IE 555 – Programming for Analytics

Module #1 – Introduction to Python

This rather lengthy document is designed to give a broad overview of the Python programming language. Numerous examples are provided to help the reader better understand some useful features (and oddities) of the language.

There are two large sections in this guide. Section 1 demonstrates the Python interactive shell, while Section 2 includes examples of Python scripts.

You should print this guide and bring it to class with you.

1 The Python Interactive Shell

The Python interactive shell simply involves the use of the command line (terminal) to interact with Python. It's a temporary session, meaning that you can't save your commands and re-use them later. However, it's useful for some simple testing of concepts.

To get started, open a terminal/command window and type `python` (or maybe `python3`).

1.1 Setting a Variable

```
x = 3
y = 3.0
z = 1.2e4
inf = float('inf')
```

Note: `x`, `y`, `z`, and `inf` are scalar values.

1.2 PyThoN iS CAse sEnsiTiVe

```
a = 2
A = 3.4
```

`a` and `A` are two different variables

1.3 Variable Naming

The following are valid variable names in Python:

```
a
a1
a_1
aB
myLongVariableName
```

However, these won't work:

- `1a` (can't start with a number)
- `a 1` (can't have a space)
- `a-1` (can't use a dash or other special character besides underscore)
- `a.1`

Unlike some other languages, Python does not require you to declare variables. You can change the type of a variable in Python:

```
>>> r = 3.2
>>> r
3.2

>>> s = int(r)
>>> s
3

>>> int(3.9)
3
```

Python recognizes the following data types: string, float, int, and boolean (among other types that are described at https://www.w3schools.com/python/python_datatypes.asp).

```
>>> a = 9.2

>>> str(a)
'9.2'

>>> float(a)
9.2

>>> int(a)
9

>>> bool(a)
True

>>> bool(0)
False

>>> bool(1)
True

>>> bool('False')
True

>>> bool(False)
False
```

1.4 Basic Math

```
>>> # Addition
>>> 2 + 3
5
>>> 2+3
5

>>> # Multiplication
>>> 3 * 2
6
>>> 3 * 3.2
9.600000000000001


```

```
>>> # Powers
>>> 3 ** 2
9

>>> # Modulus (remainder) operator
>>> 7%4
3

>>> # Division
>>> 7/4
1.75
>>> 8/4
2.0
```

1.5 round, ceil, floor, sqrt

```
>>> round(0.9)
1.0
>>> round(0.2)
0.0
>>> round(0.5)
1.0
>>> round(-0.2)
-0.0
>>> round(-0.9)
-1.0
```

Note: You do not need to import the `math` module to use the `round()` function.

```
round(3.24, 1)
```

RESULT

To use the `ceil`, `floor`, and `sqrt` functions you'll need to import the `math` module. There are a number of ways that you can import this module. The two examples below demonstrate the differences.

```
>>> import math

>>> ceil(3.2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ceil' is not defined

>>> math.ceil(3.2)
4.0

>>> math.floor(3.2)
3.0

>>> math.sqrt(3.2)
1.7888543819998317
```

Now, exit out of Python and then restart Python. We'll import the `math` module in a slightly different manner.

```
>>> from math import *

>>> ceil(3.2)
4.0

>>> math.ceil(3.2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined

>>> floor(3.2)
3.0

>>> sqrt(3.2)
1.7888543819998317
```

In general, it is a bad idea to import *.

1.6 Data Structures

Python supports the following data structures:

- Lists (ordered and changeable)
- Dictionaries (key/value pairs)
- Sets (unordered and unique)
- Tuples (ordered and unchangeable)

1.6.1 Lists

You can think of lists as arrays.

```
>>> d = [3, 4, 5]
>>> d
[3, 4, 5]

>>> print d
  File "<stdin>", line 1
    print d
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you
   mean print(d)?

>>> print(d)
[3, 4, 5]
```

Lists are "indexed", meaning that you can access individual entries in a list.

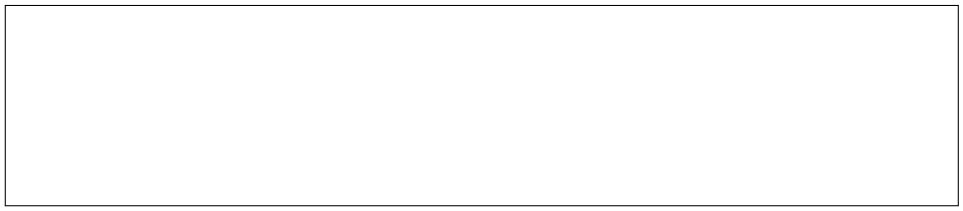
NOTE: List indices start at 0. Thus, the first item in a list is at index 0.

```
>>> d = [3, 4, 5]
>>> d[0]
3
>>> d[1]
4
>>> d[2]
5

>>> d[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Let's create an empty list and then add an element/entry/item to it:

```
d = []
d.append(97)
print(d)
print(len(d))
print(d[0])
```



Result

What happens if we try to add an entry this way?

```
myList = []
myList[0] = 34
```

Result

Try this:

```
myList = []

myList.append(82)
print(myList)

myList[0] = 34
print(myList)

myList.append('Bulls')
print(myList)
```

Result

Now let's try removing items from our list:

```
myList = [3, 9.2, 'bus', '9.2', 1.3e4]
myList.remove(9.2)
myList
```

Result

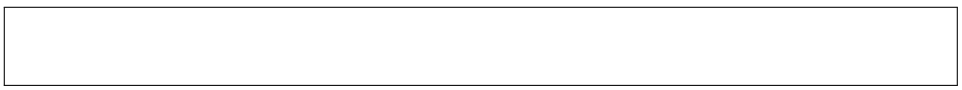
```
myList.append(9.2)
myList
```

Result

```
myList.append(9.2)
myList
```

Result

```
myList.remove('bus')
myList
```



Result

```
myList.remove(9.2)
myList
```

Result

```
myList.remove(37)
```

Result

See http://stackoverflow.com/questions/1157106/remove-all-occurrences-of-a-value-from-a-python-list for suggestions on deleting *all* matching elements from a list.

As we just saw, we can mix data types in a list:

```
>>> v = [3, 'fds', 4.5]
>>> v
[3, 'fds', 4.5]
```

There are no "matrices" in Python (* we'll talk about the `numpy` module later). However, you can have nested lists that behave like multi-dimensional arrays:

```
u = [[1, 2], 3, [4], [5, [6, 7, 8]]]
u[0]
```

Result

```
u[1]
```

Result

```
u[2]
```

Result

```
u[3]
```

Result

```
u[3][1]
```

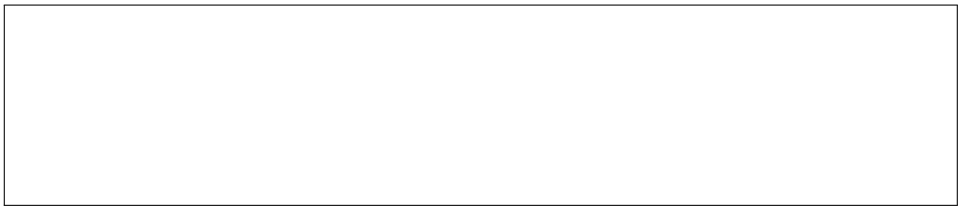
Result

```
u[3][1][2]
```

Result

Let's get some information about our list. Specifically, we want to know the length of the list, the min value in the list, and the max value.

```
pdq = [3.2, 7, 4.9]
len(pdq)
min(pdq)
max(pdq)
```



Result

Now, suppose our list has an empty element in it (a sub-list with nothing in it):

```
d = [[],4,5]
len(d)
min(d)
max(d)
```

Result

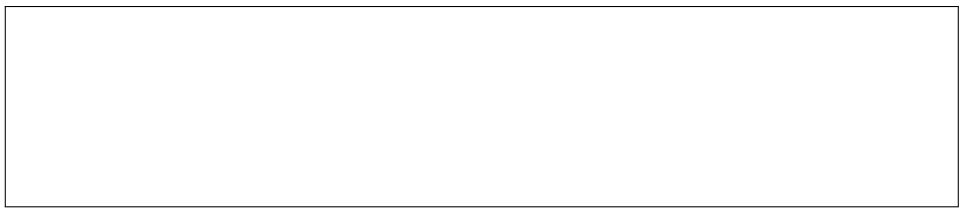
The problem is that Python doesn't know how to handle a comparison between different data types. Here's one (not particularly efficient) way to make this work as we would expect:

```
d = [[],4,5]
myMin = float('inf')
myMax = -float('inf')
for val in d:
    if (type(val) in [int, float]):
        myMin = min(val, myMin)
        myMax = max(val, myMax)
print(myMin)
print(myMax)
```

Result

Caution: Setting one list equal to another does **not** mean that a copy was created.

```
a = [1, 2, 3]
b = a             # Not a copy
c = list(a)       # Copy
a[0] = 999
a
b
c
```



Result

1.6.2 Ranges

We can also create lists using the `range()` function. Range can be used in two ways:

1. `range(stop)`
2. `range(start, stop[, step])` (the "`[, step]`" part is optional)

```
range(4)
```

Result

```
list(range(4))
```

Result

```
list(range(0, 4))
```

Result

```
list(range(0, 4, 2))
```

Result

```
list(range(0, 5, 2))
```

Result

```
list(range(0, 4, 1.5))
```

Result

```
list(range(1,4))
```

Result

```
list(range(1,4,2))
```

Result

```
list(range(5,1))
```

Result

```
list(range(5,1,-1))
```

Result

For more info, see https://docs.python.org/3.7/library/functions.html#func-range.

1.6.3 Dictionaries

Dictionaries allow you to index values using "keys", which could be numbers or strings. Dictionaries are unordered collections of *key: value* pairs.

```
>>> myDictionary = {'color': 'blue', 'age': 87, 9: 32, '
    anArray': [3, 5, 7]}
>>> myDictionary
{'color': 'blue', 9: 32, 'age': 87, 'anArray': [3, 5, 7]}

>>> myDictionary['color']
'blue'
>>> myDictionary[9]
32
>>> myDictionary['anArray']
[3, 5, 7]
>>> myDictionary['anArray'][1]
5
```

Let's add some items to our dictionary. There are several ways to do this.

```
>>> myDictionary['speed'] = 55
>>> myDictionary
{'color': 'blue', 9: 32, 'age': 87, 'anArray': [3, 5, 7], '
    speed': 55}
```

```
>>> myDictionary.update({'velocity': 32.8})
>>> myDictionary
{'velocity': 32.8, 'color': 'blue', 9: 32, 'age': 87, 'anArray
    ': [3, 5, 7], 'speed': 55}
```

```
>>> myDictionary['color'] = 'red'
>>> myDictionary
{'velocity': 32.8, 'color': 'red', 9: 32, 'age': 87, 'anArray'
    : [3, 5, 7], 'speed': 55}
```

```
>>> myDictionary.update({'color': 'green'})
>>> myDictionary
{'velocity': 32.8, 'color': 'green', 9: 32, 'age': 87, '
    anArray': [3, 5, 7], 'speed': 55}
```

```
>>> myDictionary.update({'time': 'future'})
>>> myDictionary
{'time': 'future', 'velocity': 32.8, 'color': 'green', 9: 32,
   'age': 87, 'anArray': [3, 5, 7], 'speed': 55}
```

Let's remove some items from our dictionary:

```
>>> myDictionary
{'time': 'future', 'velocity': 32.8, 'color': 'green', 9: 32,
   'age': 87, 'anArray': [3, 5, 7], 'speed': 55}
>>> del myDictionary['color']
>>> myDictionary
{'time': 'future', 'velocity': 32.8, 9: 32, 'age': 87, '
   anArray': [3, 5, 7], 'speed': 55}
```

Is 'time' a key in our dictionary?

```
>>> myDictionary
{'time': 'future', 'velocity': 32.8, 9: 32, 'age': 87, '
   anArray': [3, 5, 7], 'speed': 55}
>>> 'time' in myDictionary
True
```

Is 'future' a key in our dictionary?

```
>>> myDictionary
{'time': 'future', 'velocity': 32.8, 9: 32, 'age': 87, '
   anArray': [3, 5, 7], 'speed': 55}
>>> 'future' in myDictionary
False
```

Is 'future' a value associated with the key 'time' in our dictionary?

```
>>> myDictionary
{'time': 'future', 'velocity': 32.8, 9: 32, 'age': 87, '
   anArray': [3, 5, 7], 'speed': 55}
>>> 'future' in myDictionary['time']
True
```

What are all of the keys in our dictionary?

```
>>> list(myDictionary.keys())
```

```
['time', 'velocity', 9, 'age', 'anArray', 'speed']
```

What are all of the values in our dictionary?

```
>>> list(myDictionary.values())
['future', 32.8, 32, 87, [3, 5, 7], 55]
```

More info on dictionaries can be found here:
https://docs.python.org/3.7/tutorial/datastructures.html#dictionaries

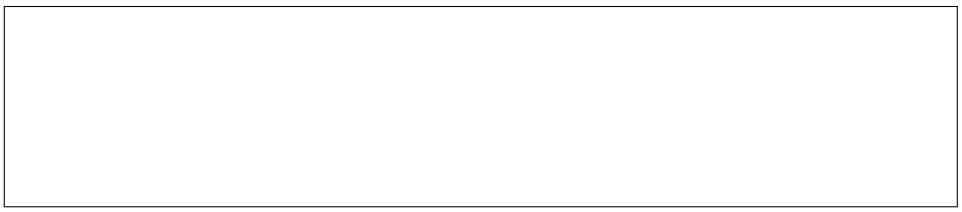
1.6.4 Sets

Another useful data type is the "set." Sets are unordered collections, like dictionaries. However, sets cannot be indexed. You can use operations like union, intersection, and difference on sets.

```
>>> mySet = {1, 99, 'purple', 3.14}
>>> mySet
{'purple', 1, 99, 3.14}
```

What happens if we try to access the first element of `mySet`?

```
mySet[0]
```



Result

Let's create a second set:

```
>>> myOtherSet = {22, 99, 'purple', 'red', 3.1417}
>>> myOtherSet
{3.1417, 'purple', 99, 22, 'red'}
```

Let's test out some set theory:

```
>>> mySet
{'purple', 1, 99, 3.14}
>>> myOtherSet
{3.1417, 'purple', 99, 22, 'red'}

>>> # Find the union of these two sets:
>>> mySet | myOtherSet
{1, 99, 3.14, 3.1417, 'red', 'purple', 22}

>>> # Find everything that's in mySet but not in myOtherSet:
>>> mySet - myOtherSet
{1, 3.14}

>>> # Find everything that's in myOtherSet but not in mySet:
>>> myOtherSet - mySet
{3.1417, 'red', 22}

>>> # Find the intersection of these two sets:
>>> mySet & myOtherSet
{'purple', 99}

>>> # Find elements that are in exactly one of the two sets:
>>> mySet ^ myOtherSet
{1, 3.1417, 3.14, 'red', 22}
```

We can modify sets as follows:

```
>>> # Create an empty set:
>>> a = set()
>>> a
set()

>>> # Add individual values to the set:
>>> a.add(19)
>>> a
{19}
>>> a.add(11)
>>> a
{11, 19}

>>> # We can also add "iterable" values to the set using "
   update":
>>> a.update([3, 4])
>>> a
```

```
{11, 3, 19, 4}

>>> # We get an error if we try to use "update" with non-
    iterable values:
>>> a.update(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable

>>> a.update([5])
>>> a
{3, 4, 5, 11, 19}


>>> # We can remove values:
>>> a.remove(19)
>>> a
{3, 4, 5, 11}

>>> # We can use the union operator to add values to the set:
>>> a = a | {33, 34}
>>> a
{33, 34, 3, 4, 5, 11}
```

More info on sets can be found here:
https://docs.python.org/3.7/tutorial/datastructures.html#sets

1.6.5 Tuples

"Tuples" are ordered collections (like lists), but they cannot be changed. Because tuples are immutable, they are often faster for the computer to process.

```
>>> myTuple = (1, 99, 'purple', 3.14)
>>> myTuple
(1, 99, 'purple', 3.14)
```

If you want a tuple with just one element, you'll need to add a trailing comma:

```
>>> notAtuple = ('hi')
>>> notAtuple
'hi'
>>> type(notAtuple)
<class 'str'>

```

```
>>> aTuple = ('hello',)
>>> aTuple
('hello',)
>>> type(aTuple)
<class 'tuple'>
```

You cannot edit a tuple.

```
>>> aTuple = ('hello',)
>>> aTuple[0]
'hello'
>>> aTuple[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

More info on tuples can be found here:
https://www.w3schools.com/python/python_tuples.asp

1.6.6 Comparing Lists, Dictionaries, Sets, and Tuples

- Initialize as empty:

```
myList = []
# or
myList = list()

myDictionary = {}
# or
myDictionary = dict()

mySet = set()

myTuple = ()
# or
myTuple = tuple()
```

- Initialize with values:

```
myList = [19, 37, 'cars']

myDictionary = {1: 19, 'blue': 37, 99: 'cars'}

mySet = {19, 37, 'cars'}

myTuple = (19, 37, 'cars',)  # trailing comma is required
   if just one element
```

- Accessing individual values:

```
>>> myList[0]
19

>>> myDictionary[99]
'cars'

>>> mySet[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing

>>> myTuple[0]
19
```

See above for details on adding/deleting/updating values in lists, dictionaries, and sets. (Tuples cannot be modified.)

1.7 Printing

```
>>> x = 1/3.0
>>> y = 7
>>> z = 'hi'

>>> print("The value of x is: ", x)
The value of x is:  0.333333333333

>>> print("x = %f" % (x))
x = 0.333333

>>> print("x = %.2f" % (x))
```

```
x = 0.33

>>> print('x = %.3f, y = %d, z = %s' % (x, y, z))
x = 0.333, y = 7, z = hi

>>> print('x = ', x, 'y = ', y, 'z = ', z)
x =  0.333333333333 y =  7 z =  hi

>>> print('x printed as an integer is %d' % (x))
x printed as an integer is 0
```

A note on the differences between single and double quotes:

```
>>> print("It's OK to use a single tic inside double quotes.")
It's OK to use a single tic inside double quotes.

>>> print('However, it's not OK to use a single tic inside
   single quotes.')
  File "<stdin>", line 1
    print('However, it's not OK to use a single tic inside
   single quotes.')
                        ^
SyntaxError: invalid syntax

>>> print('Instead, you need to "escape" the tic character.
   It\'s easy!')
Instead, you need to "escape" the tic character.  It's easy!
```

Another option is to use the `str.format` method.

```
>>> x = 1/3.0
>>> y = 7
>>> z = 'hi'

>>> print("The value of x is: {}".format(x))
The value of x is:  0.333333333333

>>> print("x = {0:.2f}".format(x))
x = 0.33

>>> print('x = {0:.2f}, y = {1:d}, z = {2}'.format(x, y, z))
x = 0.333, y = 7, z = hi

>>> print('x = {}, y = {}, z = {}'.format(x,y,z))
x =  0.333333333333 y =  7 z =  hi
```

```
16
17 >>> print('x printed as an integer is {}'.format(x))
18 x printed as an integer is 0
```

More info on string formatting can be found here:
https://www.w3schools.com/python/python_string_formatting.asp

1.8 Comparison Operators

There's a difference between **assignment** and **comparison** operators. For example:

- `a = 2` → Assign variable `a` to a value of `2`.
- `a == 2` → Check if the value of `a` is equal to `2`.

The following table comes from https://docs.python.org/3/library/stdtypes.html

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

1.9 Boolean Operations

```
>>> (1 < 2) and (2 < 5)
True

>>> (1 < 2) or (7 < 5)
True

>>> not False
True

>>> (2 < 1) or ((1 < 5) and (2 < 3))
True

>>> x = 4
>>> (1 < x <= 5)
True

>>> (1 < x) and (x <= 5)
True
```

1.10 If Statements

Python uses indentation (spaces or tabs) to control nested statements.

```
>>> x = 2
>>> if (x <= 3):
...     print('hello')
...     print(x)
...
hello
2
```

2 Python Scripts

The Python interactive shell is a nice place to test some code snippets or to use Python as a calculator. However, once you exit the shell, all of your code is lost.

If you want to write code that you can re-use, Python scripts are the way to go. A Python script is simply Python code that is saved in a plain text file with a `.py` extension.

2.1 Where to Write your Code

You may write Python scripts in any **plain text** editor, such as Notepad. Do not write code in a rich text editor like Word.

Many programmers prefer to use Integrated Development Environments (IDEs), which are customized text editors specifically for code writing. These IDEs do helpful things like colorize your code, auto-complete variables, and auto-indent. Here are some suggestions:

- Spyder (comes with Anaconda)
- VS Code (comes with Anaconda)
- TextWrangler (Mac)
- Geany (Windows, Mac, Linux)
- PyCharm
- Sublime Text Editor
- https://wiki.python.org/moin/IntegratedDevelopmentEnvironments
- Do a Google search for "Python IDE"

2.2 Comments

```
# Start a line with the pound sign (#) to create a comment.
a = "here's a #, not a comment"
```

```
'''
Or, you can do a block comment
by enclosing text within
three tic marks (the key to the left of the Enter key).
'''
```

2.3 Indentation

Python handles nested conditions via indentation (spaces or tabs)

Python

```
if (x == 2):
    y = x + 1
```

C

```
if (x == 2)  {
    y = x + 1;
}
```

MATLAB

```
if (x == 2)
    y = x + 1
endif
```

Caution: Python is very particular about consistency in indentation. The following code will give you an error:

```
if (x == 2):
    y = x + 1   # Indented with a single tab
    z = y + 1   # Indented with 4 spaces
```

The error message is "IndentationError: unindent does not match any outer indentation level."

2.4 Example 1

For our first example we'll create the ubiquitous 'hello world' program.

Start by opening your favorite plain text editor. Then, type the contents of the code below into your empty text file. Finally, save this file as `example_1.py`. *Make sure you take note of where this file is saved.*

example_1.py

```
# File:  example_1.py
# This is our first Python script.  It shows some basic Python
    features.
#
# How to Run:
# 1) Open a terminal/command window.
# 2) Change directories to the location where this file is
   saved.
# 3) Type the following command at the prompt:
#    python example_1.py

print('Hello World!')
```

The script begins with some comments that will help us to understand what this code does and how to run it. Next, we print a statement to the terminal.

Let's run our code:

1. Open a terminal window.
2. Change directories to where this file was saved.
3. `python example_1.py`

example_1.py, part 2

```
# Create a list:
a = [3, 5, 11, 97]
print(a)

# Loop over the values in this list:
print("Printing the values of list a:")
for myValue in a:
    print(myValue)
```

example_1.py, part 3

```
# Here's another way to iterate over the values in the list:
print("\nPrinting the values of a in a different way.")
print("Can you tell a difference in the output?")
for i in range(0, len(a)):
    print(a[i], end='')  # What does the 2nd term do?
print("I wish this was on a new line")
```

example_1.py, part 4

```
# Create a dictionary:
myDictionary = {4: 'yellow', 'zebra': 99, 'delta': 2.81}
print(myDictionary)

# Loop over the keys in this dictionary:
for myKey in myDictionary:
    print("myDictionary[", myKey, "] = ", myDictionary[myKey])
```

example_1.py, part 5

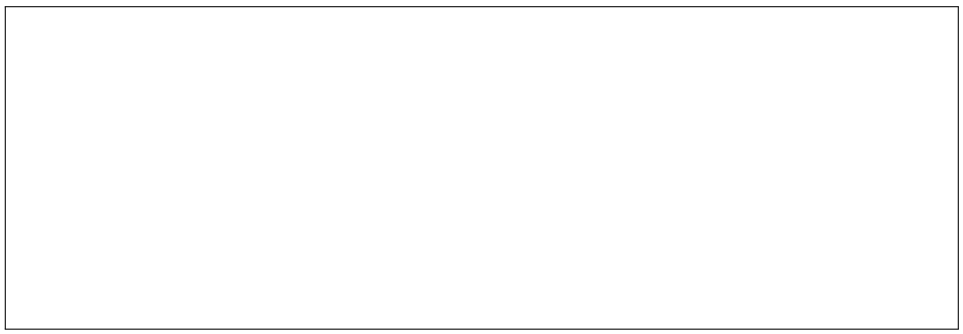
```
# Check if 4 is a key in myDictionary.
# If it is, print "Hooray".
# If it's not, print "Boo".
if (4 in myDictionary):
    print("Hooray")
else:
    print("Boo")
```

example_1.py, part 6

```
# Check if 5 is in list a or if it's a key in myDictionary.
# We'll tab our response when printing (so it's indented).
if (5 in a):
    print("\tHooray, it's in a")
elif (5 in myDictionary):
    print("\tHooray, it's in myDictionary")
else:
    print("\tBoo")
```

example_1.py, part 7

```
# Initialize x = 2
# Add 0.2 to x until x is greater than or equal to 3.3
x = 2
y = 0
while (x <= 3.3):
    x += 0.2
    y = y + 1
    print("x = {}".format(x))
print("\nAll Done!" )
print("\tx = %.3f, y = %d" % (x, y))
```



Result

2.5 Example 2

Our second example is a little more practical and has a focused purpose. Specifically, we are going to store some information about people we know. The program will search for a particular person and report information about that person.

There are two key features of this example:

1. We'll see how to use a dictionary of dictionaries to store the contact information shown in Table 1, and
2. We'll use command line arguments to pass information to the Python script.

Table 1: Contact information data for Example 2.

				Home Location	
Name	**Phone**	**Age**	**Occupation**	**x**	**y**
string	*string*	*int*	*string*	*float*	*float*
John	555-1212	57	rocket scientist	5	27
Mary	555-5555	42	brain surgeon	17	8

Let's start by opening a new plain text file, named `example_2.py`. Type the code below into the text file:

example_2.py, part 1

```
import sys

# FIXME -- Write a description of this program here.

'''
How to Run:
1) Open a terminal/command window.
2) Change directories to the location where this file is saved.
3) Type the following command at the prompt:
   python example_2.py <person's name> <travel speed in miles/hour>
   python example_2.py John 1.2
'''

# Capture 2 inputs from the command line.
#    NOTE: sys.argv[0] is the name of the python file
#    Try "print(sys.argv)" (without the quotes) to see the sys.argv list
#    2 inputs --> the sys.argv list should have 3 elements.
if (len(sys.argv) == 3):
    print("\tOK.  2 command line arguments were passed.")

    # This will print out sys.argv[1] and sys.argv[2]:
    print(sys.argv[1:3])

    # Now, we'll store the command line inputs to variables
    inputName = str(sys.argv[1])        # This is a string
    inputSpeed = float(sys.argv[2])     # This is a float (decimal) number
else:
    print('ERROR: You passed', len(sys.argv)-1, 'input parameters.')
    quit()

print("inputName = %s" % (inputName))
print("inputSpeed = %f miles/hour" % (inputSpeed))
```

NOTES:

1. We could hard-code the values of inputName and inputSpeed directly into our Python script. However, each time we use the program we would need to manually edit/re-save the Python script, which isn't very efficient if we're going to use this script repeatedly for different users.
2. Alternatively, we could prompt the user to type this information when the program is running. This would work well if we want an interactive program, but it requires the user to be present when the program is running.
3. Thus, what we have created is a script that could be run "in the background", without needing a user to interact directly with the program. *What's an application for this type of program?*

Now, let's edit our script to include some contact information for a user named "John":

example_2.py, part 2

```
33 # Create a dictionary of dictionaries.
34 # Start with an empty dictionary:
35 contactInfo = {}
36
37 # Our contactInfo dictionary has a key of 'John'.  That
      dictionary also contains
38 # a dictionary with other keys ('phone', 'age', etc.).
39 contactInfo['John'] = {'phone': '555-1212', 'age': 57, 'job':
      'rocket scientist', 'home': [5, 27]}
40
41 # Let's see what we have:
42 print(contactInfo['John'])
```

To make things a little more interesting, let's add a second user:

example_2.py, part 3

```
43 # We'll add another key for user 'Mary':
44 contactInfo['Mary'] = {'phone': '555-5555', 'age': 42, 'job':
      'brain surgeon', 'home': [17, 8]}
45
46 # Let's print out some individual values from our dictionary:
47 print(contactInfo['John']['phone'])
48 print(contactInfo['Mary']['home'])
49 print(contactInfo['Mary']['home'][0])
50 print(contactInfo['Mary']['home'][1])
```

Now, let's make use of one of our command line inputs. Specifically, we want to see if the name provided in the command line is in our records. If it is, we'll print some information about that person. If it's not in our dictionary, we'll print an error message.

example_2.py, part 4

```
51 if (inputName in contactInfo):
52     print("Here's some information about %s:" % (inputName))
53     print("\tPhone Number: %s" % (contactInfo[inputName]['
       phone']))
54     print("\tAge: %d" % (contactInfo[inputName]['age']))
55     print("\tOccupation: %s" % (contactInfo[inputName]['job'])
       )
56     print("\tHome Location: x = %f, y = %f" % (contactInfo[
       inputName]['home'][0], contactInfo[inputName]['home'][1]))
57 else:
58     print("Sorry,", inputName, "(case sensitive) isn't in our
       records.  Goodbye.")
59     quit()
```

2.6 Example 3

This example builds on Example 2 to add "functions."

Let's start by copying `example_2.py` to `example_3.py`. Open `example_3.py` in your text editor and update the preamble of the file (the top part) with the correct file name. Then, remove some of the excess print statements and comments to make your code more concise. Your new file should look like this:

example_3.py, part 1

```
1  import sys
2
3  # FIXME -- Write a description of this program here.
4  #
5  # How to Run:
6  # 1) Open a terminal/command window.
7  # 2) Change directories to the location where this file is saved.
8  # 3) Type the following command at the prompt:
9  #    python example_3.py <person's name> <travel speed in miles/hour>
10 #    python example_3.py John 1.2
11
12 # Capture 2 inputs from the command line.
13 #    NOTE: sys.argv[0] is the name of the python file
14 #    Try "print(sys.argv)" (without the quotes) to see the sys.argv list
15 #    2 inputs --> the sys.argv list should have 3 elements.
16 if (len(sys.argv) == 3):
17     print("\tOK.  2 command line arguments were passed.")
18
19     # Now, we'll store the command line inputs to variables
20     inputName = str(sys.argv[1])        # This is a string
21     inputSpeed = float(sys.argv[2])     # This is a float (decimal) number
22 else:
```

```
    print('ERROR: You passed', len(sys.argv)-1, 'input parameters.')
    quit()

print("inputName = %s" % (inputName))
print("inputSpeed = %f miles/hour" % (inputSpeed))


# Create a dictionary of dictionaries.
# Start with an empty dictionary:
contactInfo = {}
contactInfo['John'] = {'phone': '555-1212', 'age': 57, 'job': 'rocket
    scientist', 'home': [5, 27]}
contactInfo['Mary'] = {'phone': '555-5555', 'age': 42, 'job': 'brain
    surgeon', 'home': [17, 8]}

if (inputName in contactInfo):
    print("Here's some information about %s:" % (inputName))
    print("\tPhone Number: %s" % (contactInfo[inputName]['phone']))
    print("\tAge: %d" % (contactInfo[inputName]['age']))
    print("\tOccupation: %s" % (contactInfo[inputName]['job']))
    print("\tHome Location: x = %f, y = %f" % (contactInfo[inputName]['home
    '][0], contactInfo[inputName]['home'][1]))
else:
    print("Sorry,", inputName, "(case sensitive) isn't in our records.
    Goodbye.")
    quit()
```

We want our new script to calculate the travel distance and the travel time from the matching contact person to each of the locations shown in Table 2.

Table 2: Location information for Example 3.

Location ID	**x**	**y**
int	*float*	*float*
101	4.1	11.5
102	6.2	21.0
113	2.7	13.6
237	8.0	42.2

Let's create a dictionary in `example_3.py` to store this information:

example_3.py, part 2

```
# Create an empty dictionary:
myLocations = {}

# For each locationID, store the x and y coordinates:
myLocations[101] = [4, 11]
myLocations[102] = [6, 21]
myLocations[113] = [2, 13]
myLocations[237] = [8, 42]
```

Now, we want to find the distance between our person's home and each of these new locations. For now, suppose we have a tool that will tell us the distance according to the Manhattan metric (like walking on the streets in NYC). This tool requires four pieces of information:

1. The x-coordinate of the home location,
2. The y-coordinate of the home location,
3. The x-coordinate of the destination, and
4. The y-coordinate of the destination.

If we give these coordinates to our tool, the tool will simply tell us the distance. We're going to create this tool (or "function"). Functions take inputs, do some calculations, and return something.

Let's edit our `example_3.py` script so it calls a function named `getManhattanDistance()` with the four coordinates described above. We'll use the resulting distance and the speed value provided in the command line arguments to also calculate the walking time. We want this information for each of the destination locations.

example_3.py, part 3

```
53 xHome = contactInfo[inputName]['home'][0]
54 yHome = contactInfo[inputName]['home'][1]
55 for id in myLocations:
56     xLoc = myLocations[id][0]
57     yLoc = myLocations[id][1]
58     distance = distanceFunctions.getManhattanDistance(xHome,
   yHome, xLoc, yLoc)
59     print("The walking distance from %s's home to location %d
   is %.2f miles." % (inputName, id, distance))
60     print("\tTraveling at %f miles/hour, it will take %s %.1f
   seconds to reach the destination." % (inputSpeed, inputName
   , (distance/inputSpeed)*60.0*60.0))
```

We need to write the function now. To do this, open an empty text file and save it as `distanceFunctions.py`. This file should be saved in the same directory where you saved `example_3.py`. Next, type the contents below into `distanceFunctions.py`.

distanceFunctions.py, part 1

```
1 # FILE:  distanceFunctions.py
2 #
3 # This file contains functions related to calculating
   distances.
4
5 def getManhattanDistance(x1, y1, x2, y2):
6     myDistance = abs(x1-x2) + abs(y1-y2)
7
8     return myDistance
```

At the top of your `example_3.py` file, you need to import the `distanceFunctions` module that we just created:
`import distanceFunctions`
Now, you can run your script.

Let's add another function to `distanceFunctions.py`. We'll call this function `getDirections()`. It will take the same inputs as `getManhattanDistance()`, but it will calculate something different. Specifically, `getDirections()` will return two boolean values:

1. `travelEast` – True if our person must travel east to get to the destination (False otherwise), and
2. `travelNorth` – True if our person must travel north to get to the destination (False otherwise).

distanceFunctions.py, part 2

```
9  def getDirections(xOrig, yOrig, xDest, yDest):
10     if (xOrig < xDest):
11         travelEast = True
12     else:
13         travelEast = False
14
15     if (yOrig < yDest):
16         travelNorth = True
17     else:
18         travelNorth = False
19
20     return (travelEast, travelNorth)
```

Now, back to our `example_3.py` script, let's call the new `getDirections()` function within our `for` loop:

example_3.py, part 4

```
61     [doesTravelEast, doesTravelNorth] = distanceFunctions.
       getDirections(xHome, yHome, xLoc, yLoc)
62     print("\tTravel East?", doesTravelEast)
63     print("\tTravel North?", doesTravelNorth)
```

2.7 Example 4

In this last example we'll see how to read from, and write to, text files.

Let's start by creating a file that will store data we'll want to access from a Python script. This file will be saved in the `.csv` (comma separated values) format, which can be opened in a plain text editor or in Excel. Open a new file in your text editor and save it as `contact_records.csv`. It should be saved in the same directory with your other `.py` example files.

Enter the following information in `contact_records.csv`

contact_records.csv

```
% Name, Instrument, Favorite Number
Robert, vocals, 32
John, drums, 27.2
Jimmy, guitar, -13.0
John Paul, bass, 9.2e7
```

example_4.py, version 1

```
# File:  example_4.py
# This Python script reads from contact_records.csv and
# writes to dummy_file.txt.
#
# How to Run:
# 1) Open a terminal/command window.
# 2) Change directories to the location where this file is
   saved.
# 3) Type the following command at the prompt:
#    python example_4.py

import csv

# Original/Testing Version:
with open('contact_records.csv', 'r') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',', quotechar=
   '|')
    for row in spamreader:
        print(row)
        print("\tThis row has %d columns." % (len(row)))
        print("\tThe first column in this row is:", row[0])
        print("\tThe first character of the first column in
   this row is:", row[0][0])
```

This first version of `example_4.py` can be improved. In particular:

- We want to store the information we've read into a dictionary, and
- We want to ignore the first/header line of the `.csv` file (which starts with the "%" character)

So, let's edit our Python script so it looks like this:

example_4.py, version 2, part 1

```
# File:  example_4.py
# This Python script reads from contact_records.csv and
# writes to dummy_file.txt.
#
# How to Run:
# 1) Open a terminal/command window.
# 2) Change directories to the location where this file is
   saved.
# 3) Type the following command at the prompt:
#    python example_4.py

import csv

myDictionary = {}

# A better version.
# Ignore rows/lines that start with the "%" character.
# Save information to our dictionary.
# NOTE:  We are assuming that the .csv file has a pre-
   specified format.
#   Column 0 -- Name (string)
#   Column 1 -- Instrument (string)
#   Column 2 -- Favorite Number (float)
with open('contact_records.csv', 'r') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',', quotechar=
   '|')
    for row in spamreader:
        # Only read rows that do NOT start with the "%"
   character.
        if (row[0][0] != '%'):
            name = str(row[0])
            instrument = str(row[1])
            number = float(row[2])
            myDictionary[name] = {'instrument': instrument, '
   favNumber': number}
```

Now, let's print out what we've saved in our dictionary:

example_4.py, version 2, part 2

```
# Let's see what we have in our dictionary:
for name in myDictionary:
    print(name)
    print(myDictionary[name])
```

The last task for this example is to write some information to a text file. This is useful if we want to store some information instead of simply printing it to the screen.

example_4.py, version 2, part 3

```
35 # Write to a file named dummy_file.txt
36 # DO NOT USE THE csv MODULE
37 print('\nWriting to dummy_file.txt...')
38 myfile = open('dummy_file.txt', 'w')    # 'w' means write. 'a'
       would be append.
39 myfile.write('Instrument, Name, Favorite Number Times 2 \n')
40 for name in myDictionary:
41     myString = "%s, %s, %f \n" % (myDictionary[name]['
       instrument'], name, myDictionary[name]['favNumber']*2)
42     myfile.write(str(myString))
43 myfile.close()
44 print('DONE')
```