ',' + lstRow[1] + ',' + lstRow[2] + '\n')
objF.write('-------------------------\n')

for col in lstRow:
    objF.write(col + '\n')

objF.close()
```

Listing 23

Note: When you want to include a new-line for each line of text data, **you may have to use either "\n", "\r" or "\n\r"** depending on the operations system and version of Python. **Always double-check your output!**



```
1,Bob Smith,BSmith@Hotmail.com
-------------------------
1
Bob Smith
BSmith@Hotmail.com
```

Figure x. The results of listing 23

Summary

In this module, we looked at how you work with collections of data using Python Strings, Tuples, and Lists. We introduced the For-Loop and how it is used with collections.

At this point, you should try and answer as many of the following questions as you can from memory, then review the subjects that you cannot. Imagine being asked these questions by a co-worker or in an interview to connect this learning with aspects of your life.

- What is a collection of data?
- What is another word for a collection in Python?
- A String is a collection of what?
- How do you access individual values in a String?
- A Tuple is a collection of what?
- How do you access individual values in a Tuple?
- A List is a collection of what?
- How do you access individual values in a List?
- How do you write data from a String into a text file?
- How do you write data from a List into a text file?

When you can answer all of these from memory, it is time to complete the module's assignment and move on to the next module.