Python Programming for Beginners

Module 5

Go over homework.

Review – ask for questions

Introduction to Lists:

Previously, we talked about four types of data: integer, float, string, Boolean. But imagine that you want to represent a lot of data, for example, the names of all the students in a class, or better yet, the names of all students in a school, or in a city or in a state. What if we wanted to represent the scores of a test in a class, or the grades of all students in the school? So far, my definition of a variable allows us to only represent a single piece of data in a variable. Therefore, if we wanted to represent all students, we would have to do something like this:

```
Student1 = 'Joe Schmoe'
Student2 = 'Sally Smith'
Student3 = 'Henry Jones'
Student4 = 'Betty Johnson'
```

As you might guess, this would become unmanageable very quickly. Every time we have a new student, we need a new variable to keep track of that person. And whatever we do with these variables, we have to modify code to add in the new variable everywhere.

Collections of Data

Think about other places where you have a lot of related data. Examples:

- High score leader board
- Bookmarked sites
- Purchases with a credit card
- Phone numbers
- Classes you are enrolled in
- Members of a team
- Teams in a tournament
- Names of clubs at a school

Lists

Rather than using individually named variables to represent these types of data, most programming languages allow you to represent a group of data like this using a single name. In Python this is called a "list". This is a new variable type. Let me give a formal definition:

DEF: List – an ordered collection of items that is referred to by a single variable name.

(In most other languages, the same concept is called an "array".)

Example, a shopping list:

'apples'
'bananas'
'carrots'
'dates'
'eggs'

In Python, a list can contain any type of data. For example, here is a list of test scores:

99
72
88
82
54

Lists can also be created using a mix of data types:

99
‘Joe Blow’
True
1234.56
‘Here is another string’
True
-123

To make things simple to start, we’ll build lists of strings.

Element

DEF: Element – a single member of a list. (Also known as an “item” in the list)

Given a list like:

‘apples
‘bananas’
‘carrots
‘dates’
‘eggs’

Each thing in the list is an element of the list.

Python Syntax for a List

Let’s see how this looks in Python. In Python, a list is assigned to a variable (just like another type of data). However, a list needs something to identify it as a list. For that, we use square brackets. A list is represented by an open square bracket, elements separated by commas, and a closing square bracket:

[<element>, <element>, ...<element>]

A list is typically created using an assignment statement.

<myListVariable> = [<element>, <element>, ...<element>]

Lists can have any number of elements. The number of possible elements is limited only by the amount of memory in the computer.

Here are some examples:

shoppingList = [‘apples’, ‘bananas, ‘carrots’, ‘dates’, ‘eggs’]

scoresList = [24, 33, 22, 45, 56, 33, 45]

mixedList = [True, 5, ‘some string’, 123.45, False]

(Try this in the Python shell or in a Python file)

add:

```
print(shoppingList)
print(scoresList)
print(mixedList)
```

Empty List

And, you can even have an empty list:

```
someList = []  # a list with no elements.

print(someList)
```

We’ll use this later by creating an empty list, then adding elements to it on the fly.

You can think of the empty list in relation to lists like zero in comparison to numbers.

Position of an Element in a List - Index

We've seen that we can create a list with the square bracket syntax, and we can print a list using the print statement, but the power of a list comes from the ability to use the individual elements in the list. So, we need a way to reference an individual element of a list.

You can think of any physical list (like a shopping list) as a numbered list. That is, we could assign a consecutive integer to each element, and reference any element in a list using that number. In fact, that number has a clear definition.

DEF index – the position (or number) of an element in a list. (Sometimes referred to as a 'subscript')

An index is always an integer value.

Looking at our shopping list, what would you think would be the index (element number) of carrots in the list?

The obvious answer would be 3. To us humans, in the above list, apples is element number 1, bananas would be obviously be element number 2, and carrots would be element number 3. To humans, this makes perfect sense.

Sample shoppingList:

Human Number	Element
1	'apples'
2	'bananas'
3	'carrots'
4	'dates'
5	'eggs'

BUT … in Python, and most other computer languages, the elements in a list are numbered starting at zero. All lists start at an index of zero. The indexes for the above list are 0, 1, 2, 3, and 4. 'apples' is element number zero. 'bananas' is element number 1, … 'eggs' is element 4. This is very important – and will cause you much grief!

Python Index	Element
0	'apples'
1	'bananas'
2	'carrots
3	'dates'
4	'eggs'

The list has 5 elements, but they are numbered 0 to 4.

Accessing an Element in a List

We now have a way of representing a list of data in a single variable (enclosing the list in brackets, separating elements by commas). But we need some way to get at the individual elements in the list.

The way we do that is to use this syntax:

<listVariable>[<index>]

Assume an assignment like this:

numbersList = [20, -34, 486, 3129]

Then we could access each element as follows:

```
numbersList[0]    # would evaluate to 20
numbersList[1]    # would evaluate to -34
numbersList[2]    # would evaluate to 486
numbersList[3]    # would evaluate to 3129
```

This gives the value of the element in the list at the given index.

Try this:

shoppingList = [‘apples’, ‘bananas’, ‘carrots’, ‘dates’, ‘eggs’]

print(shoppingList[2])
print(shoppingList[4])
print(scoresList[1])
print(mixedList[0])

How to Read Square Brackets

When you see something like this:

```
shoppingList[2]
```

You should read it as “shoppingList element 2” or “the element in position 2 of shoppingList” or “element 2 of shoppingList” or (oldschool) “shoppingList sub 2”

Using a Variable or Expression as an Index

The index can also be a variable or any expression. This will be very useful by the end of this module. For example:

```
myIndex = input(‘Enter an index: ‘)
myIndex = int(myIndex)
print(shoppingList[myIndex])

# Alternatively, combine two steps in one line
# and save the element in a variable

myIndex = input(‘Enter an index: ‘)
thisElement = shoppingList[int(myIndex)]
```

```
print(thisElement)
```

Enter a number between 0 and 4. It should work correctly.

Using a variable as an index into a list is extremely common, and I will give some examples in just a little bit.

Now try it again, and enter 100 as the index. What error do you get? The error tells you exactly what happened. You tried to access an element that is outside the valid range of the indices. There is no element 100, so you get an error. This is called 'range checking' and is automatic in Python. Most other languages do not do this.

So far, I've shown how to use a list with an index on the right hand side of an assignment statement or in a call to print. This is how you 'get' (or retrieve) a value from a list.

Changing a Value in a List

You can also 'set' a value in a list, that is, replace the current element in a list, by putting the list variable on the left hand side of an assignment like this:

```
print(shoppingList)
shoppingList[3] = 'cat food'
print(shoppingList)
```

This changes the value of an element at the given index to a new value. Python lists are "mutable" (changeable).

Negative Indexes

Here's a special thing you can do in Python. It is something I've never seen in any other language. What do you think this would do?:

```
print(shoppingList[-1])
```

Try it. Then try shoppingList[-2]

Reg. Index	Neg. Index	value
0	-5	'apples'
1	-4	'bananas'
2	-3	'carrots'
3	-2	'dates'
4	-1	'eggs'

A negative index means to count backwards from the end (or length) of the list. That is, take the number of elements in the list, and then add the negative index amount.

The most common use is to get the last element in a list using an index of -1.

Building a Simple MadLibs Game

Let me give you an example of how to use lists in a real program. We'll build an old favorite game, MadLibs. We'll start by getting input from the user like a real MadLibs game (this doesn't have anything to do with lists, but be patient).

This program is all about strings. Remember that when we want to put strings together it is called "concatenation". And the concatenation operator is the plus sign between strings. Just as we can add a long group of numbers, we can also concatenate multiple strings.

```
while True:
    name = input('Enter a name: ')
    verb = input('Enter a verb: ')
```

```
    adjective = input('Enter an adjective: ')
    noun = input('Enter a noun: ')


    sentence = name + ' ' + verb + ' down the hill hoping to escape the ' +
adjective + ' ' + noun + '.'
    print
    print(sentence)
    print

    goAgain = input('Return/Enter to continue, anything else to quit ')
    if goAgain != '':
       break
```

Adding a List to our MadLibs Program

Now, we'll change the program. Rather than having the user enter a name, we'll build a list of names like this:

nameList = ['Joe Schmoe', 'Madonna', 'President Obama', 'Neil Armstrong', 'The StayPuffed Marshmallow Man']

And we'll build some code to choose a random name from this list as the name in each MadLib.

Determining Number of Elements in a List – the len Function

However, the list of names could contain any number of names. Ideally, we would want to write code that would be able to work for any number of elements in the list. Therefore, we need a way to find how many elements are in a list.

For this, there is a built in function called 'len'.

len(<listVariable>)

To find the length of a list, that is, the number of elements in a list, you call the len function, and pass in the variable name of the list:

```
myList = ['abc', 'def', 'ghijk', 'mnop', 'qrs', 'tuv', 'x']
nElements = len(myList)
print(nElements)
```

Will print 7 (but remember: elements are numbered are 0 to 6)

Use randrange to Choose Random Element

Back to our list of names, let's make the code independent of the number of names. Obviously, no matter how many elements are in the list, we should use zero for the lower limit. For the upper limit, we'll use the number of items in the list – which we get by using the len function.

nameList = ['Joe Schmoe', 'Madonna', 'President Obama', 'Neil Armstrong', 'The Staypuffed Marshmallow Man']

```
nNames = len(nameList)     # in this case, sets nNames to 5
nameIndex = random.randrange(0, nNames)  # 0 to 4
randomName = nameList[nameIndex]
```

Again, our goal is to select a random number to use as an index to an element in the list. In this case, since we have 5 names in our list, when we call random.randrange, and pass in a value of 5, randrange would return a random integer of 0, 1, 2, 3, or 4 (up to but not including 5). Then use that index to pull out a random name.

[If there is time: Show how to modify program to build a single choose random from list using built-in function.]

In-class assignment:

Modify the program to include a list of verbs, a list of adjectives, and a list of nouns (similar to the names above). The program should randomly choose a name, verb, adjective, and noun and create and print a Madlib.

Accessing All Elements of a List - iteration

Now we have a way to access any element in a list. But we need a way to access ALL elements in a list. As a simple example, let's say that we just wanted to print out the value of all the elements of a list. We can say:

print(myList)

But what if we wanted to print one element per line? Or what if the list contains numbers, and we want to add them up. We need some way to get at all elements, but one at a time.

DEF Iterate – to traverse through, or visit all elements of a list.

To do this, we certainly have to have some type of loop like we have seen before. Here is one way to iterate through all elements using the 'while loop' that we already know. (Do not write this down, this is NOT the best way):

```
myList = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
nItems = len(myList)
myIndex = 0
while myIndex < nItems:
        print(myList[myIndex])
        myIndex = myIndex + 1
```

Run it – it works fine. But this seems a little clunky. Seems like you have to remember a lot of details, and get them all right, in order to make this loop work.

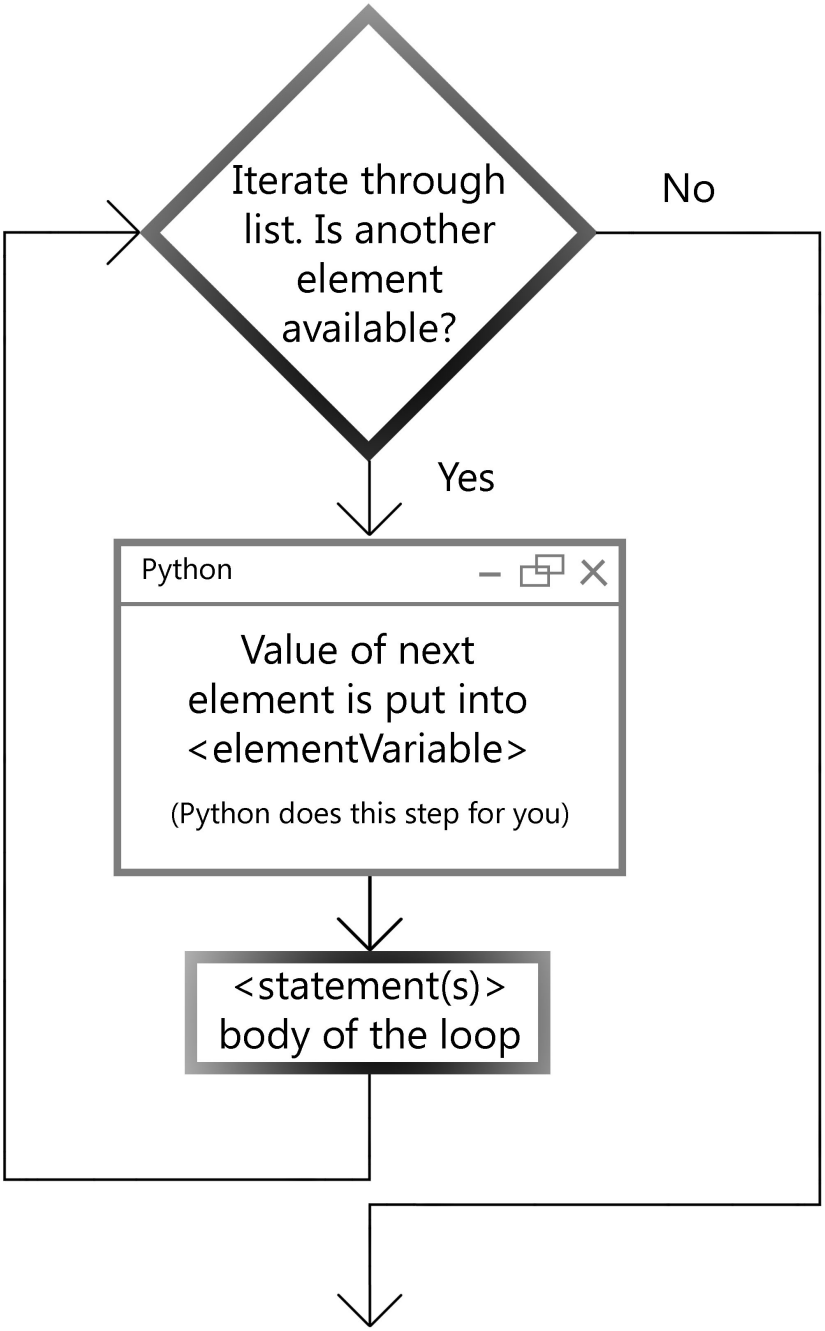
for Statement and for Loops

The people who designed Python came up with a better way to handle iterating through a list. As long-time programmers, they noticed that this pattern of looping and doing something with each element of a list; happens very often. So, they came up with an additional statement and loop, which gives you an extremely simple way to iterate through a list. It is called the for statement. Here is the generic form:

```
for <elementVariable> in <collection>:
        <indented block using  <elementVariable>  >
```

Notice the new keyword “in”. It makes the statement very readable. Think of this as saying “for each element in the list”.

Flowchart of for Loop:



For example, let's say you want to print out your shopping list with one element per line.

```
shoppingList = [‘apples’, ‘bananas’, ‘carrots’, ‘dates’, ‘eggs’]
for anItem in shoppingList:
        print(anItem)
```

A Simple Example of iteration:

```
friendsList = [‘Joe’, ‘Sue’, ‘Marcia’]
for friend in friendsList:
        print(‘Happy New Year ’ + friend)


numberList = [8, 22, 35, 6, 6, 22]
for number in numberList:
        print(‘The next number is’, number)
```

[if there is time]
In-class assignment1:

Write a program that uses a for loop to add up a list of numbers. For example, start with a list like this:

```
        numberList = [23,14,13, 10, 4, 32, 32, 22, 11]
```

Then use a ‘for’ loop to add them up.

Creating a Collection of Consecutive Numbers

There are times when you would like to have a consecutively ordered sequence of numbers. For example, if you are doing some math like adding up the numbers from 1 to n.

There is a built-in function in Python called range, that creates an ordered sequence of numbers, given a low end and a high end:

range(<low>, <upToButNotIncluding>)

The collection starts at <low>, and goes up to but not including the second value (high end).

For example, if you use 1 and 8, you will get a collection that starts at 1, and goes through 7 (does not include 8). This is identical to the way we specified a range of integers we made calls using random.randrange().

You can also call the range function with a single high value. In that case the function assumes that you want to use a low end of zero:

range(10)

Is exactly the same as:

range(0, 10)

These would both create a collection of numbers from 0 to 9.

The interesting use case for the range function is in a for statement. Remember that a for statement is designed to let you easily iterate through a collection – typically a list. Here is an example of how it could be used:

```
for number in range(0, 10):
    print(number)
```

The for statement iterates through the sequence, and every time through the loop, it assigns the next value to the variable "number". This code would print the numbers 0 to 9, each on a separate line.

In-class assignment 2:

Write a program that reads in an integer from the user, then uses a for loop and the range function to add up the numbers from 1 to the given number.

Looping a Given Number of Times

In addition to iterating through a list or sequence of numbers, we often want to go through a loop a given number of times. For example, we may want to play a game 3 times, or generate a given number of random results. The range function is ideal for these types of uses. When we want to run a loop n times, we can build a for loop this way:

```
for <someLoopVariable> in range(0, nTimes):
    # body of your loop.
```

In a loop like this, we may not even use the loop variable. We only care about running the loop a known number of times. For example, if we wanted to run our loop 5 times, we could code this:

```
for number in range(0, 5):
     # body of the loop
```

In this loop, the variable “number” will be given a different value in the sequence 0, 1, ... 4 each time through the loop. We might not even use the variable “number” inside our loop. It is just used to count the iterations.

Scientific Simulation

Remember the dice homework? Rather than doing a roll and asking if the user wants to roll again, let’s rewrite the program to simulate a large number of rolls. At the end, we’ll report the number and percentage of doubles.

First, let's do a little math to see what we would expect for an answer. Here is a chart of all possible rolls:

Die 1 \ Die 2	1	2	3	4	5	6
1	Doubles					
2		Doubles				
3			Doubles			
4				Doubles		
5					Doubles	
6						Doubles

Out of 36 possible rolls, 6 of them are doubles. So, we would expect 6/36 or 1/6 or 16% to be doubles.

Here's the code:

```
# Dice: count doubles in user-defined number of rounds ... repeated

import random

# simulate rolling a six-sided die and return its value
def rollOneDie():
    # generate a random number between 1 and 6
    thisFace = random.randrange(1, 7)
    return thisFace

while True:
    nDoubles = 0

    maxRounds = input('How many rounds do you want to do? (Or ENTER
to quit): ')
    if maxRounds == '':
        break
```

```
    maxRounds = int(maxRounds)

    for roundNumber in range(0, maxRounds):  # could use range(maxRounds)
        die1 = rollOneDie()
        die2 = rollOneDie()

        if die1 == die2:
            nDoubles = nDoubles + 1

    percent = (nDoubles * 100.0) / maxRounds
    print('Out of', maxRounds, 'you rolled', nDoubles, 'doubles, or',
percent, '%')

print('OK Bye')
```

Add try/except to ensure that the user has entered a valid integer for number of rounds:

```
    try:
        maxRounds = int(maxRounds)
    except:
        print('Please enter an integer')
        continue   # go back to the while statement
```

First, this is very quick. Even millions of runs only takes a few seconds. Second, this shows a very even distribution of random numbers.

List Manipulation

Think about a shopping list. There are many things that you might want to do to manipulate the elements in the list. In a list, you can also add elements and remove elements at any time.

For example ... you add things to this type of list before you go to the store. At the store, you might have Coke in your list, but you find that Pepsi is on sale, so you choose that instead. You see some cookies that you like, so you effectively add it to your list by putting it in your cart.

And if you are like me, you eliminate items from your list by crossing them off as you put them in your cart.

Python provides many built-in operations that allow you to manipulate and search through lists. Here are some of the most useful. The general syntax is:

<listVariable>.<operation>(<any argument(s)>)

Add an element:
<myList>.append(<value>)

Add an element at an index:
<myList.insert(<index>, <value>)

Delete the last element:
<myList>.pop()

Optionally, returns the value of that element:

<myVar> = <myList>.pop()

Delete an element at an index:

<myList>.pop(<index>)

Optionally, returns the value of that element

<myVar> = <myList>.pop(<index>)

Find out if a value is in a list:

<myBooleanVar> = <value> in <myList> # new keyword 'in'

Find out the index of a value in a list:
<myVar> = <myList>.index(<value>)

Find out how many times a value occurs in a list:

<myVar> = <myList>.count(<value>)

List Manipulation Example:

Demonstration of pizza topping program – an example program that manipulates a list.

Show: PizzaOrderFromMenu.py

Demo program, then show how it uses a number of these list manipulation calls.

DICTIONARY

In a list (or a string), we store information in a ordered way. We can retrieve information by using the index of the data (or character). However, there is another way to store information in Python called a dictionary. In a dictionary, the way you store information in a series of what are called key/value pairs. A dictionary looks like this:

{<key1>:<value1>, <key2>:<value2>, ... <keyN>: <valueN>}

This is the only place in Python where we use curly brackets.

The basic idea here is that a dictionary can be used to represent the attributes or properties of some physical object. Let's say we wanted to represent the following attributes of a car:

color – blue

style – hatchback
doors – 4
mileage – 35000
make – Toyota
model – Prius

We could create a dictionary using an assignment statement like this:

```
myCar = {'color':'blue', 'style':'hatchback', 'doors':4, 'mileage':35000, \
        'make':'Toyota', 'model':'Prius'}
```

This creates a car dictionary. We can check for the type using the type function:

```
print(type(myCar))     #  says it's a dict
```

We can print the whole dictionary with a print statement:

```
print(myCar)   # prints the dictionary as a dictionary, with braces
```

The important thing to know is that the keys in a dictionary are not ordered. But since you always use a key to access a value in a dictionary, it doesn't matter.

To access any piece of data using the key (rather than an index):

```
print(myCar['color'])  # prints blue

print(myCar['model'])  # prints Prius
```

Here is another example. Let's say that we wanted to represent the attributes of a house. For example, imagine that we wanted to describe a house that has the following attributes:

```
color = 'blue'
style = 'colonial'
numberOfBedrooms = 4
garage = True
```

```
burglarAlarm = False
streetNumber = 123
streetName = 'Any Street'
city = 'Anytown'
state = 'CA'
price = 625000
```

Rather than putting these values into a list (where each position has meaning), a dictionary allows us to use name/value pairs which make things much more clear:

```
houseDict = {'color' : 'blue', 'style' : 'colonial', 'numberOfBedrooms' : 4, \
        'garage' : True, 'BurglarAlarm' : False, 'streetNumber' : 123, \
        'streetName' : 'Any Street', 'City' : 'Anytown', 'state' : 'CA', \
        'price' : 625000}
```

We are naming this dictionary, houseDict, to make it clear that this is a dictionary. Again, this is not a requirement; we are using a name like this as an extension to our naming convention. In this example, all the keys of this dictionary are strings, and this is a very common practice. Each key in a dictionary must be unique. The values in a dictionary can be of any type.

Let's print out houseDict to show that Python understands the dictionary data structure:

```
print(houseDict)
{'color': 'blue', 'price': 625000, 'burglarAlarm': False,
'numberOfBedrooms': 4, 'streetName': 'Any Street', 'city': 'Anytown',
'style': 'colonial', 'garage': True, 'state': 'CA', 'streetNumber': 123}
```

In the output, notice that the key/value pairs are not necessarily in the same order in which they were entered. Python optimizes the arrangement of the keys so that it can access the data as fast as possible.

Dictionaries rely on the key/value pair relationships rather than on positioning. Therefore, when we want to access any piece of data in a

dictionary, we do it by using a key as an index, (rather than a position index that we use with a list). Here are some examples:

```
>>> print(houseDict['color'])
blue
>>> print(houseDict['state'])
CA
>>> print(houseDict['numberOfBedrooms'])
4
>>>
```

We can use the “in” operator to see if a key exists in a dictionary. For example, let’s see if the dictionary contains an entry for a ‘streetName’.

```
>>> print(houseDict)
{'color': 'blue', 'burglarAlarm': False, 'numberOfBathrooms': 2.5,
'numberOfBedrooms': 4, 'streetName': 'Any Street', 'city': 'Anytown',
'style': 'colonial', 'value': 625000, 'garage': True, 'state': 'CA',
'streetNumber': 123}
>>>
>>> print(houseDict['streetName'])
Any Street
>>>
```

Now let’s see what happens if we try to get the roofType.

```
>>> print(houseDict['roofType'])

Traceback (most recent call last):
  File "<pyshell#65>", line 1, in <module>
    print(houseDict['roofType'])
KeyError: 'roofType'
>>>
```

That says that roofType is not in the dictionary. We can check to see if a key in is a dictionary by using the “in” operator:

<key> in <dictionary>

It returns True if the key is found in the dictionary, or False if the key is not found.

```
>>> print('city' in houseDict)
True
>>> print('roofType' in houseDict)
False
>>>
```

In any code where we think that a key might not be found, it's a good idea to add some defensive coding to check and ensure that the key is in the dictionary before we attempt to use it on the dictionary. Typically, we build this type of check using an if statement:

```
if myKey in myDict:
    # OK, we can now successfully use myDict[myKey]
else:
    #  The key was not found, print some error message or take some
other action
```

Sometimes, it may make more logical sense to code the reverse test. We can use not in to test for the key not being in the dictionary:

```
if myKey not in myDict:
    # The key was not found, do whatever you need to do
```

[Show Population.py – shows use of reaching into a dictionary]

[Dictionary.py shows how to add to a dictionary]