Data Types and Referencing

Programming languages and applications need data. We define applications to work with data, and we need to have containers that can be used to hold it. This chapter is all about defining containers and using them to work with application data. Whether the data we are using is coming from a keyboard entry or if we are working with a database, there needs to be a way to temporarily store it in our programs so that it can be manipulated and used. Once we're done working with the data then these temporary containers can be destroyed in order to make room for new constructs.

We'll start by taking a look at the different data types that are offered by the Python language, and then we'll follow by discussing how to use that data once it has been collected and stored. We will compare and contrast the different types of structures that we have in our arsenal, and we'll give some examples of which structures to use for working with different types of data. There are a multitude of tasks that can be accomplished through the use of lists, dictionaries, and tuples and we will try to cover many of them. Once you learn how to define and use these structures, then we'll talk a bit about what happens to them once they are no longer needed by our application.

Let's begin our journey into exploring data types and structures within the Python programming language. . .these are skills that you will use in each and every practical Jython program.

Python Data Types

As we've discussed, there is a need to store and manipulate data within programs. In order to do so then we must also have the ability to create containers used to hold that data so that the program can use it. The language needs to know how to handle data once it is stored, and we can do that by assigning data type to our containers in Java. However, in Python it is not a requirement to do so because the interpreter is able to determine which type of data we are storing in a dynamic fashion.

Table 2-1 lists each data type and gives a brief description of the characteristics that define each of them.

Table 2-1. Python Data Types

Data Type	Characteristics
None	NULL value object
int	Plain integer (e.g., 32)
long	Long integer. Integer literal with an 'L' suffix, too long to be a plain integer
float	Floating-point number. Numeric literal containing decimal or exponent sign
complex	Complex number. Expressed as a sum of a numeric literal with a real and imaginary

	part
Boolean	True or False value (also characterized as numeric values of 1 and 0 respectively)
Sequence	Includes the following types: string, unicode string, basestring, list, tuple
Mapping	Includes the dictionary type
Set	Unordered collection of distinct objects; includes the following types: set, frozenset
File	Used to make use of file system objects
Iterator	Allows for iteration over a container. See section on Iterators for more details

Given all of that information and the example above, we should officially discuss how to declare a variable in the Python language. Let's take a look at some examples of defining variables in the following lines of code.

Listing 2-1. Defining Variables in Jython

```
# Defining a String

x = 'Hello World'

x = "Hello World Two"

#  Defining an integer

y = 10

#  Float

z = 8.75

# Complex

i = 1 + 8.07j
```

An important point to note is that there really are no types in Jython. Every object is an instance of a class. In order to find the type of an object, simply use the type() function.

Listing 2-2.

```
# Return the type of an object using the type function

>>> i = 1 + 8.07j

>>> type(i)

<type 'complex'>

>>> a = 'Hello'

>>> type(a)

<type 'str'>
```

A nice feature to note is multiple assignment. Quite often it is necessary to assign a number of values to different variables. Using multiple assignment in Python, it is possible to do this in one line.

Listing 2-3. Multiple Assignment

```
>>> x, y, z = 1, 2, 3
>>> print x
1
>>> print z
3
>>>
```

Strings and String Methods

Strings are a special type within most programming languages because they are often used to manipulate data. A string in Python is a sequence of characters, which is immutable. An immutable object is one that cannot be changed after it is created. The opposite would be a mutable object, which can be altered after creation. This is very important to know as it has a large impact on the overall understanding of strings. However, there are quite a few string methods that can be used to manipulate the contents of a particular string. We never actually manipulate the contents though, these methods return a manipulated copy of the string. The original string is left unchanged.

Prior to the release of Jython 2.5.0, CPython and Jython treated strings a bit differently. There are two types of string objects in CPython, these are known as Standard strings and Unicode strings. There is a lot of documentation available that specifically focuses on the differences between the two types of strings, this reference will only cover the basics. It is worth noting that Python contains an abstract string type known as basestring so that it is possible to check any type of string to ensure that it is a string instance.

Prior to the release of Jython 2.5.0 there was only one string type. The string type in Jython supported full two-byte Unicode characters and all functions contained in the string module were Unicode-aware. If the u'' string modifier is specified, it is ignored by Jython. Since the release of 2.5.0, strings in Jython are treated just like those in CPython, so the same rules will apply to both implementations. If you are interested in learning more about String encoding, there are many great references available on the topic. It is also worth noting that Jython uses character methods from the Java platform. Therefore properties such as isupper and islower, which we will discuss later in the section, are based upon the Java methods, although they actually work the same way as their CPython counterparts

In the remainder of this section, we will go through each of the many string functions that are at our disposal. These functions will work on both Standard and Unicode strings. As with many of the other features in Python and other programming languages, at times there is more than one way to accomplish a task. In the case of strings and string manipulation, this holds true. However, you will find that in most cases, although there are more than one way to do things, Python experts have added functions which allow us to achieve better performing and easier to read code.

Table 2-2 lists all of the string methods that have been built into the Python language as of the 2.5 release. Because Python is an evolving language, this list is sure to change in future releases. Most often, additions to the language will be made, or existing features are enhanced. Following the table, we will give numerous examples of the methods and how they are used. Although we cannot provide an example of how each of these methods work (that would be a book in itself), they all function in the same manner so it should be rather easy to pick up.

Table 2-2. String Methods

Method	Description of Functionality
capitalize()	Returns a capitalized copy of string
center (width[,fill])	Returns a repositioned string with specified width and provide optional padding filler character
count(sub[,start[,end]])	Count the number of distinct times the substring occurs within the string
decode([encoding[,errors]])	Decodes and returns Unicode string
encode([encoding[,errors]])	Returns an encoded version of a string
endswith(suffix[,start[,end]])	Returns a boolean to state whether the string ends in a given pattern
expandtabs([tabsize])	Converts tabs within a string into spaces
find(sub[,start[,end]])	Returns the index of the position where the first occurrence of the given substring begins
index(sub[,start[,end])	Returns the index of the position where the first occurrence of the given substring begins. Raises a ValueError with the substring is not found.
isalnum()	Returns a boolean to state whether the string contain only alphabetic and numeric characters
isalpha()	Returns a boolean to state whether the string contains all alphabetic characters
isdigit()	Returns a boolean to state whether the string contains all numeric characters
islower()	Returns a boolean to state whether a string contains all lowercase characters
isspace()	Returns a boolean to state whether the string consists of all whitespace
istitle()	Returns a boolean to state whether the first character of each word

	in the string is capitalized
isupper()	Returns a boolean to state whether all characters within the string are uppercase
join(sequence)	Returns a copy of sequence joined together with the original string placed between each element
ljust(width[,fillchar])	Returns a string of the specified width along with a copy of the original string at the leftmost bit. (Optionally padding empty space with fillchar)
lower()	Returns a copy of the original string with all characters in the string converted to lowercase
lstrip([chars])	Removes the first found characters in the string from the left that match the given characters. Also removes whitespace from the left. Whitespace removal is default when specified with no arguments.
partition(separator)	Returns a partitioned string starting from the left using the provided separator
replace(old,new[,count])	Returns a copy of the original string replacing the portion of string given in old with the portion given in new
rfind(sub[,start[,end]])	Searches string from right to left and finds the first occurrence of the given string and returns highest index where sub is found
rindex(sub[,start[,end]])	Searches string from right to left and finds the first occurrence of the given string and either returns highest index where sub is found or raises an exception
rjust(width[,fillchar])	Returns copy of string Aligned to the right by width
rpartition(separator)	Returns a copy of stringPartitioned starting from the right using the provided separator object
rsplit([separator[,maxsplit]])	Returns list of words in string and splits the string from the right side and uses the given separator as a delimiter. If maxsplit is specified then at most maxsplit splits are done (from the right).
rstrip([chars])	Returns copy of string removing the first found characters in the string from the right that match those given. Also removes whitespace from the right when no argument is specified.
split([separator[,maxsplit]])	Returns a list of words in string and splits the string from the left side and uses the given separator as a delimiter.
splitlines([keepends])	Splits the string into a list of lines. Keepends denotes if newline delimiters are removed. Returns the list of lines in the string.
startswith(prefix[,start[,end]])	Returns a boolean to state whether the string starts with the given prefix
strip([chars])	Returns a copy of string with the given characters removed from the string. If no argument is specified then whitespace is removed.
swapcase()	Returns a copy of the string the case of each character in the string converted.
title()	Returns a copy of the string with the first character in each word uppercase.

translate(table[,deletechars])	Returns a copy of the string using the given character translation table to translate the string. All characters occurring in optional deletechars argument are removed.
upper()	Returns a copy of string with all of the characters in the string converted to uppercase
zfill(width)	Returns a numeric string padded from the left with zeros for the specified width.

Now let's take a look at some examples so that you get an idea of how to use the string methods. As stated previously, most of them work in a similar manner.

Listing 2-4. Using String Methods

```
our_string='python is the best language ever'

# Capitalize first character of a String

>>> our_string.capitalize()

'Python is the best language ever'

# Center string

>>> our_string.center(50)

'         python is the best language ever         '

>>> our_string.center(50,'-')

'---------python is the best language ever---------'

# Count substring within a string

>>> our_string.count('a')

2

# Count occurrences of substrings

>>> state = 'Mississippi'

>>> state.count('ss')

2

# Partition a string returning a 3-tuple including the portion of string

# prior to separator, the separator

# and the portion of string after the separator

>>> x = "Hello, my name is Josh"
```

```
>>> x.partition('n')

('Hello, my ', 'n', 'ame is Josh')

# Assuming the same x as above, split the string using 'l' as the separator

>>> x.split('l')

['He', '', 'o, my name is Josh']

# As you can see, the tuple returned does not contain the separator value

# Now if we add maxsplits value of 1, you can see that the right-most split
is

# taken.  If we specify maxsplits value of 2, the two right-most splits are
taken

>>> x.split('l',1)

['He', 'lo, my name is Josh']

>>> x.split('l',2)

['He', '', 'o, my name is Josh']
```

String Formatting

You have many options when printing strings using the print statement. Much like the C programming language, Python string formatting allows you to make use of a number of different conversion types when printing.

Listing 2-5. Using String Formatting

```
# The two syntaxes below work the same

>>> x = "Josh"

>>> print "My name is %s" % (x)

My name is Josh

>>> print "My name is %s" % x

My name is Josh

# An example using more than one argument

>>> name = 'Josh'

>>> language = 'Python'

>>> print "My name is %s and I speak %s" % (name, language)
```

```
My name is Josh and I speak Python

# And now for some fun, here's a different conversion type

# Mind you, I'm not sure where in the world the temperature would

# fluctuate so much!

>>> day1_temp = 65

>>> day2_temp = 68

>>> day3_temp = 84

>>> print "Given the temparatures %d, %d, and %d, the average would be %f" %
(day1_temp, day2_temp, day3_temp, (day1_temp + day2_temp + day3_temp)/3)

Given the temperatures 65, 68, and 83, the average would be 72.333333
```

Table 2-3 lists the conversion types.

Table 2-3. Conversion Types

Type	Description
d	signed integer decimal
i	signed integer
o	unsigned octal
u	unsigned decimal
x	unsigned hexidecimal (lowercase)
X	unsigned hexidecimal (uppercase letters)
E	floating point exponential format (uppercase ‘E’)
e	floating point exponential format (lowercase ‘e’)
f	floating point decimal format (lowercase)
F	floating point decimal format (same as ‘f’)
g	floating point exponential format if exponent < -4, otherwise float
G	floating point exponential format (uppercase) if exponent < -4, otherwise float
c	single character
r	string (converts any python object using repr())
s	string (converts any python object using str())
%	no conversion, results in a percent (%) character if specified twice

Listing 2-6.

```
>>> x = 10

>>> y = 5.75
```

```
>>> print 'The expression %d * %f results in %f' % (x, y, x*y)

The expression 10 * 5.750000 results in 57.500000

# Example of using percentage

>>> test1 = 87

>>> test2 = 89

>>> test3 = 92

>>> "The gradepoint average of three students is %d%%" % (avg)

'The gradepoint average of three students is 89%'
```

Lists, Dictionaries, Sets, and Tuples

Lists, dictionaries, sets, and tuples all offer similar functionality and usability, but they each have their own niche in the language. We'll go through several examples of each since they all play an important role under certain circumstances. Unlike strings, all of the containers discussed in this section (except tuples) are mutable objects, so they can be manipulated after they have been created.

Because these containers are so important, we'll go through an exercise at the end of this chapter, which will give you a chance to try them out for yourself.

Lists

Perhaps one of the most used constructs within the Python programming language is the list. Most other programming languages provide similar containers for storing and manipulating data within an application. The Python list provides an advantage over those similar constructs that are available in statically typed languages. The dynamic tendencies of the Python language help the list construct to harness the great feature of having the ability to contain values of different types. This means that a list can be used to store any Python data type, and these types can be mixed within a single list. In other languages, this type of construct is often defined as a typed object, which locks the construct to using only one data type.

The creation and usage of Python lists is just the same as the rest of the language. . .very simple and easy to use. Simply assigning a set of empty square brackets to a variable creates an empty list. We can also use the built-in list() function to create a list. The list can be constructed and modified as the application runs, they are not declared with a static length. They are easy to traverse through the usage of loops, and indexes can also be used for positional placement or removal of particular items in the list. We'll start out by showing some examples of defining lists, and then go through each of the different avenues which the Python language provides us for working with lists.

Listing 2-7. Defining Lists

```
# Define an empty list

my_list = []

my_list = list()  # rarely used

# Single Item List

>>> my_list = [1]

>>> my_list           # note that there is no need to use print to display a
variable
in the

>>> # interpreter

[1]

# Define a list of string values

my_string_list = ['Hello', 'Jython' ,'Lists']

# Define a list containing mulitple data types

multi_list = [1, 2, 'three', 4, 'five', 'six']

# Define a list containing a list

combo_list = [1, my_string_list, multi_list]

# Define a list containing a list inline

>>> my_new_list = ['new_item1', 'new_item2', [1, 2, 3, 4], 'new_item3']

>>> print my_new_list

['new_item1', 'new_item2', [1, 2, 3, 4], 'new_item3']
```

As stated previously, in order to obtain the values from a list we can make use of indexes. Much like the Array in the Java language, using the list[index] notation will allow us to access an item. If we wish to obtain a range or set of values from a list, we can provide a starting index, and/or an ending index. This technique is also known as slicing. What's more, we can also return a set of values from the list along with a stepping pattern by providing a step index as well. One key to remember is that while accessing a list via indexing, the first element in the list is contained within the 0 index. Note that when slicing a list, a new list is always returned. One way to create a shallow copy of a list is to use slice notation without specifying an upper or lower bound. The lower bound defaults to zero, and the upper bound defaults to the length of the list.

Note that a shallow copy constructs a new compound object (list or other object containing objects) and then inserts references into it to the original objects. A deep copy constructs a new compound object and then inserts copies into it based upon the objects found in the original.

Listing 2-8. Accessing a List

```
# Obtain elements in the list

>>> my_string_list[0]

'Hello'

>>> my_string_list[2]

'Lists'

# Negative indexes start with the last element in the list and work back
towards the
first

# item

>>> my_string_list[-1]

'Lists'

>>> my_string_list[-2]

'Jython'

# Using slicing (Note that slice includes element at starting index and
excludes the
end)

>>> my_string_list[0:2]

['Hello', 'Jython']

# Create a shallow copy of a list using slice

>>> my_string_list_copy = my_string_list[:]

>>> my_string_list_copy

['Hello', 'Jython', 'Lists']

# Return every other element in a list

>>> new_list=[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

# Using a third parameter in the slice will cause a stepping action to take
place

# In this example we step by one
```

```
>>> new_list[0:10:1]

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

# And here we step by two

>>> new_list[0:10:2]

[2, 6, 10, 14, 18]

# Leaving a positional index blank will also work as the default is 0 for the
start,
and the length of the string for the end.

>>> new_list[::2]

[2, 6, 10, 14, 18]
```

Modifying a list is much the same, you can use the index in order to insert or remove items from a particular position. There are also many other ways that you can insert or remove elements from the list. Python provides each of these different options as they provide different functionality for your operations.

Listing 2-9.

```
# Modify an element in a list.  In this case we'll modify the element in the
9th
position

>>> new_list[9] = 25

>>> new list

[2, 4, 6, 8, 10, 12, 14, 16, 18, 25]
```

You can make use of the append() method in order to add an item to the end of a list. The extend() method allows you to add copy of an entire list or sequence to the end of a list. Lastly, the insert() method allows you to place an item or another list into a particular position of an existing list by utilizing positional indexes. If another list is inserted into an existing list then it is not combined with the original list, but rather it acts as a separate item contained within the original list. You will find examples of each method below.

Similarly, we have plenty of options for removing items from a list. The del statement, as explained in Chapter 1, can be used to remove or delete an entire list or values from a list using the index notation. You can also use the pop() or remove() method to remove single values from a list. The pop() method will remove a single value from the end of the list, and it will also return that value at the same time. If an index is provided to the pop() function, then it will remove and return the value at that index. The remove() method can be used to find and remove a particular value in the list. In other words, remove() will delete the first matching element from the list. If

more than one value in the list matches the value passed into the remove() function, the first one will be removed. Another note about the remove() function is that the value removed is not returned. Let's take a look at these examples of modifying a list.

Listing 2-10. Modifying a List

```
# Adding values to a list using the append method

>>> new_list=['a','b','c','d','e','f','g']

>>> new_list.append('h')

>>> print new_list

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

# Add another list to the existing list

>>> new_list2=['h','i','j','k','l','m','n','o','p']

>>> new_list.extend(new_list2)

>>> print new_list

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h','h', 'i', 'j', 'k', 'l', 'm', 'n',
'o', 'p']

# Insert a value into a particular location via the index.

# In this example, we add a 'c' into the third position in the list

# (Remember that list indicies start with 0, so the second index is actually
the
third

# position)

>>> new_list.insert(2,'c')

>>> print new_list

['a', 'b', 'c', 'c', 'd', 'e', 'f', 'g', 'h', 'h','i', 'j', 'k', 'l', 'm',
'n', 'o',
'p']

# Insert a list into a particular postion via the index

>>> another_list = ['a', 'b', 'c']

>>> another_list.insert(2, new_list)

>>> another_list

['a', 'b', [2, 4, 8, 10, 12, 14, 16, 18, 25], 'c']
```

```
# Use the slice notation to overwrite part of a list or sequence

>>> new_listA=[100,200,300,400]

>>> new_listB=[500,600,700,800]

>>> new_listA[0:2]=new_listB

>>> print new_listA

[500, 600, 700, 800, 300, 400]

# Assign a list to another list using the empty slice notation

>>> one = ['a', 'b', 'c', 'd']

>>> two = ['e', 'f']

>>> one

['a', 'b', 'c', 'd']

>>> two

['e', 'f']

# Obtain an empty slice from a list by using the same start and end position.

# Any start and end position will work, as long as they are the same number.

>>> one[2:2]

[]

# In itself, this is not very interesting - you could have made an empty list

# very easily. The useful thing about this is that you can assign to this
empty slice

# Now, assign the 'two' list to an empty slice for the 'one' list which
essentially

# inserts the 'two' list into the 'one' list

>>> one[2:2] = two          # the empty list between elements 1 and 2 of list
'one'
is

>>>              # replaced by the list 'two'

>>> one

['a', 'b', 'c', 'd', 'e', 'f']

# Use the del statement to remove a value or range of values from a list
```

```
# Note that all other elements are shifted to fill the empty space

>>> new_list3=['a','b','c','d','e','f']

>>> del new_list3[2]

>>> new_list3

['a', 'b', 'd', 'e', 'f']

>>> del new_list3[1:3]

>>> new_list3

['a', 'e', 'f']

# Use the del statement to delete a list

>>> new_list3=[1,2,3,4,5]

>>> print new_list3

[1, 2, 3, 4, 5]

>>> del new_list3

>>> print new_list3

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'new_list3' is not defined

# Remove values from a list using pop and remove functions

>>> print new list

['a', 'b', 'c', 'c', 'd', 'e', 'f', 'g', 'h','h', 'i', 'j', 'k', 'l', 'm',
'n', 'o',
'p']

# pop the element at index 2

>>> new_list.pop(2)

'c'

>>> print new_list

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h','h', 'i', 'j', 'k', 'l', 'm', 'n',
'o', 'p']

# Remove the first occurrence of the letter 'h' from the list

>>> new_list.remove('h')
```

```
>>> print new_list

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p']

# Useful example of using pop() function

>>> x = 5

>>> times_list = [1,2,3,4,5]

>>> while times_list:

...     print x * times_list.pop(0)

...

5

10

15

20

25
```

Now that we know how to add and remove items from a list, it is time to learn how to manipulate the data within them. Python provides a number of different methods that can be used to help us manage our lists. See Table 2-4 for a list of these functions and what they can do.

Table 2-4. Python List Methods

Method	Tasks Performed
index	Returns the index of the first value in the list which matches a given value.
count	Returns the number of items in the list which equal a given value.
sort	Sorts the items contained within the list and returns the list
reverse	Reverses the order of the items contained within the list, and returns the list

Let's take a look at some examples of how these functions can be used on lists.

Listing 2-11. Utilizing List Functions

```
# Returning the index for any given value

>>> new_list=[1,2,3,4,5,6,7,8,9,10]

>>> new_list.index(4)

3
```

```
#  Change the value of the element at index 4

>>> new_list[4] = 30

>>> new_list

[1, 2, 3, 4, 30, 6, 7, 8, 9, 10]

# Ok, let's change it back

>>> new_list[4] = 5

>>> new_list

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Add a duplicate value into the list and then return the index

# Note that index returns the index of the first matching value it encounters

>>> new_list.append(6)

>>> new_list

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 6]

>>> new_list.index(6)

5

# Using count() function to return the number of items which  equal a given
value

>>> new_list.count(2)

1

>>> new_list.count(6)

2

# Sort the values in the list

>>> new_list.sort()

>>> new_list

[1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10]

# Reverse the order of the value in the list

>>> new_list.reverse()

>>> new_list
```

```
[10, 9, 8, 7, 6, 6, 5, 4, 3, 2, 1]
```

Traversing and Searching Lists

Moving around within a list is quite simple. Once a list is populated, often times we wish to traverse through it and perform some action against each element contained within it. You can use any of the Python looping constructs to traverse through each element within a list. While there are plenty of options available, the for loop works especially well. This is because of the simple syntax that the Python for loop uses. This section will show you how to traverse a list using each of the different Python looping constructs. You will see that each of them has advantages and disadvantages.

Let's first take a look at the syntax that is used to traverse a list using a for loop. This is by far one of the easiest modes of going through each of the values contained within a list. The for loop traverses the list one element at a time, allowing the developer to perform some action on each element if so desired.

Listing 2-12. Traversing a List Using a 'for' Loop

```
>>> ourList=[1,2,3,4,5,6,7,8,9,10]
>>> for elem in ourList:
...     print elem
...
1
2
3
4
5
6
7
8
9
10
```

As you can see from this simple example, it is quite easy to go through a list and work with each item individually. The for loop syntax requires a variable to which each element in the list will be assigned for each pass of the loop.

It is also possible to combine slicing with the use of the for loop. In this case, we'll simply use a list slice to retrieve the exact elements we want to see. For instance, take a look a the following code which traverses through the first 5 elements in our list.

Listing 2-13.

```
>>> for elem in ourList[:5]:

...     print elem

...

1

2

3

4

5
```

As you can see, doing so is quite easy by simply making use of the built-in features that Python offers.

List Comprehensions

As we've seen in the previous section, we can create a copy of a list using the slicing. Another more powerful way to do so is via the list comprehension. There are some advanced features for lists that can help to make a developer's life easier. One such feature is known as a list comprehension. While this concept may be daunting at first, it offers a good alternative to creating many separate lists manually. List comprehensions take a given list, and then iterate through it and apply a given expression against each of the objects in the list.

Listing 2-14. Simple List Comprehension

```
# Multiply each number in a list by 2 using a list comprehension

# Note that list comprehension returns a new list

>>> num_list = [1, 2, 3, 4]

>>> [num * 2 for num in num_list]

[2, 4, 6, 8]

# We could assign a list comprehension to a variable

>>> num_list2 = [num * 2 for num in num_list]
```

```
>>> num_list2

[2, 4, 6, 8]
```

As you can see, this allows one to quickly take a list and alter it via the use of the provided expression. Of course, as with many other Python methods the list comprehension returns an altered copy of the list. The list comprehension produces a new list and the original list is left untouched.

Let's take a look at the syntax for a list comprehension. They are basically comprised of an expression of some kind followed by a for statement and then optionally more for or if statements. The basic functionality of a list comprehension is to iterate over the items of a list, and then apply some expression against each of the list's members. Syntactically, a list comprehension reads as follows:

Iterate through a list and optionally perform an expression on each element, then either return a new list containing the resulting elements or evaluate each element given an optional clause.

```
[list-element (optional expression) for list-element in list (optional
clause)]
```

Listing 2-15. Using an If Clause in a List Comprehension

```
# The following example returns each element

# in the list that is greater than the number 4

>>> nums = [2, 4, 6, 8]

>>> [num for num in nums if num > 4]

[6, 8]
```

Let's take a look at some more examples. Once you've seen list comprehensions in action you are sure to understand them and see how useful they can be.

Listing 2-16. Python List Comprehensions

```
# Create a list of ages and add one to each of those ages using a list
comprehension

>>> ages=[20,25,28,30]

>>> [age+1 for age in ages]

[21, 26, 29, 31]

# Create a list of names and convert the first letter of each name to
uppercase as it
```

```
should be

>>> names=['jim','frank','vic','leo','josh']

>>> [name.title() for name in names]

['Jim', 'Frank', 'Vic', 'Leo', 'Josh']

# Create a list of numbers and return the square of each EVEN number

>>> numList=[1,2,3,4,5,6,7,8,9,10,11,12]

>>> [num*num for num in numList if num % 2 == 0]

[4, 16, 36, 64, 100, 144]

# Use a list comprehension with a range

>>> [x*5 for x in range(1,20)]

[5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]

# Use a for clause to perform calculations against elements of two different
lists

>>> list1 = [5, 10, 15]

>>> list2 = [2, 4, 6]

>>> [e1 + e2 for e1 in list1 for e2 in list2]

[7, 9, 11, 12, 14, 16, 17, 19, 21]
```

List comprehensions can make code much more concise and allows one to apply expressions or functions to list elements quite easily. Let's take a quick look at an example written in Java for performing the same type of work as an list comprehension. It is plain to see that list comprehensions are much more concise.

Listing 2-17. Java Code to Take a List of Ages and Add One Year to Each Age

```
int[] ages = {20, 25, 28, 30};

int[] ages2 = new int[ages.length];

// Use a  Java for loop to go through each element in the array

for (int x = 0; x <= ages.length; x++){

;

ages2[x] = ages[x]+1;

}
```

Tuples

Tuples are much like lists; however, they are immutable. Once a tuple has been defined, it cannot be changed. They contain indexes just like lists, but again, they cannot be altered once defined. Therefore, the index in a tuple may be used to retrieve a particular value and not to assign or modify. While tuples may appear similar to lists, they are quite different in that tuples usually contain heterogeneous elements, whereas lists oftentimes contain elements that are related in some way. For instance, a common use case for tuples is to pass parameters to a function, method, and so on.

Since tuples are a member of the sequence type, they can use the same set of methods an operations available to all sequence types.

Listing 2-18. Examples of Tuples

```
# Creating an empty tuple

>>> myTuple = ()

# Creating tuples and using them

>>> myTuple2 = (1, 'two',3, 'four')

>>> myTuple2

(1, 'two', 3, 'four')

# To create a single-item tuple, include a trailing comma

>>> myteam = 'Bears',

>>> myteam

('Bears',)
```

As mentioned previously, tuples can be quite useful for passing to functions, methods, classes, and so on. Oftentimes, it is nice to have an immutable object for passing multiple values. One such case would be using a tuple to pass coordinates in a geographical information system or another application of the kind. They are also nice to use in situations where an immutable object is warranted. Because they are immutable, their size does not grow once they have been defined, so tuples can also play an important role when memory allocation is a concern.

Dictionaries

A Python dictionary is a key-value store container. A dictionary is quite different than a typical list in Python as there is no automatically populated index for any given element within the dictionary. When you use a list, you need not worry about assigning an index to any value that is placed within it. A dictionary allows the developer to assign an index or “key” for every

element that is placed into the construct. Therefore, each entry into a dictionary requires two values, the key and the element.

The beauty of the dictionary is that it allows the developer to choose the data type of the key value. Therefore, if one wishes to use a string or any other hashable object such as an int or float value as a key then it is entirely possible. Dictionaries also have a multitude of methods and operations that can be applied to them to make them easier to work with. Table 2-5 lists dictionary methods and functions.

Listing 2-19. Basic Dictionary Examples

```
# Create an empty dictionary and a populated dictionary

>>> myDict={}

>>> myDict.values()

[]

# Assign key-value pairs to dictionary

>>> myDict['one'] = 'first'

>>> myDict['two'] = 'second'

>>> myDict

{'two': 'second', 'one': 'first'}
```

Table 2-5. Dictionary Methods and Functions

Method or Function	Description
len(dictionary)	Function that returns number of items within the given dictionary.
dictionary [key]	Returns the item from the dictionary that is associated with the given key.
dictionary[key] = value	Sets the associated item in the dictionary to the given value.
del dictionary[key]	Deletes the given key/value pair from the dictionary.
dictionary.clear()	Method that removes all items from the dictionary.
dictionary.copy()	Method that creates a shallow copy of the dictionary.
has_key(key)	Function that returns a boolean stating whether the dictionary contains the given key. (Deprecated in favor of using in')
key in d	Returns a boolean stating whether the given key is found in the dictionary
key not in d	Returns a boolean stating whether the given key is not found in the dictionary
items()	Returns a list of tuples including a copy of the key/value pairs within the dictionary.

keys()	Returns the a list of keys within the dictionary.
update([dictionary2])	Updates dictionary with the key/value pairs from the given dictionary. Existing keys will be overwritten.
fromkeys(sequence[,value])	Creates a new dictionary with keys from the given sequence. The values will be set to the value given.
values()	Returns the values within the dictionary as a list.
get(key[, b])	Returns the value associated with the given key. If the key does not exist, then returns b.
setdefault(key[, b])	Returns the value associated with the given key. If the key does not exist, then the key value is set to b (mydict[key] = b)
pop(key[, b])	Returns and removes the key/value pair associated with the given key. If the key does not exist then returns b.
popItem()	An arbitrary key/value pair is popped from the dictionary
iteritems()	Returns an iterator over the key/value pairs in the dictionary.
iterkeys()	Returns an iterator over the keys in the dictionary.
itervalues()	Returns an iterator over the values in the dictionary.

Now we will take a look at some dictionary examples. This reference will not show you an example of using each of the dictionary methods and functions, but it should provide you with a good enough base understanding of how they work.

Listing 2-20. Working with Python Dictionaries

```
# Create an empty dictionary and a populated dictionary

>>> mydict = {}

# Try to find a key in the dictionary

>>> 'firstkey' in mydict

False

# Add key/value pair to dictionary

>>> mydict['firstkey'] = 'firstval'

>>> 'firstkey' in mydict

True

# List the values in the dictionary

>>> mydict.values()

['firstval']

# List the keys in the dictionary

>>> mydict.keys()
```

```
['firstkey']

# Display the length of the dictionary (how many  key/value pairs are in it)

>>> len(mydict)

1

# Print the contents of the dictionary

>>> mydict

{'firstkey': 'firstval'}

>>>

# Replace the original dictionary with a dictionary containing string-based
keys

# The following dictionary represents a hockey team line

>>> myDict =
{'r_wing':'Josh','l_wing':'Frank','center':'Jim','l_defense':'Leo','r_defense
':'Vic'}

>>> myDict.values()

['Josh', 'Vic', 'Jim', 'Frank', 'Leo']

>>> myDict.get('r_wing')

'Josh'

>>> myDict['r_wing']

'Josh'

# Try to obtain the value for a key that does not exist

>>> myDict['goalie']

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'goalie'

# Try to obtain a value for a key that does not exist using get()

>>> myDict.get('goalie')

# Now use a default message that will be displayed if the key does not exist

>>> myDict.get('goalie','Invalid Position')
```

```
'Invalid Position'

# Iterate over the items in the dictionary

>>> for player in myDict.iterItems():

...     print player

...

('r_wing', 'Josh')

('r_defense', 'Vic')

('center', 'Jim')

('l_wing', 'Frank')

('l_defense', 'Leo')

# Assign keys and values to separate objects and then print

>>> for key,value in myDict.iteritems():

...     print key, value

...

r_wing Josh

r_defense Vic

center Jim

l_wing Frank

l_defense Leo
```

Sets

Sets are unordered collections of unique elements. What makes sets different than other sequence types is that they contain no indexing or duplicates. They are also unlike dictionaries because there are no key values associated with the elements. They are an arbitrary collection of unique elements. Sets cannot contain mutable objects, but sets themselves can be mutable. Another thing to note is that sets are note available to use by default, you must import set from the Sets module before using.

Listing 2-21. Examples of Sets

```
# In order to use a Set, we must first import it

>>> from sets import Set
```

```
# To create a set use the following syntax

>>> myset = Set([1,2,3,4,5])

>>> myset

Set([5, 3, 2, 1, 4])

# Add a value to the set – See Table 2-7 for more details

>>> myset.add(6)

>>> myset

Set([6, 5, 3, 2, 1, 4])

# Try to add a duplicate

>>> myset.add(4)

>>> myset

Set([6, 5, 3, 2, 1, 4])
```

There are two different types of sets, namely set and frozenset. The difference between the two is quite easily conveyed from the name itself. A regular set is a mutable collection object, whereas a frozen set is immutable. Remember, immutable objects cannot be altered once they have been created whereas mutable objects can be altered after creation. Much like sequences and mapping types, sets have an assortment of methods and operations that can be used on them. Many of the operations and methods work on both mutable and immutable sets. However, there are a number of them that only work on the mutable set types. In Tables 2-6 and 2-7, we'll take a look at the different methods and operations.

Table 2-6. Set Type Methods and Functions

Method or Operation	Description
len(set)	Returns the number of elements in a given set
copy()	Returns a new shallow copy of the set
difference(set2)	Returns a new set that contains all elements that are in the calling set, but not in set2
intersection(set2)	Returns a new set that contains all elements that the calling set and set2 have in common
issubbset(set2)	Returns a Boolean stating whether all elements in calling set are also in set2
issuperset(set2)	Returns a Boolean stating whether all elements in set2 are contained in calling set
symmetric_difference(set2)	Returns a new set containing elements either from the calling set or set2 but not from both (set1 ^ set2)
x in set	Tests whether x is contained in the set, returns boolean

x not in set	Tests whether x is not contained in the set, returns boolean
union(set2)	Returns a new set containing elements that are contained in both the calling set and set2

Listing 2-22. Using Set Type Methods and Functions

```
# Create two sets

>>> s1 = Set(['jython','cpython','ironpython'])

>>> s2 = Set(['jython','ironpython','pypy'])

# Make a copy of a set

>>> s3 = s1.copy()

>>> s3

Set(['cpython', 'jython', 'ironpython'])

# Obtain a new set containing all elements that are in s1 but not s2

>>> s1.difference(s2)

Set(['cpython'])

# Obtain a new set containing all elements from each set

>>> s1.union(s2)

Set(['cpython', 'pypy', 'jython', 'ironpython'])

# Obtain a new set containing elements from either set that are not contained
in both

>>> s1.symmetric_difference(s2)

Set(['cpython', 'pypy'])
```

Table 2-7. Mutable Set Type Methods

Method or Operation	Description
add(item)	Adds an item to a set if it is not already in the set
clear()	Removes all items in a set
difference_update(set2)	Returns the set with all elements contained in set2 removed
discard(element)	Removes designated element from set if present
intersection_update(set2)	Returns the set keeping only those elements that are also in set2
pop()	Return an arbitrary element from the set

remove(element)	Remove element from set if present, if not then KeyError is raised
symmetric_difference_update(set2)	Replace the calling set with a set containing elements from either the calling set or set2 but not both, and return it
update(set2)	Returns set including all elements from set2

Listing 2-23. More Using Sets

```
# Create three sets
>>> s1 = Set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> s2 = Set([5, 10, 15, 20])
>>> s3 = Set([2, 4, 6, 8, 10])
# Remove arbitrary element from s2
>>> s2.pop()
20
>>> s2
Set([5, 15, 10])
# Discard the element that equals 3 from s1 (if exists)
>>> s1.discard(3)
>>> s1
Set([6, 5, 7, 8, 2, 9, 10, 1, 4])
# Update s1 to include only those elements contained in both s1 and s2
>>> s1.intersection_update(s2)
>>> s1
Set([5, 10])
>>> s2
Set([5, 15, 10])
# Remove all elements in s2
>>> s2.clear()
>>> s2
Set([])
```

```
# Updates set s1 to include all elements in s3

>>> s1.update(s3)

>>> s1

Set([6, 5, 8, 2, 10, 4])
```

Ranges

The range is a special function that allows one to iterate between a range of numbers or list a specific range of numbers. It is especially helpful for performing mathematical iterations, but it can also be used for simple iterations.

The format for using the range function includes an optional starting number, an ending number, and an optional stepping number. If specified, the starting number tells the range where to begin, whereas the ending number specifies where the range should end. The starting index is inclusive whereas the ending index is not. The optional step number tells the range how many numbers should be placed between each number contained within the range output. The step number is added to the previous number and if that number exceeds the end point then the range stops.

Range Format

```
range([start], stop, [step])
```

Listing 2-24. Using the Range Function

```
#Simple range starting with zero, note that the end point is not included in
the
range

>>>range(0,10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(50, 65)

[50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64]

>>>range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Include a step of two in the range

>>>range(0,10,2)

[0, 2, 4, 6, 8]
```

```
# Including a negative step performs the same functionality...the step is
added to
the previously

# number in the range

>>> range(100,0,-10)

[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
```

One of the most common uses for this function is in a for loop. The following example displays a couple ways of using the range function within a for loop context.

Listing 2-25. Using the Range Function Within a For Loop

```
>>> for i in range(10):

...         print i

...

0

1

2

3

4

5

6

7

8

9

# Multiplication Example

>>> x = 1

>>> for i in range(2, 10, 2):

...         x = x + (i * x)

...         print x

...

3
```

```
15

105

945
```

As you can see, a range can be used to iterate through just about any number set. . .be it going up or down, positive or negative in step. Ranges are also a good way to create a list of numbers. In order to do so, simply pass a range to list() as shown in the following example.

Listing 2-26. Create a List from a Range

```
>>> my_number_list = list(range(10))

>>> my_number_list

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

As you can see, not only are ranges useful for iterative purposes but they are also a good way to create numeric lists.

Jython-specific Collections

There are a number of Jython-specific collection objects that are available for use. Most of these collection objects are used to pass data into Java classes and so forth, but they add additional functionality into the Jython implementation that will assist Python newcomers that are coming from the Java world. Nonetheless, many of these additional collection objects can be quite useful under certain situations.

In the Jython 2.2 release, Java collection integration was introduced. This enables a bidirectional interaction between Jython and Java collection types. For instance, a Java ArrayList can be imported in Jython and then used as if it were part of the language. Prior to 2.2, Java collection objects could act as a Jython object, but Jython objects could not act as Java objects. For instance, it is possible to use a Java ArrayList in Jython and use methods such as add(), remove(), and get(). You will see in the example below that using the add() method of an ArrayList will add an element to the list and return a boolean to signify the success or failure of the addition. The remove() method acts similarly, except that it removes an element rather than adding it.

Listing 2-27. Example of Using Java Oriented Collection in Jython

```
# Import and use a Java ArrayList

>>> import java.util.ArrayList as ArrayList

>>> arr = ArrayList()
```

```
#  Add method will add an element to the list and return a boolean to signify
successsful addition

>>> arr.add(1)

True

>>> arr.add(2)

True

>>> print arr

[1, 2]
```

Ahead of the integration of Java collections, Jython also had implemented the jarray object which basically allows for the construction of a Java array in Jython. In order to work with a jarray, simply define a sequence type in Jython and pass it to the jarray object along with the type of object contained within the sequence. The jarray is definitely useful for creating Java arrays and then passing them into java objects, but it is not very useful for working in Jython objects. Moreover, all values within a jarray must be the same type. If you try to pass a sequence containing multiple types to a jarray then you'll be given a TypeError of one kind or another. See Table 2-8 for a listing of character typecodes used with jarray.

Table 2-8. Character Typecodes for Use With Jarray

Character	Java Equivalent
z	boolean
b	byte
c	char
d	Double
f	Float
h	Short
i	Int
l	Long

Listing 2-28. Jarray Usage

```
>>> my_seq = (1,2,3,4,5)

>>> from jarray import array

>>> array(my_seq,'i')

array('i', [1, 2, 3, 4, 5])

>>> myStr = "Hello Jython"
```

```
>>> array(myStr,'c')

array('c', 'Hello Jython')
```

Another useful feature of the jarray is that we can create empty arrays if we wish by using the zeros() method. The zeros() method works in a similar fashion to the array() method which we've already demonstrated. In order to create an array that is empty, simply pass the length of the array along with the type to the zeros() method. Let's take a quick look at an example.

Listing 2-29. Create an Empty Boolean Array

```
>>> arr = zeros(10,'z')

>>> arr

array('z', [False, False, False, False, False, False, False, False, False,
False])
```

Listing 2-30. Create an Empty Integer Array

```
>>> arr2 = zeros(6, 'i')

>>> arr2

array('i', [0, 0, 0, 0, 0, 0])
```

In some circumstances when working with Java objects, you will need to call a Java method that requires a Java array as an argument. Using the jarray object allows for a simple way of creating Java arrays when needed.

Files

File objects are used to read and write data to a file on disk. The file object is used to obtain a reference to the file on disk and open it for reading, writing, appending, or a number of different tasks. If we simply use the open(filename[, mode]) function, we can return a file object and assign it to a variable for processing. If the file does not yet exist on disk, then it will automatically be created. The mode argument is used to tell what type of processing we wish to perform on the file. This argument is optional and if omitted then the file is opened in read-only mode. See Table 2-9.

Table 2-9. Modes of Operations for File Types

Mode	Description
'r'	read only
'w'	write (Note: This overwrites anything else in the file, so use with caution)
'a'	append

‘r+’	read and write
‘rb’	binary file read
‘wb’	binary file write
‘r+b’	binary file read and write

Listing 2-31.

```
# Open a file and assign it to variable f

>>> f = open('newfile.txt','w')
```

There are plenty of methods that can be used on file objects for manipulation of the file content. We can call read([size]) on a file in order to read its content. Size is an optional argument here and it is used to tell how much content to read from the file. If it is omitted then the entire file content is read. The readline() method can be used to read a single line from a file. readlines([size]) is used to return a list containing all of the lines of data that are contained within a file. Again, there is an optional size parameter that can be used to tell how many bytes from the file to read. If we wish to place content into the file, the write(string) method does just that. The write() method writes a string to the file.

When writing to a file it is oftentimes important to know exactly what position in the file you are going to write to. There are a group of methods to help us out with positioning within a file using integers to represent bytes in the file. The tell() method can be called on a file to give the file object’s current position. The integer returned is in a number of bytes and is an offset from the beginning of the file. The seek(offset, from) method can be used to change position in a file. The offset is the number in bytes of the position you’d like to go, and from represents the place in the file where you’d like to calculate the offset from. If from equals 0, then the offset will be calculated from the beginning of the file. Likewise, if it equals 1 then it is calculated from the current file position, and 2 will be from the end of the file. The default is 0 if from is omitted.

Lastly, it is important to allocate and de-allocate resources efficiently in our programs or we will incur a memory overhead and leaks. Resources are usually handled a bit differently between CPython and Jython because garbage collection acts differently. In CPython, it is not as important to worry about de-allocating resources as they are automatically de-allocated when they go out of scope. The JVM does note immediately garbage collect, so proper de-allocation of resources is more important. The close() method should be called on a file when we are through working with it. The proper methodology to use when working with a file is to open, process, and then close each time. However, there are more efficient ways of performing such tasks. In Chapter 7 we will discuss the use of context managers to perform the same functionality in a more efficient manner.

Listing 2-32. File Manipulation in Python

```
# Create a file, write to it, and then read its content
```

```
>>> f = open('newfile.txt','r+')

>>> f.write('This is some new text for our file\n')

>>> f.write('This should be another line in our file\n')

#  No lines will be read because we are at the end of the written content

>>> f.read()

''

>>> f.readlines()

[]

>>> f.tell()

75L

# Move our position back to the beginning of the file

>>> f.seek(0)

>>> f.read()

'This is some new text for our file\nThis should be another line in our
file\n'

>>> f.seek(0)

>>> f.readlines()

['This is some new text for our file\n', 'This should be another line in our
file\n']

# Closing the file to de-allocate

>>> f.close()
```

Iterators

The iterator was introduced into Python back in version 2.2. It allows for iteration over Python containers. All iterable containers have built-in support for the iterator type. For instance, sequence objects are iterable as they allow for iteration over each element within the sequence. If you try to return an iterator on an object that does not support iteration, you will most likely receive an AttributeError which tells you that __iter__ has not been defined as an attribute for that object. It is important to note that Python method names using double-underscores are special methods. For instance, in Python a class can be initialized using the __init__() method. . .much like a Java constructor. For more details on classes and special class methods, please refer to Chapter 7.

Iterators allow for easy access to sequences and other iterable containers. Some containers such as dictionaries have specialized iteration methods built into them as you have seen in previous sections. Iterator objects are required to support two main methods that form the iterator protocol. Those methods are defined below in Table 2-10.To return an iterator on a container, just assign container.__iter__() to some variable. That variable will become the iterator for the object. This affords one the ability to pass iterators around, into functions and the like. The iterator is then itself like a changing variable that maintains its state. We can use work with the iterator without affecting the original object. If using the next() call, it will continue to return the next item within the list until all items have been retrieved. Once this occurs, a StopIteration exception is issued. The important thing to note here is that we are actually creating a copy of the list when we return the iterator and assign it to a variable. That variable returns and removes an item from that copy each time the next() method is called on it. If we continue to call next() on the iterator variable until the StopIteration error is issued, the variable will no longer contain any items and is empty. For instance, if we created an iterator from a list then called the next() method on it until it had retrieved all values then the iterator would be empty and the original list would be left untouched.

Listing 2-33. Create an Iterator from a List and Use It

```
>>> hockey_roster = ['Josh', 'Leo', 'Frank', 'Jim', 'Vic']

>>> hockey_itr = hockey_roster.__iter__()

>>> hockey_itr = hockey_roster.__iter__()

>>> hockey_itr.next()

'Josh'

>>> for x in hockey_itr:

...     print x

...

Leo

Frank

Jim

Vic

# Try to call next() on iterator after it has already used all of its
elements

>>> hockey_itr.next()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>
```

```
StopIteration
```

Listing 2-34. Iteration Over Sequence and List

```
# Iterate over a string and a list

>>> str_a = 'Hello'

>>> list_b = ['Hello','World']

>>> for x in str_a:

...     print x

...

H

e

l

l

o

>>> for y in list_b:

...     print y + '!'

...

Hello!

World!
```

Referencing and Copies

Creating copies and referencing items in the Python language is fairly straightforward. The only thing you'll need to keep in mind is that the techniques used to copy mutable and immutable objects differ a bit.

In order to create a copy of an immutable object, you simply assign it to a different variable. The new variable is an exact copy of the object. If you attempt to do the same with a mutable object, you will actually just create a reference to the original object. Therefore, if you perform operations on the "copy" of the original then the same operation will actually be performed on the original. This occurs because the new assignment references the same mutable object in memory as the original. It is kind of like someone calling you by a different name. One person may call you by your birth name and another may call you by your nickname, but both names will reference you of course.

Listing 2-35. Working with Copies

```
# Strings are immutable, so when you assign a string to another variable, it
creates
a real copy

>>> mystring = "I am a string, and I am an immutable object"

>>> my_copy = mystring

>>> my_copy

'I am a string, and I am an immutable object'

>>> mystring

'I am a string, and I am an immutable object'

>>> my_copy = "Changing the copy of mystring"

>>> my_copy

'Changing the copy of mystring'

>>> mystring

'I am a string, and I am an immutable object'

# Lists are mutable objects, so assigning a list to a variable

# creates a reference to that list. Changing one of these variables will also

# change the other one – they are just references to the same object.

>>> listA = [1,2,3,4,5,6]

>>> print listA

[1, 2, 3, 4, 5, 6]

>>> listB = listA

>>> print listB

[1, 2, 3, 4, 5, 6]

>>> del listB[2]

# Oops, we've altered the original list!

>>> print listA

[1, 2, 4, 5, 6]

# If you want a new list which contains the same things, but isn't just a
reference
```

```
# to your original list, you need the copy module

>>> import copy

>>> a = [[]]

>>> b = copy.copy(a)

>>> b

[[]]

# b is not the same list as a, just a copy

>>> b is a

False

# But the list b[0] is the same the same list as the list a[0], and changing
one will

# also change the other. This is what is known as a shallow copy – a and b
are

# different at the top level, but if you go one level down, you have
references to

# to the same things – if you go deep enough, it's not a copy,

# it's the same object.

>>> b[0].append('test')

>>> a

[['test']]

>>> b

[['test']]
```

To effectively create a copy of a mutable object, you have two choices. You can either create what is known as a shallow copy or a deep copy of the original object. The difference is that a shallow copy of an object will create a new object and then populate it with references to the items that are contained in the original object. Hence, if you modify any of those items then each object will be affected since they both reference the same items.

A deep copy creates a new object and then recursively copies the contents of the original object into the new copy. Once you perform a deep copy of an object then you can perform operations on any object contained in the copy without affecting the original. You can use the deepcopy function in the copy module of the Python standard library to create such a copy. Let's look at some more examples of creating copies in order to give you a better idea of how this works.

Listing 2-36.

```
# Create an integer variable, copy it, and modify the copy
>>> a = 5
>>> b = a
>>> print b
5
>>> b = a * 5
>>> b
25
>>> a
5
# Create a deep copy of the list and modify it
>>> import copy
>>> listA = [1,2,3,4,5,6]
>>> listB = copy.deepcopy(listA)
>>> print listB
[1, 2, 3, 4, 5, 6]
>>> del listB[2]
>>> print listB
[1, 2, 4, 5, 6]
>>> print listA
[1, 2, 3, 4, 5, 6]
```

Garbage Collection

This is one of those major differences between CPython and Jython. In CPython, an object is garbage collected when it goes out of scope or is no longer needed. This occurs automatically and rarely needs to be tracked by the developer. Behind the scenes, CPython uses a reference counting technique to maintain a count on each object which effectively determines if the object is still in use. Unlike CPython, Jython does not implement a reference counting technique for aging out or garbage collection unused objects. Instead, Jython makes use of the garbage collection mechanisms that the Java platform provides. When a Jython object becomes stale or

unreachable, the JVM may or may not reclaim it. One of the main aspects of the JVM that made developers so happy in the early days is that there was no longer a need to worry about cleaning up after your code. In the C programming language, one must maintain an awareness of which objects are currently being used so that when they are no longer needed the program would perform some clean up. Not in the Java world, the gc thread on the JVM takes care of all garbage collection and cleanup for you.

Even though we haven't spoken about classes in detail yet, you saw a short example of how them in Chapter 1. It is a good time to mention that Python provides a mechanism for object cleanup. A finalizer method can be defined in any class in order to ensure that the garbage collector performs specific tasks. Any cleanup code that needs to be performed when an object goes out of scope can be placed within this finalizer method. It is important to note that the finalizer method cannot be counted on as a method which will always be invoked when an object is stale. This is the case because the finalizer method is invoked by the Java garbage collection thread, and there is no way to be sure when and if the garbage collector will be called on an object. Another issue of note with the finalizer is that they incur a performance penalty. If you're coding an application that already performs poorly then it may not be a good idea to throw lots of finalizers into it.

The following is an example of a Python finalizer. It is an instance method that must be named __del__.

Listing 2-37. Python Finalizer Example

```
class MyClass:

def __del__(self):

pass    # Perform some cleanup here
```

The downside to using the JVM garbage collection mechanisms is that there is really no guarantee as to when and if an object will be reclaimed. Therefore, when working with performance intensive objects it is best to not rely on a finalizer to be called. It is always important to ensure that proper coding techniques are used in such cases when working with objects like files and databases. Never code the close() method for a file into a finalizer because it may cause an issue if the finalizer is not invoked. Best practice is to ensure that all mandatory cleanup activities are performed before a finalizer would be invoked.

Summary

A lot of material was covered in this chapter. You should be feeling better acquainted with Python after reading through this material. We began the chapter by covering the basics of assignment an assigning data to particular objects or data types. You learned that working with each type of data object opens different doors as the way we work with each type of data object differs. Our journey into data objects began with numbers and strings, and we discussed the many methods available to the string object. We learned that strings are part of the sequence

family of Python collection objects along with lists and tuples. We covered how to create and work with lists, and the variety of options available to us when using lists. You discovered that list comprehensions can help create copies of a given list and manipulate their elements according to an expression or function. After discussing lists, we went on to discuss dictionaries, sets and tuples.

After discussing the collection types, we learned that Jython has its own set of collection objects that differ from those in Python. We can leverage the advantage of having the Java platform at our fingertips and use Java collection types from within Jython. We finished up by discussing referencing, copies, and garbage collection. Creating different copies of objects does not always give you what you'd expect, and that Jython garbage collection differs quite a bit from that of Python.

Source: http://www.jython.org/jythonbook/en/1.0/DataTypes.html