Chapter 7: Data Structures

Python offers many data types and structures. Common data types in Python are integers, floats (decimal numbers), and strings (any sequence of characters contained in quotes). In addition, Python offers several data structures, including lists, tuples, sets, arrays (via numpy), and dictionaries. Chapter 7 will explore the complex numerical data type, as well as several data structure options.

A **data structure**, in a very broad sense, is an organized method for storing and managing data. A filing cabinet is a type of data structure, especially if it is alphabetized. Not only do data structures organize data, but they can also offer methods for retrieval, appending, removing, analyzing, manipulating, and accessing data.

7.1: Complex Numbers in Python

Python also offers the **complex** data type for numerical data. While this book does not focus on advanced or complex math, it is worth noting that the statement, **`import cmath`** will allow complex math method utilization. The following lines of code will illustrate the complex data type and will review the other common Python data types.

```
import cmath

z=cmath.sqrt(-9)
type(z)
Out[245]: complex

z
Out[246]: 3j

x=1
type(x)
Out[248]: int

y=3.46
type(y)
Out[250]: float

n="Hello !!"
type(n)
Out[252]: str
```

Recall that complex numbers (sometimes call imaginary numbers) are the result of the square root of -1, which in Python is called, **`j`**. The above statements show the use of importing the cmath library which allows for complex math. The following statements review data types, such

as *int*, *float*, and *str*. In addition to these common types, Python also offers several other data structure types, such as lists, tuples, dictionaries, sets, and arrays.

7.2: Mutable versus Immutable

An interesting difference between some types and classes in Python, is that some are **immutable** while others are not. An immutable object is one that cannot be changed in memory once it is created. In other words, once an immutable object is created and assigned to a variable, it can only be changed by deleting it in memory and creating a new object that reflects the desired change.

Alternatively, a **mutable** object is a **reference** to a memory location. If a second variable is assigned the value of the first, it is actually assigned the same reference in memory. This means that if either variable value is altered, the value in the reference is altered, and the change affects both variables.

In other words, mutable objects can be changed in their memory location. Immutable objects cannot. For this reason, immutable sequences (such as tuples discussed below) can be faster because they are not dynamic in nature. Mutable objects (such as lists) offer a greater number of method and options but do not offer the same time or space efficiency.

The best way to better understand mutable versus immutable is through example. Numeric types, such as integers, floats, and complex values, as well as strings and tuples are all immutable. If a copy of any of these objects is made, the new copy has its own memory reference and changes to the original or the copy do not affect the other.

Mutable (changeable) objects include lists, dictionaries, and sets. These objects are copied by reference and so changes to one copy will affect all copies. As a note, there are method options from the **import copy** library that offer true copies of mutable objects.

Example 7.2.1: Mutable versus Immutable

```
def main():

# b is a float, which is not mutable
# c is a string, which is not mutable
# d is a list, which IS mutable
# e is a tuple, which is not mutable

    b = 3.56
    c = "Fred"
    d = [1,3,5]
    e = (3,6,9)

    print("The data type for b = 3.56 is: ", type(b))
    print("The data type for c = \"Fred\" is: ", type(c))
    print("The data type for d = [1,3,5] is: ", type(d))
```

```
    print("The data type for e = (3,6,9) is: ", type(e))

    r=b
    s=c
    q=d
    w=e

    print("The value of r is: ", r)
    print("The value of s is: ", s)
    print("The value of q is: ", q)
    print("The value of w is: ", w)

    b = 3.56 + 1
    c = "Fred"+"Smith"
    d = d.append(7)
    e = (3,6,9,11)

    print("The value of r is: ", r)
    print("The value of s is: ", s)
    print("The value of q is: ", q)
    print("The value of w is: ", w)


main()
```

The output for the Program:

```
The data type for b = 3.56 is:  <class 'float'>
The data type for c = "Fred" is:  <class 'str'>
The data type for d = [1,3,5] is:  <class 'list'>
The data type for e = (3,6,9) is:  <class 'tuple'>

The value of r is:  3.56
The value of s is:  Fred
The value of q is:  [1, 3, 5]
The value of w is:  (3, 6, 9)

The value of r is:  3.56
The value of s is:  Fred
The value of q is:  [1, 3, 5, 7]
The value of w is:  (3, 6, 9)
```

In the program example above, four variables are created. Their types are printed as output and are float, string, list, and tuple. Recall that floats, strings, and tuples are immutable, but lists are mutable.

Next, four new variables are created and are set equal to the first four variables, so that:

```
r=b
s=c
q=d
w=e
```

Next, each of the original variables, b, c, d, and e, are changed. The question is whether the changes to b, c, d, and e will affect r, s, q, or w respectively.

The result shows that changes to b, c, and e do not affect r, s, and w, because these variables are immutable. However, the append change to d does affect the value of q because d is a list and lists are mutable. This is noted in bold in the output above.

There are options for creating true copies of mutable objects. Use of the **`import copy`** library is required as is the method, **`deepcopy()`.**

Example:

```
import copy


NewList1=[1,3,5]

NewList1
Out[71]: [1, 3, 5]

TrueCopy=copy.deepcopy(NewList1)

TrueCopy
Out[73]: [1, 3, 5]

MutableCopy=NewList1

MutableCopy
Out[75]: [1, 3, 5]

NewList1.append(7)

NewList1
Out[77]: [1, 3, 5, 7]

TrueCopy
Out[78]: [1, 3, 5]
```

```
MutableCopy
Out[79]: [1, 3, 5, 7]
```

In this example, a list is created called `NewList1`. Next, two new variables are created and set equal to `NewList1`. The first copy of `NewList1` is called `TrueCopy`, which is created using the `copy.deepcopy()` method. The second copy of `NewList1` is called `MutableCopy`.

When `NewList1` changed using the `append()` method, `TrueCopy` is not affected by the change (as it was created using the deepcopy method). However, `MutableCopy` is affected by the change.

Lists are mutable objects. If a list is copied without using the deepcopy method, then changes made in memory to the list will affect all copies of the list.

7.3: Lists in Python

A **list** is a data structure that is a **class type** in Python, and is standard. The general structure of a list is the following.

```
MyList = [element0, element1, element2, … , elementn]
```

A list can be a sequence of anything. It is possible to have a list of lists. It is possible to have a list that contains other data structures, such as tuples. It is possible to have list of some numbers and some words, etc. Like strings, lists can be indexed and are very versatile. There are several methods, functions, and rules for lists.

Example 7.3.1: Example of a list

```
mylist1=[1, 2.45, "ted", 3, "bill", "@!@!", (3,4)]

mylist1[0]
Out[38]: 1

mylist1[2]
Out[40]: 'ted'

type(mylist1)
Out[41]: list

type(mylist1[1])
Out[42]: float
```

To create a list, use the square brackets, [ ]. As shown above, lists may contains any type of information. Each element of a list can be accessed using the name of the list followed by square brackets containing the **index** of the location of the element of interest. All list indices start at "0".

In the example above, the list is called `mylist1`. The list contains seven elements. The first and third elements are integers.

```
mylist[0] = 1
mylist[2] = 3
```

The second element is a float.

```
mylist1[1] = 2.45
type(mylist[1]) = float
```

The third, fifth, and sixth elements are strings.

The last element is a tuple.

```
mylist1[-1]
Out[41]: (3,4)

type(mylist1[-1])
Out[42]: tuple

mylist1[6]
Out[43]: (3,4)
```

Lists are very powerful and very versatile. Like strings, lists can be indexed. The first element in a list starts at index 0. The last element of a list can be indexed with -1. Table 7.3.1 illustrates the indexing of the list, `mylist1=[1, 2.45, "ted", 3, "bill", "@!@!", (3,4)]`

Table 7.3.1: Indexing a List

element	1	2.45	"ted"	3	"bill"	"@!@!"	(3,4)
forward index	0	1	2	3	4	5	6
reverse index	-7	-6	-5	-4	-3	-2	-1

The colon can also be used for indexing a range within a list. The range does not include the element of the last index and so will not include mylist1[5].

```
mylist1[2:5]
```

```
Out[86]: ['ted', 3, 'bill']
```

Range Indexing Syntax

```
NewList = MyList[a: b]
```

The name of the list is MyList. The first element in NewList will be MyList[a]. The last element in NewList wil be MyList[b-1].

7.3.1: List Methods

Lists offer a set of methods to make their utilization more effective.

The append Method

The append method will append or include onto the end of the list any element. The syntax for append is:

```
MyList.append(newElement)
```

Example:

```
MyList=[2.2, "bob", [4,5], (1,2,3), "hello"]

MyList
Out[93]: [2.2, 'bob', [4, 5], (1, 2, 3), 'hello']

MyList.append("There!")

MyList
Out[95]: [2.2, 'bob', [4, 5], (1, 2, 3), 'hello', 'There!']
```

The extend Method

The extend method will extend a list with the elements from another list. The syntax for extend is the following:

```
MyList.extend(<list>)
```

Example:

```
MyList
Out[95]: [1, 2.2, 'bob', [4, 5], (1, 2, 3), 'hello',
'There!']

NewList=[1, "happy", 4.55]
```

```
MyList.extend(NewList)

MyList
Out[98]: [1, 2.2, 'bob', [4, 5], (1, 2, 3), 'hello',
'There!', 1, 'happy', 4.55]
```

Note that each element in `NewList` was appended on individually to `MyList`. The key difference between append and extend is that if an element (such as a list, tuple, string, number, etc. is appended to a list, it will be added to the list while still retaining its structure. Alternatively, if extend is used, each element in the newlist with be individually appended and will not retain original structure.

Example:

```
List1=[1, 2.2, "bob"]

List2=[5.5,"hello"]


List1.append(List2)

List1
Out[105]: [1, 2.2, 'bob', [5.5, 'hello']]


List1=[1, 2.2, "bob"]

List1.extend(List2)

List1
Out[108]: [1, 2.2, 'bob', 5.5, 'hello']
```

Above, when append is used, List 2 retains its list structure and is added on to List 1 as a list itself. However, when extend is used, the elements of List 2 are separated and appended individually onto List1. They do not remain in their own sub-list.

The count Method

The count method will give a count of the number of elements in a list. The syntax for the count method is:

```
MyList.count(<element>)
```

Example:

```
MyList=[1, 0, "bob", 1, 0, 0, [3,4,5], (6, 8.7), "bob"]

MyList
Out[114]: [1, 0, 'bob', 1, 0, 0, [3, 4, 5], (6, 8.7),
'bob']

MyList.count("bob")
Out[115]: 2

MyList.count(0)
Out[116]: 3

MyList.count((6, 8.7))
Out[117]: 1
```

The insert, remove, and pop methods

In the next example set, the list methods, insert, remove, and pop, will be illustrated. The **insert** method allows for the insertion of an element into a list at any location. The insert method requires two parameters, the location in the list for the insertion (given that the list starts at an index of 0), and the element to be inserted. Any element can be inserted, including a string, a number, a list, a tuple, etc.

Syntax for the insert method:

```
MyList.insert(index location, element to insert)
```

The **remove** method allows for the first occurrence of any element of a list to be removed. The method requires one parameter, namely the element to be removed.

Syntax for the remove method:

```
MyList.remove(element)
```

The **pop** method allows for the last element of any list to be removed (popped out).

Syntax for the pop method:

```
RemovedElement = MyList.pop()
```

Example 7.3.2: Illustration of insert, remove, and pop list methods

```
AnotherList=["Harry", 7, 7, (8, 9, 10), 4.45, "!!Wow!!"]
```

```
AnotherList
Out[85]: ['Harry', 7, 7, (8, 9, 10), 4.45, '!!Wow!!']

AnotherList.insert(1,"Potter")

AnotherList
Out[87]: ['Harry', 'Potter', 7, 7, (8, 9, 10), 4.45, '!!Wow!!']

AnotherList.remove(7)

AnotherList
Out[89]: ['Harry', 'Potter', 7, (8, 9, 10), 4.45, '!!Wow!!']

AnotherList.pop()
Out[92]: '!!Wow!!'

AnotherList
Out[93]: ['Harry', 'Potter', 7, (8, 9, 10), 4.45]

AnotherList.remove((8,9,10))

AnotherList
Out[95]: ['Harry', 'Potter', 7, 4.45]

AnotherList.insert(2, "Goblet")
AnotherList
Out[97]: ['Harry', 'Potter', 'Goblet', 7, 4.45]
```

The example above starts with a new list called, `AnotherList`, which contains the string “Harry”, the integer 7, another integer 7, the tuple (8,9,10), the float 4.45, and the string "!!Wow!!".

The statement, `AnotherList.insert(1,"Potter"),` inserts the string "Potter" at index 1. The statement, `AnotherList.remove(7)`, locates and removes the first occurrence of the integer, 7, from the list. The statement, `AnotherList.pop(),` removes (pops off the end) the last element of the list and returns it. In this case, it returns, *'!!Wow!!'*.

The sort and reverse methods

The next two methods for lists are **sort** and **reverse.** The sort method performs an in place sorting.

The syntax for the sort method is:

```
MyList.sort()   #Sort in ascending order
MyList.sort(reverse=True)   #Sort in descending order
```

Sorting numbers works well. However, for lists composed of a variety of types, the sort method may result in an error or unexpected results.

Sorting Examples:

```
List1=[3,7,1,9,10,5]

List1.sort()

List1
Out[112]: [1, 3, 5, 7, 9, 10]

List1.sort(reverse=True)

List1
Out[114]: [10, 9, 7, 5, 3, 1]

List2=["a", "y", "r", "e", "w"]

List2.sort()

List2
Out[117]: ['a', 'e', 'r', 'w', 'y']

List3=["bob", "zebra", "apple", "fiddle"]

List3.sort(reverse=True)

List3
Out[120]: ['zebra', 'fiddle', 'bob', 'apple']

List4 = ["bob", "a", "Happy", "z!", "wow??"]

List4.sort()

List4
Out[125]: ['Happy', 'a', 'bob', 'wow??', 'z!']


list5=["!", "?", "%", "$", "#", "@", "*", "&", "^"]

list5.sort()

list5
Out[140]: ['!', '#', '$', '%', '&', '*', '?', '@', '^']

List6 = ["bob", 1, 4.56, "!!wow!!"]

List6.sort()

TypeError: unorderable types: int() < str()
```

In the above examples, numbers can be sorted, and any collection of strings can be sorted. However, when types are mixed, the sorting can cause an error. Strings (including single characters) are sorted using the first character in the string. Capital letters sort above (before) lower case letters, and symbols sort according to their ASCII order.

The reverse Method:

The reverse method is very straight forward and works for all lists including those with various element types. The reverse method reverses the order of the elements in the list.

The syntax for the reverse method is:

```
MyList.reverse()
```

Example:

```
List5 = ["bob", "a", "Happy", "z!", "wow??", "!H!", 3.45]

List5.reverse()

List5
Out[147]: [3.45, '!H!', 'wow??', 'z!', 'Happy', 'a', 'bob']
```

Concatenation and Indexing

Like strings, lists can also be indexed and concatenated.

The syntax for list indexing is:

```
MyList=[element1, element2, …, elementN]
MyList[0]=element1
MyList[1]=element2
MyList[-1]=MyList[N-1]=element
MyList[a:b] = [element[a], …, element[b-1]]
```

The syntax for list concatenation is:

```
MyList1 + MyList2
```

Example:

```
MyList1=[1,2,3]

MyList2=["bob", "fred", (2,5)]
```

```
MyList1+MyList2
Out[133]: [1, 2, 3, 'bob', 'fred', (2, 5)]
```

The colon operator for lists:

The colon ":" operator can be used to access portions of a list.

The syntax for the list colon operator is:

```
myList[a:b:c]
```

The "a" is the starting index, "b" is the ending index (which is not included in the final list), and "c" is the step (the separation between each element). Note that the new list will always include values up *until* "b" (the ending index) but not including "b". It is optional to omit a, b, or c.

Examples:

```
myList
Out[190]: [10, 20, 30, 40, 50, 60, 70, 80]

myList[0:5:1]
Out[191]: [10, 20, 30, 40, 50]

myList[0:5:2]
Out[192]: [10, 30, 50]

myList[0:5:3]
Out[193]: [10, 40]

myList[:]
Out[194]: [10, 20, 30, 40, 50, 60, 70, 80]

myList[1:3]
Out[195]: [20, 30]

myList[2:]
Out[196]: [30, 40, 50, 60, 70, 80]

myList[:2]
Out[197]: [10, 20]
```

The statement, `myList[0:5:1]`, specifies to start at index "0" (which has the value 10 in myList), to end at, but not include index 5 (so the last value is myList[4] which is 50), with a step of "1" (which means do not skip any indices).

The statement, `myList[0:5:2]` returns the list [10, 30, 50] as it starts at index "0", ends before index "5" and the step is "2", which means "every other value". Similarly, the statement: `myList[0:5:3]`, returns the list, [10, 40]. It starts at index 0, ends before index 5, and the step is 3 (every third value).

Consider the statement, `myList[:2]`. Here, the starting index is blank and so it defaults to "0". The ending index is 2 and so the new list ends before index 2. The step is omitted and so the default is 1. The result is: [10, 20].

Similarly, the statement, `myList[2:]`, has the starting index at 2. It omits the ending index and so the default is the entire remainder of list. The step is also omitted and so the default is 1. The result is: [30, 40, 50, 60, 70, 80].

As a final note, recall that lists are copied by reference, not by copy. List are mutable objects. As such, to make another true copy of a list so that changes to one list do not affect changes to the other, the **import copy** is required as is the method `copy.deepcopy(<listname>)`.

Example: Review of mutable lists and the deepcopy method.

```
import copy

a = [1,2,3]
c= a

c
Out[214]: [1, 2, 3]

a.append(4)
a
Out[216]: [1, 2, 3, 4]

c
Out[217]: [1, 2, 3, 4]

## Notice when a is updated, c is automatically updated.

b=copy.deepcopy(a)
b
Out[219]: [1, 2, 3, 4]

a.append(5)
a
Out[221]: [1, 2, 3, 4, 5]

b
Out[222]: [1, 2, 3, 4]
# Updates to a do not affect b as it was made via deepcopy
```

```
c
Out[223]: [1, 2, 3, 4, 5]
```

In the example above, `c = a`. Therefore, `c` updated automatically when `a` is updated (because they both reference the same location in memory). However, using the `import copy` and the `copy.deepcopy()` method, the list `b` becomes a true copy that is not connected to `a` or `c`.

Table 7.3.2 offers a non-exhaustive collection of common list methods.

list.append(x) Add an object to the end of the list. list.extend(<list>) Extend the list by appending all the items in the given list. list.insert(index, objectname) Insert an object at a given location (index). list.remove(objectname) Remove the first object in the list whose value is the object. list.pop([index]) Remove the object from the given location (index) of the list, and return it. list.clear() Remove all objects from the list. Equivalent to del MyList[:]. list.index(objectname) Return the index in the list of the first whose value is objectname. list.count(objectname) Return the number of times objectname appears in the list. list.sort() Sort the objects in the list in place. This is possible as lists are mutable. list.reverse() Reverse the objects in the list in place. list.copy() Make a new copy of the list that does not reference the original

Table 7.3.2: Common List Methods

7.4: Tuples and Dictionaries

7.4.1: Tuples

A **tuple** is data structure type in Python. Like a list, a tuple is also a sequence of data. Tuples (like strings, but unlike lists), are **immutable**, which is one of the key difference between tuples and lists. A tuple is a sequence of values, each separated by a comma, and contained in parentheses. Tuple elements can be of any type, including other tuples. Like lists and strings, tuples can be indexed and can use the colon operator.

The syntax for a tuple is:

```
MyTuple = (element1, element2, …, elementN)
```

.

Example:

```
NewTuple=(1, 'bob', 5.67, 3.141592653589793, (3, 4, 5))

NewTuple
Out[313]: (1, 'bob', 5.67, 3.141592653589793, (3, 4, 5))

NewTuple[3]
Out[314]: 3.141592653589793

NewTuple[4]
Out[315]: (3, 4, 5)

NewTuple[2:4]
Out[316]: (5.67, 3.141592653589793)
```

Because the key difference between a tuple and a list is the immutability of a tuple, tuples are faster, but not as effective if changes must be made to the sequence. Therefore, if speed is critical and the sequence will not require changes, a tuple is a better option. Tuples do not offer many of the methods that lists offer, such as append, remove, sort, reverse, etc., because tuples are immutable. However, if changes to the elements are likely, such as additions and deletions, a list is a more versatile option.

7.4.2: Dictionaries

A **dictionary** is an unordered sequence structure in Python that is indexed using “keys”. Dictionaries use the curly braces, { }, and contain key:value pairs separated by commas. The order of the elements in a dictionary does not matter as elements are indexed via key and not location.

The syntax of a dictionary:

```
MyDict={Key1:Element1, Key2:Element2, Key3:Element3, …}
```

Example:

```
MyDict1={"Firstname":"Bob", "Lastname":"Smith", "Age":28}

MyDict1
Out[155]: {'Age': 28, 'Firstname': 'Bob', 'Lastname':
'Smith'}

MyDict1["Age"]
Out[158]: 28

MyDict1["Firstname"]+ " " + MyDict1["Lastname"]
Out[160]: 'Bob Smith'

MyDict2={1:"One", 2:"Two", "GO":"Three"}

MyDict2[2]
Out[162]: 'Two'

MyDict2["GO"]
Out[163]: 'Three'

MyDict3={0:0, 1:1, 10:2, 11:3, 100:4, 101:5}

MyDict3[10]
Out[166]: 2
```

Recall that lists (mutable) and tuples (immutable) are sequences of elements that can be indexed by numbers, or ranges of numbers. For example, suppose MyList=[1,3,5,7], then MyList[1] is the value 3, and MyList[1:3] is [3,5].

A dictionary is also a sequences of values (just like lists, tuples, and strings), but it has a different structure and indexing method. Specifically, a dictionary can be thought of as an **unordered collection** of ***key:value*** **pairs**, where the ***key*** acts as the index or locator for the ***value*** it is associated with.

The *key* itself can be any immutable (unchangeable) type, such as integers or strings. It is also possible to use a tuple as a key (because tuple are immutable), but only under the circumstance that the tuple contains only immutable elements, such as number types, strings, or other tuples.

To reiterate and further solidify the difference between mutable and immutable, it is worth noting that keys in dictionaries cannot be lists because lists can be updated or modified in place in memory (this is the definition of mutable). However, it would be non-functional for a key to

be modifiable in memory, because if it were, it may no longer associate with the correct value. This concept is very much like owning a key to your front door. It would not be a good idea to make this key alterable by any other method. If the key were changed, it would no longer open the door.

Dictionary Alternatives

Alternative 1: Using the dict function

Dictionaries can also be built using a couple of other methods. The first is a sequence of ***key=name*** pairs and the **`dict()`** function, where the keys are strings.

The syntax for the dict function:

```
MyDict = dict(<string>=element, <string>=element, …)
```

Example:

```
NewDict4=dict(firstname="Freddy", studentID="G123456",
age=21)

NewDict4
Out[349]: {'age': 21, 'firstname': 'Freddy', 'studentID':
'G123456'}

NewDict4["age"]
Out[350]: 21
```

Alternative 2: Using a List of (key,value) pairs and the dict function

The next option is using a list of *(key, value)* pairs, where the keys can be numbers or strings.

The syntax is:

```
MyDict = dict( [ (key, value), (key2, value2) , …] )
```

For example:

```
NewDict5=dict([ ('firsname', "Fred"), ('ID', 3333), (3,
"last")])

NewDict5
Out[352]: {'firsname': 'Fred', 3: 'last', 'ID': 3333}

NewDict5[3]
```

```
Out[353]: 'last'
```

Dictionaries are very versatile data structures. To get an idea of what they can do and how they work, review the following example. Note also that this example reviews a number of other concepts covered in the book. It is recommended that you type in, save, and run this example. Review each line of code to be sure it is clear. Next, make alterations to the program and evaluate the results.

Example 7.4.1:

```
# DictExample1.py
# by Ami Gates

def main():

    #------An empty Dictionary
    EmpDict={"firstname":"empty", "lastname":"empty", "age":"empty"}
    NewDict=CreateDict(EmpDict)
    #------Get info and put it into dict
    Info=input("What information do you need from the dictionary?
(firstname, lastname, or age):")
    GetInfoFromDict(Info,NewDict)
    #-----Sort the dicct keys option
    answer=input("Would you like to see a sorted list of the keys in
the dict?: ")
    if answer[0]=="Y" or answer[0]=="y":
        sortedkeys=SortedDictKeys(NewDict)
        print(sortedkeys)
    #------Remove an element option
    answer2=input("Would you like to remove an item from the dict?: ")
    if answer2[0]=="Y" or answer2[0]=="y":
        whichitem=input("Which key would you like removed? (firstname,
lastname, or age):")
        del NewDict[whichitem]
    #-----Add an element option
    answer2=input("Would you like to add an item to the dict?: ")
    if answer2[0]=="Y" or answer2[0]=="y":
        keyname=input("What is the key name?:")
        value=input("What is the value?:")
        NewDict=AddDictItem(NewDict,keyname,value)
    #----print out the finalized dict
    print(NewDict)
###----END OF MAIN---###

### This function populations the dict
def CreateDict(EmpDict):
    #print(EmpDict)
    for k, v in EmpDict.items():
        #The items() method will place the key into k and value into v
here
        print("Please enter your ",k, ":")
        nextdata=input()
        EmpDict[k]=nextdata
```

```
    #print(EmpDict)
    return EmpDict
#--------------------------------------
### This function adds key (k) value (v) pair to the dict
def AddDictItem(dic,k,v):
    new={k:v}
    dic.update(new)
    return dic
#--------------------------------------
## This function gets info from the dict based on a key (input1)
def GetInfoFromDict(input1,dict1):
    print(dict1[input1])
#--------------------------------------
## This function sorts and returns all dict key names
def SortedDictKeys(dict3):
    getsortedlist=sorted((dict3.keys()))
    return getsortedlist
#----------------------------------------
## Calls main to start
main()
```

The following is a possible input/output for the program above:

```
Please enter your  firstname :

John
Please enter your  lastname :

Smith
Please enter your  age :

33

What information do you need from the dictionary? (firstname, lastname,
or age):lastname
Smith

Would you like to see a sorted list of the keys in the dict?: y
['age', 'firstname', 'lastname']

Would you like to remove an item from the dict?: yes

Which key would you like removed? (firstname, lastname, or age):age

Would you like to add an item to the dict?: Yes

What is the key name?:ID

What is the value?:G23456712
{'firstname': 'John', 'lastname': 'Smith', 'ID': 'G23456712'}
```

As you review the program and the output above, make note of the function calls, the use of decision structures, and the methods that can be used with dictionaries. In the next section, the data structure object, **set**, will be discussed.

7. 5: Sets and Frozen Sets

In the mathematical sense, a **set** is a collection or grouping of objects or elements, with no repeats. Sets are mutable, but they can only contain immutable objects (such as numbers, strings, tuples, or frozen sets).

The syntax for sets:

```
MySet={element, element, …}
```

In Python, a set is defined as a collection of unordered and immutable objects. Therefore, one can have a set of numbers or a set of strings, but not a set of lists. Unlike lists, tuples, and dictionaries, sets are not designed to allow indexed access to individual elements. Instead, sets allow for the use of methods, such as intersection, union, length, and pop (remove an element).

Sets in the Python work very much like mathematical sets. The following program example will illustrate the creation and manipulation of sets. A **frozen set** is a set that cannot be changed; it is immutable.

Example 7.5.1:

```
# SetExample.py
# by Ami Gates

def main():

    set1={1,2,3,4,1,4,3,5,1}
    print("set1 is: ",set1)
    print("The length of set1 is ",len(set1))

    set2={1,2,3}
    print("set2 is: ",set2)
    print("The length of set2 is ",len(set2))

    set3={"Fran", "Bob", "Albert"}
    print("set3 is: ",set3)
    print("The length of set3 is ",len(set3))

    print("The intersection between set1 and set2 is ", set1 &
set2)
    print("The union between set1 and set2 is ", set1 | set2)
    print("Is set2 a subset of set1? ", set2 < set1)

    print("Add element 10 to set1 using set1.add(10) to give: ")
    set1.add(10)
    print(set1)
```

```
    print("Remove element 1 from set2 using set2.remove(1) to
give: ")
    set2.remove(1)
    print(set2)

    for i in set1:
        print(i+5)

## END OF MAIN

main()
```

The output for the above program is:

```
set1 is:  {1, 2, 3, 4, 5}
The length of set1 is  5
set2 is:  {1, 2, 3}
The length of set2 is  3
set3 is:  {'Bob', 'Fran', 'Albert'}
The length of set3 is  3
The intersection between set1 and set2 is  {1, 2, 3}
The union between set1 and set2 is  {1, 2, 3, 4, 5}
Is set2 a subset of set1?  True
Add element 10 to set1 using set1.add(10) to give:
{1, 2, 3, 4, 5, 10}
Remove element 1 from set2 using set2.remove(1) to give:
{2, 3}
6
7
8
9
10
15
```

Notice that "&" is the operator for **set intersection** and "|" is the operator for **set union**. Similarly, "<" is the operator for **"is subset of"** and the result is True or False. Elements in a set can be added or removed.

The key difference between a set and a list is that sets do not have duplicated elements (any duplicates are removed automatically and are not counted in the length) and sets are not and cannot be indexed. It would not be feasible to have MySet={1,2,3} and then try to access MySet[1] to get the value 2. Because sets are unordered and do not allow repeats, the elements in a set are not index able.

Alternatively, the following list: MyList=[1,2,3] is ordered and does allow indexing: MyList[1] is the value 2. Sets are defined with { } and lists are defined with [ ].

7.6: Review and Comparison: Strings, Lists, Tuples, Dictionaries, and Sets

Strings, lists, tuples, dictionaries, and sets are all data structure objects in Python that organize and assist in the manipulation of data. The string class is designed specifically for ordered sequences of characters (including symbols and numbers). Any word or phrase contained in quotes is a string. There are several methods and operations available for strings, including indexing and concatenation. Strings are immutable. This means that once a string is created, it cannot be changed. Copies of strings do not share memory locations, and changes to strings result in the creation of a new string in memory. While a string can be any sequence of characters (including numbers and symbols), strings cannot contain other objects (such as tuples or lists).

The list class is also an ordered sequence, but unlike strings, lists can be a sequence of any objects. So a list can contain a set, or a tuple, or a string, or all three. Lists have many built-in methods and are versatile and flexible. Recall that lists are mutable, which means that if a list is created, it can be changed, updated, reduced, etc., in place in memory (it is dynamic). In addition, if another variable is set equal to an existing list, changes to the list affect all variables that reference it. This can be mitigated by creating a deepcopy.

Tuples are very similar to lists in that they are ordered sequences of anything. However, unlike lists, tuples are immutable (not dynamic in memory). This makes them faster to operate when speed is critical. However, it makes them a poor choice for sequences that are likely to change. Tuples, lists, and strings can all be indexed and can utilize splicing methods, as they are ordered.

Sets and dictionaries are unordered. A dictionary is a collection of key:value pairs where the value is indexed by the name of its associated key. Dictionaries offer many methods for updating, organizing, and managing unordered data. Sets are a collection of immutable types (such as numbers or strings). Sets are not only unordered, but they also cannot contain duplicate values. Sets in Python are very similar to common mathematical sets, as are the methods that work on sets, such as union and intersection.

A Note about Arrays and NumPy:

NumPy is a package for scientific computing in Python (http://www.numpy.org/). The key data structure in NumPy is an N-dim array. While arrays and NumPy will not be discussed in this chapter, we will return to packages and arrays for Python in a later chapter.