Operators, Expressions, and Program Flow

The focus of this chapter is an in-depth look at each of the ways that we can evaluate code, and write meaningful blocks of conditional logic. We'll cover the details of many operators that can be used in Python expressions. This chapter will also cover some topics that have already been discussed in more meaningful detail such as the looping constructs, and some basic program flow.

We'll begin by discussing details of expressions. If you'll remember from Chapter 1, an expression is a piece of code that evaluates to produce a value. We have already seen some expressions in use while reading through the previous chapters. In this chapter, we'll focus more on the internals of operators used to create expressions, and also different types of expressions that we can use. This chapter will go into further detail on how we can define blocks of code for looping and conditionals.

This chapter will also go into detail on how you write and evaluate mathematical expressions, and Boolean expressions. And last but not least, we'll discuss how you can use augmented assignment operations to combine two or more operations into one.

Types of Expressions

An expression in Python is a piece of code that produces a result or value. Most often, we think of expressions that are used to perform mathematical operations within our code. However, there are a multitude of expressions used for other purposes as well. In Chapter 2, we covered the details of String manipulation, sequence and dictionary operations, and touched upon working with sets. All of the operations performed on these objects are forms of expressions in Python. Other examples of expressions could be pieces of code that call methods or functions, and also working with lists using slicing and indexing.

Mathematical Operations

The Python contains all of your basic mathematical operations. This section will briefly touch upon each operator and how it functions. You will also learn about a few built-in functions which can be used to assist in your mathematical expressions.

Assuming that this is not the first programming language you are learning, there is no doubt that you are at least somewhat familiar with performing mathematical operations within your programs. Python is no different than the rest when it comes to mathematics, as with most programming languages, performing mathematical computations and working with numeric expressions is straightforward. Table 3-1 lists the numeric operators.

Table 3-1. Numeric Operators

Operator	Description

+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Truncating Division
%	Modulo (Remainder of Division)
**	Power Operator
+var	Unary Plus
-var	Unary Minus

Most of the operators in Table 3-1 work exactly as you would expect, so for example:

Listing 3-1. Mathematical Operator

```
# Performing basic mathematical computations
>>> 10 - 6
4
>>> 9 * 7
63
```

However, division, truncating division, modulo, power, and the unary operators could use some explanation. Truncating division will automatically truncate a division result into an integer by rounding down, and modulo will return the remainder of a truncated division operation. The power operator does just what you'd expect as it returns the result of the number to the left of the operator multiplied by itself n times, where n represents the number to the right of the operator.

Listing 3-2. Truncating Division and Powers

```
>>> 36 // 5
7
# Modulo returns the remainder
>>> 36 % 5
1
# Using powers, in this case 5 to the power of 2
>>> 5**2
25
# 100 to the power of 2
>>> 100**2
10000
```

Division itself is an interesting subject as its current implementation is somewhat controversial in some situations. The problem 10/5 = 2 definitely holds true. However, in its current implementation, division rounds numbers in such a way that sometimes yields unexpected results. There is a new means of division available in Jython 2.5 by importing from __future__. In a standard division for 2.5 and previous releases, the quotient returned is the floor (nearest integer after rounding down) of the quotient when arguments are ints or longs. However, a

reasonable approximation of the division is returned if the arguments are floats or complex. Often times this solution is not what was expected as the quotient should be the reasonable approximation or “true division” in any case. When we import division from the __future__ module then we alter the return value of division by causing true division when using the / operator, and floor division only when using the , // operator. In an effort to not break backward compatibility, the developers have placed the repaired division implementation in a module known as __future__. The __future__ module actually contains code that is meant to be included as a part of the standard language in some future revision. In order to use the new repaired version of division, it is important that you always import from __future__ prior to working with division. Take a look at the following piece of code.

Listing 3-3. Division Rounding Issues

```
# Works as expected
>>> 14/2
7
>>> 10/5
2
>>> 27/3
9
# Now divide some numbers that should result in decimals
# Here we would expect 1.5
>>> 3/2
1
# The following should give us 1.4
>>> 7/5
1
# In the following case, we'd expect 2.3333
>>> 14/6
2
```

As you can see, when we’d expect to see a decimal value we are actually receiving an integer value. The developers of this original division implementation have acknowledged this issue and repaired it using the new __future__ implementation.

Listing 3-4. Working With __future__ Division

```
# We first import division from __future__
from __future__ import division
# We then work with division as usual and see the expected results
>>> 14/2
7.0
>>> 10/5
2.0
>>> 27/3
9.0
>>> 3/2
1.5
>>> 7/5
1.4
>>> 14/6
```

```
2.3333333333333335
```

It is important to note that the Jython implementation differs somewhat from CPython in that Java provides extra rounding in some cases. The differences are in display of the rounding only as both Jython and CPython use the same IEEE float for storage. Let's take a look at one such case.

Listing 3-5. Subtle Differences Between Jython and CPython Division

```
# CPython 2.5 Rounding
>>> 5.1/1
5.0999999999999996
# Jython 2.5
>>> 5.1/1
5.1
```

Unary operators can be used to evaluate positive or negative numbers. The unary plus operator multiplies a number by positive 1 (which generally doesn't change it at all), and a unary minus operator multiplies a number by negative 1.

Listing 3-6. Unary Operators

```
# Unary minus
>>> -10 + 5
-5
>>> +5 - 5
0
>>> -(1 + 2)
-3
```

As stated at the beginning of the section, there are a number of built-in mathematical functions that are at your disposal. Table 3-2 lists the built-in mathematical functions.

Table 3-2. Mathematical Built-in Functions

Function	Description
abs(var)	Absolute value
pow(x, y)	Can be used in place of ** operator
pow(x,y,modulo)	Ternary power-modulo (x **y) % modulo
round(var[, n])	Returns a value rounded to the nearest 10-n or (10**-n), where n defaults to 0)
divmod(x, y)	Returns a tuple of the quotient and the remainder of division

Listing 3-7. Mathematical Built-ins

```
#  The following code provides some examples for using mathematical built-ins
# Absolute value of 9
```

```
>>> abs(9)
9
# Absolute value of -9
>>> abs(-9)
9
# Divide 8 by 4 and return quotient, remainder tuple
>>> divmod(8,4)
(2, 0)
# Do the same, but this time returning a remainder (modulo)
>>> divmod(8,3)
(2, 2)
# Obtain 8 to the power of 2
>>> pow(8,2)
64
# Obtain 8 to the power of 2 modulo 3  ((8 **2) % 3)
>>> pow(8,2,3)
1
# Perform rounding
>>> round(5.67,1)
5.7
>>> round(5.67)
6.00
```

Comparison Operators

Comparison operators can be used for comparison of two or more expressions or variables. As with the mathematical operators described above, these operators have no significant difference to that of Java. See Table 3-3.

Table 3-3. Comparison Operators

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
!=	Not equal
==	Equal

Listing 3-8. Examples of Comparison Operators

```
# Simple comparisons
>>> 8 > 10
False
>>> 256 < 725
True
>>> 10 == 10
True
# Use comparisons in an expression
>>> x = 2*8
```

```
>>> y = 2
>>> while x != y:
...     print 'Doing some work...'
...     y = y + 2
...
Doing some work...
Doing some work...
Doing some work...
Doing some work...
Doing some work...
Doing some work...
Doing some work...
# Combining comparisons
>>> 3<2<3
False
>>> 3<4<8
True
```

Bitwise Operators

Bitwise operators in Python are a set of operators that are used to work on numbers in a two's complement binary fashion. That is, when working with bitwise operators numbers are treated as a string of bits consisting of 0s and 1s. If you are unfamiliar with the concept of two's complement, a good place to start would be at the Wikipedia page discussing the topic: (http://en.wikipedia.org/wiki/Two's_complement). It is important to know that bitwise operators can only be applied to integers and long integers. Let's take a look at the different bitwise operators that are available to us (Table 3-4), and then we'll go through a few examples.

Table 3-4. Bitwise Operators

Operator	Description
&	Bitwise and operator copies a bit to the result if a bit appears in both operands
||	Bitwise or operator copies a bit to the result if it exists in either of the operands
^	Bitwise xor operator copies a bit to the result if it exists in only one operand
~	Bitwise negation operator flips the bits, and returns the exact opposite of each bit

Suppose we have a couple of numbers in binary format and we would like to work with them using the bitwise operators. Let's work with the numbers 14 and 27. The binary (two's complement) representation of the number 14 is 00001110, and for 27 it is 00011011. The bitwise operators look at each 1 and 0 in the binary format of the number and perform their respective operations, and then return a result. Python does not return the bits, but rather the integer value of the resulting bits. In the following examples, we take the numbers 14 and 27 and work with them using the bitwise operators.

Listing 3-9. Bitwise Operator Examples

```
>>> 14 & 27
10
```

```
>>> 14 | 27
31
>>> 14 ^ 27
21
>>> ~14
-15
>>> ~27
-28
```

To summarize the examples above, let's work through the operations using the binary representations for each of the numbers.

```
14 & 27 = 00001110 and 00011011 = 00001010 (The integer 10)
14 | 27 = 00001110 or 000110011 = 00011111 (The integer 31)
14 ^ 27 = 00001110 xor 000110011 = 00010101 (The integer 21)
~14 = 00001110 = 11110001 (The integer -15)
```

The shift operators (see Table 3-5) are similar in that they work with the binary bit representation of a number. The left shift operator moves the left operand's value to the left by the number of bits specified by the right operand. The right shift operator does the exact opposite as it shifts the left operand's value to the right by the number of bits specified by the right operand. Essentially this translates to the left shift operator multiplying the operand on the left by the number two as many times as specified by the right operand. The opposite holds true for the right shift operator that divides the operand on the left by the number two as many times as specified by the right operand.

Table 3-5. Shift Operators

x<<n	Shift left (The equivalent of multiplying the number x by 2, n times)
x>>n	Shift right (The equivalent of dividing the number x by 2, n times)

More specifically, the left shift operator (<<) will multiply a number by two n times, n being the number that is to the right of the shift operator. The right shift operator will divide a number by two n times, n being the number to the right of the shift operator. The __future__division import does not make a difference in the outcome of such operations.

Listing 3-10. Shift Operator Examples

```
# Shift left, in this case  3*2
>>> 3<<1
6
# Equivalent of 3*2*2
>>> 3<<2
12
# Equivalent of 3*2*2*2*2*2
>>> 3<<5
96
# Shift right
```

```
# Equivalent of 3/2
>>> 3>>1
1
# Equivalent of 9/2
>>> 9>>1
4
# Equivalent of 10/2
>>> 10>>1
5
# Equivalent of 10/2/2
>>> 10>>2
2
```

While bitwise operators are not the most commonly used operators, they are good to have on hand. They are especially important if you are working in mathematical situations.

Augmented Assignment

Augmented assignment operators (see Table 3-6) combine an operation with an assignment. They can be used to do things like assign a variable to the value it previously held, modified in some way. While augmented assignment can assist in coding concisely, some say that too many such operators can make code more difficult to read.

Listing 3-11. Augmented Assignment Code Examples

```
>>> x = 5
>>> x
5
# Add one to the value of x and then assign that value to x
>>> x+=1
>>> x
6
# Multiply the value of x by 5 and then assign that value to x
>>> x*=5
>>> x
30
```

Table 3-6. Augmented Assignment Operators

Operator	Equivalent
a += b	a = a + b
a -= b	a = a – b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b
a //= b	a = a // b
a **= b	a = a** b
a &= b	a = a & b

a |= b	a = a | b
a ^= b	a = a ^ b
a >>= b	a = a >> b
a <<= b	a = a << b

Boolean Expressions

Evaluating two or more values or expressions also uses a similar syntax to that of other languages, and the logic is quite the same. Note that in Python, True and False are very similar to constants in the Java language. True actually represents the number 1, and False represents the number 0. One could just as easily code using 0 and 1 to represent the Boolean values, but for readability and maintenance the True and False “constants” are preferred. Java developers, make sure that you capitalize the first letter of these two words as you will receive an ugly NameError if you do not.

Boolean properties are not limited to working with int and bool values, but they also work with other values and objects. For instance, simply passing any non-empty object into a Boolean expression will evaluate to True in a Boolean context. This is a good way to determine whether a string contains anything. See Table 3-7.

Listing 3-12. Testing a String

```
>>> mystr = ''
>>> if mystr:
...     'Now I contain the following: %s' % (mystr)
... else:
...     'I do not contain anything'
...
'I do not contain anything'
>>> mystr = 'Now I have a value'
>>> if mystr:
...     'Now I contain the following: %s' % (mystr)
... else:
...     'I do not contain anything'
...
'Now I contain the following: Now I have a value'
```

Table 3-7. Boolean Conditionals

Conditional	Logic
and	In an x and y evaluation, if x evaluates to false then its value is returned, otherwise y is evaluated and the resulting value is returned
or	In an x or y evaluation, if x evaluates to true then its value is returned, otherwise y is evaluated and the resulting value is returned
not	In a not x evaluation, if not x, we mean the opposite of x

As with all programming languages, there is an order of operations for deciding what operators are evaluated first. For instance, if we have an expression a + b *c, then which operation would take place first? The order of operations for Python is shown in Table 3-8 with those operators that receive the highest precedence shown first, and those with the lowest shown last. Repeats of the same operator are grouped from left to the right with the exception of the power (*) operator.

Table 3-8. Python Order of Operations

Operator Precedence from Highest to Lowest	Name
+var, -var, ~var	Unary Operations
**	Power Operations
*, /, //, %	Multiplication, Division, Floor Division, Modulo
+, -	Addition, Subtraction
<<, >>	Left and Right Shift
&	Bitwise And
^	Bitwise Exclusive Or
|	Bitwise Or
<, >, <=. >= , <>	Comparison Operators
==, != , is, is not, in, not in	Equality and Membership
and, or, not	Boolean Conditionals

An important note is that when working with Boolean conditionals, ‘and’ and ‘or’ group from the left to the right. Let’s take a look at a few examples.

Listing 3-13. Order of Operations Examples

```
# Define a few variables
>>> x = 10
>>> y = 12
>>> z = 14
# (y*z) is evaluated first, then x is added
>>> x + y * z
178
# (x * y) is evaluated first, then z is subtracted from the result
>>> x * y - z
106
# When chaining comparisons, a logical 'and' is implied.  In this
# case, x < y and y <= z and z > x
>>> x < y <= z > x
True
#  (2 * 0) is evaluated first and since it is False or zero, it is returned
>>> 2 * 0 and 5 + 1
0
# (2 * 1) is evaluated first, and since it is True or not zero, the (5 + 1)
is evaluated and
# returned
>>> 2 * 1 and 5 + 1
```

```
6
# x is returned if it is True, otherwise y is returned if it is False.  If
neither
# of those two conditions occur, then z is returned.
>>> x or (y and z)
10
# In this example, the (7 - 2) is evaluated and returned because of the 'and'
'or'
# logic
>>> 2 * 0 or ((6 + 8) and (7 - 2))
5
# In this case, the power operation is evaluated first, and then the addition
>>> 2 ** 2 + 8
12
```

Conversions

There are a number of conversion functions built into the language in order to help conversion of one data type to another (see Table 3-9). While every data type in Jython is actually a class object, these conversion functions will really convert one class type into another. For the most part, the built-in conversion functions are easy to remember because they are primarily named after the type to which you are trying to convert.

Table 3-9. Conversion Functions

Function	Description
chr(value)	Converts integer to a character
complex(real [,imag])	Produces a complex number
dict(sequence)	Produces a dictionary from a given sequence of (key, value) tuples
eval(string)	Evaluates a string to return an object…useful for mathematical computations. Note: This function should be used with extreme caution as it can pose a security hazard if not used properly.
float(value)	Converts number to float
frozenset(set) |Converts a set into a frozen set	
hex(value)	Converts an integer into a string representing that number in hex
int(value [, base])	Converts to an integer using a base if a string is given
list(sequence)	Converts a given sequence into a list
long(value [, base])	Converts to a long using a base if a string is given
oct(value)	Converts an integer to a string representing that number as an octal
ord(value)	Converts a character into its integer value
repr(value)	Converts object into an expression string. Same as enclosing expression in reverse quotes ($x + y$). Returns a string containing a printable and evaluable representation of the object
set(sequence)	Converts a sequence into a set

str(value)	Converts an object into a string Returns a string containing a printable representation of the value, but not an evaluable string
tuple(sequence)	Converts a given sequence to a tuple
unichr(value)	Converts integer to a Unicode character

Listing 3-14. Conversion Function Examples

```
# Return the character representation of the integers
>>> chr(4)
'\x04'
>>> chr(10)
'\n'
# Convert intger to float
>>> float(8)
8.0
# Convert character to its integer value
>>> ord('A')
65
>>> ord('C')
67
>>> ord('z')
122
# Use repr() with any object
>>> repr(3.14)
'3.14'
>>> x = 40 * 5
>>> y = 2**8
>>> repr((x, y, ('one','two','three')))
"(200, 256, ('one', 'two', 'three'))"
```

The following is an example of using the eval() functionality as it is perhaps the one conversion function for which an example helps to understand. Again, please note that using the eval() function can be dangerous and impose a security threat if used incorrectly. If using the eval() function to accept text from a user, standard security precautions should be set into place to ensure that the string being evaluated is not going to compromise security.

Listing 3-15. Example of eval()

```
# Suppose keyboard input contains an expression in string format (x * y)
>>> x = 5
>>> y = 12
>>> keyboardInput = 'x * y'
# We should provide some security checks on the keyboard input here to
# ensure that the string is safe for evaluation. Such a task is out of scope
# for this chapter, but it is good to note that comparisons on the keyboard
# input to check for possibly dangerous code should be performed prior to
# evaluation.
>>> eval(keyboardInput)
60
```

Using Expressions to Control Program Flow

As you've learned in previous references in this book, the statements that make up programs in Python are structured with attention to spacing, order, and technique. Each section of code must be consistently spaced as to set each control structure apart from others. One of the great advantages to Python's syntax is that the consistent spacing allows for delimiters such as the curly braces {} to go away. For instance, in Java one must use curly braces around a for loop to signify a start and an end point. Simply spacing a for loop in Python correctly takes place of the braces. Convention and good practice adhere to using four spaces of indentation per statement throughout the entire program. For more information on convention, please see PEP 8, Style Guide for Python Code (www.python.org/dev/peps/pep-0008/). Follow this convention along with some control flow and you're sure to develop some easily maintainable software.

if-elif-else Statement

The standard Python if-elif-else conditional statement is used in order to evaluate expressions and branch program logic based upon the outcome. An if-elif-else statement can consist of any expressions we've discussed previously. The objective is to write and compare expressions in order to evaluate to a True or False outcome. As shown in Chapter 1, the logic for an if-elif-else statement follows one path if an expression evaluates to True, or a different path if it evaluates to False.

You can chain as many if-else expressions together as needed. The combining if-else keyword is elif, which is used for every expression in between the first and the last expressions within a conditional statement.

The elif portion of the statement helps to ensure better readability of program logic. Too many if statements nested within each other can lead to programs that are difficult to maintain. The initial if expression is evaluated, and if it evaluates to False, the next elif expression is evaluated, and if it evaluates to False then the process continues. If any of the if or elif expressions evaluate to True then the statements within that portion of the if statement are processed. Eventually if all of the expressions evaluate to False then the final else expression is evaluated.

These next examples show a few ways for making use of a standard if-elif-else statement. Note that any expression can be evaluated in an if-elif-else construct. These are only some simplistic examples, but the logic inside the expressions could become as complex as needed.

Listing 3-16. Standard if-elif-else

```
# terminal symbols are left out of this example so that you can see the
precise indentation
pi =3.14
x = 2.7 * 1.45
if x == pi:
    print 'The number is pi'
```

```
elif x > pi:
    print 'The number is greater than pi'
else:
    print 'The number is less than pi'
```

Empty lists or strings will evaluate to False as well, making it easy to use them for comparison purposes in an if-elif-else statement.

Listing 3-17. Evaluate Empty List

```
# Use an if-statement to determine whether a list is empty
# Suppose mylist is going to be a list of names
>>> mylist = []
>>> if mylist:
...     for person in mylist:
...         print person
... else:
...     print 'The list is empty'
...
The list is empty
```

while Loop

Another construct that we touched upon in Chapter 1 was the loop. Every programming language provides looping implementations, and Python is no different. To recap, the Python language provides two main types of loops known as the while and the for loop.

The while loop logic follows the same semantics as the while loop in Java. The while loop evaluates a given expression and continues to loop through its statements until the results of the expression no longer hold true and evaluate to False. Most while loops contain a comparison expression such as x <= y or the like, in this case the expression would evaluate to False when x becomes greater than y. The loop will continue processing until the expression evaluates to False. At this time the looping ends and that would be it for the Java implementation. Python on the other hand allows an else clause which is executed when the loop is completed.

Listing 3-18. Python while Statement

```
>>> x = 0
>>> y = 10
>>> while x <= y:
...     print 'The current value of x is: %d' % (x)
...     x += 1
... else:
...     print 'Processing Complete...'
...
The current value of x is: 0
The current value of x is: 1
The current value of x is: 2
```

```
The current value of x is: 3
The current value of x is: 4
The current value of x is: 5
The current value of x is: 6
The current value of x is: 7
The current value of x is: 8
The current value of x is: 9
The current value of x is: 10
Processing Complete...
```

This else clause can come in handy while performing intensive processing so that we can inform the user of the completion of such tasks. It can also be handy when debugging code, or when some sort of cleanup is required after the loop completes

Listing 3-19. Resetting Counter Using with-else

```
>>> total = 0
>>> x = 0
>>> y = 20
>>> while x <= y:
...     total += x
...     x += 1
... else:
...     print total
...     total = 0
...
210
```

continue Statement

The continue statement is to be used when you are within a looping construct, and you have the requirement to tell Python to continue processing past the rest of the statements in the current loop. Once the Python interpreter sees a continue statement, it ends the current iteration of the loop and goes on to continue processing the next iteration. The continue statement can be used with any for or while loop.

Listing 3-20. Continue Statement

```
# Iterate over range and print out only the positive numbers
>>> x = 0
>>> while x < 10:
...     x += 1
...     if x % 2 != 0:
...         continue
...     print x
...
2
4
6
8
```

```
10
```

In this example, whenever x is odd, the ‘continue’ causes execution to move on to the next iteration of the loop. When x is even, it is printed out.

break Statement

Much like the continue statement, the break statement can be used inside of a loop. We use the break statement in order to stop the loop completely so that a program can move on to its next task. This differs from continue because the continue statement only stops the current iteration of the loop and moves onto the next iteration. Let’s check it out:

Listing 3-21. Break Statement

```
>>> x = 10
>>> while True:
...     if x == 0:
...         print 'x is now equal to zero!'
...         break
...     if x % 2 == 0:
...         print x
...     x -= 1
...
10
8
6
4
2
x is now equal to zero!
```

In the previous example, the loop termination condition is always True, so execution only leaves the loop when a break is encountered. If we are working with a break statement that resides within a loop that is contained in another loop (nested loop construct), then only the inner loop will be terminated.

for Loop

The for loop can be used on any iterable object. It will simply iterate through the object and perform some processing during each pass. Both the break and continue statements can also be used within the for loop. The for statement in Python also differs from the same statement in Java because in Python we also have the else clause with this construct. Once again, the else clause is executed when the for loop processes to completion without any break intervention or raised exceptions. Also, if you are familiar with pre-Java 5 for loops then you will love the Python syntax. In Java 5, the syntax of the for statement was adjusted a bit to make it more in line with syntactically easy languages such as Python.

Listing 3-22. Comparing Java and Python for-loop

Example of Java for-loop (pre Java 5)

```
for(x = 0; x <= myList.size(); x++){
// processing statements iterating through myList
System.out.println("The current index is: " + x);
}
```

Listing 3-23. Example of Python for-loop

```
my_list = [1,2,3,4,5]
>>> for value in my_list:
# processing statements using value as the current item in my_list
...     print 'The current value is %s' % (value)
...
The current value is 1
The current value is 2
The current value is 3
The current value is 4
The current value is 5
```

As you can see, the Python syntax is a little easier to understand, but it doesn't really save too many keystrokes at this point. We still have to manage the index (x in this case) by ourselves by incrementing it with each iteration of the loop. However, Python does provide a built-in function that can save us some keystrokes and provides a similar functionality to that of Java with the automatically incrementing index on the for loop. The enumerate(sequence) function does just that. It will provide an index for our use and automatically manage it for us.

Listing 3-24. Enumerate() Functionality

```
>>> myList = ['jython','java','python','jruby','groovy']
>>> for index, value in enumerate(myList):
...     print index, value
...
0 jython
1 java
2 python
3 jruby
4 groovy
```

If we do not require the use of an index, it can be removed and the syntax can be cleaned up a bit.

```
>>> myList = ['jython', 'java', 'python', 'jruby', 'groovy']
>>> for item in myList:
...     print item
...
jython
java
python
jruby
```

```
groovy
```

Now we have covered the program flow for conditionals and looping constructs in the Python language. However, good programming practice will tell you to keep it as simple as possible or the logic will become too hard to follow. In practicing proper coding techniques, it is also good to know that lists, dictionaries, and other containers can be iterated over just like other objects. Iteration over containers using the for loop is a very useful strategy. Here is an example of iterating over a dictionary object.

Listing 3-25. Iteration Over Containers

```
# Define a dictionary and then iterate over it to print each value
>>> my_dict = {'Jython':'Java', 'CPython':'C',
'IronPython':'.NET','PyPy':'Python'}
>>> for key in my_dict:
...     print key
...
Jython
IronPython
CPython
PyPy
```

It is useful to know that we can also obtain the values of a dictionary object via each iteration by calling my_dict.values().

Example Code

Let's take a look at an example program that uses some of the program flow which was discussed in this chapter. The example program simply makes use of an external text file to manage a list of players on a sports team. You will see how to follow proper program structure and use spacing effectively in this example. You will also see file utilization in action, along with utilization of the raw_input() function.

Listing 3-26. # import os module

```
import os

# Create empty dictionary
player_dict = {}
# Create an empty string
enter_player = ''

# Enter a loop to enter inforation from keyboard
while enter_player.upper() != 'X':

    print 'Sports Team Administration App'

    # If the file exists, then allow us to manage it, otherwise force
creation.
```

```
    if os.path.isfile('players.txt'):
        enter_player = raw_input("Would you like to create a team or manage
an existing team?\n (Enter 'C' for create, 'M' for manage, 'X' to exit) ")
    else:
        # Force creation of file if it does not yet exist.
        enter_player = 'C'

    # Check to determine which action to take.  C = create, M = manage, X =
Exit and Save
    if enter_player.upper() == 'C':

    # Enter a player for the team
        print 'Enter a list of players on our team along with their position'
        enter_cont = 'Y'

        #  While continuing to enter new player's, perform the following
        while enter_cont.upper() == 'Y':
            # Capture keyboard entry into name variable
            name = raw_input('Enter players first name: ')
            # Capture keyboard entry into position variable
            position = raw_input('Enter players position: ')
            # Assign position to a dictionary key of the player name
            player_dict[name] = position
            enter_cont = raw_input("Enter another player? (Press 'N' to exit
or 'Y' to continue)")
        else:
            enter_player = 'X'

    # Manage player.txt entries
    elif enter_player.upper() == 'M':

        # Read values from the external file into a dictionary object
        print
        print 'Manage the Team'
        # Open file and assign to playerfile
        playerfile = open('players.txt','r')
        # Use the for-loop to iterate over the entries in the file
        for player in playerfile:
            # Split entries into key/value pairs and add to list
            playerList = player.split(':')
            # Build dictionary using list values from file
            player dict[playerList[0]] = playerList[1]
        # Close the file
        playerfile.close()

        print 'Team Listing'
        print '++++++++++++'

        # Iterate over dictionary values and print key/value pairs
        for i, player in enumerate(player_dict):
            print 'Player %s Name: %s -- Position: %s' %(i, player,
player_dict[player])

else:
    # Save the external file and close resources
    if player_dict:
```

```
        print 'Saving Team Data...'
        # Open the file
        playerfile = open('players.txt','w')
        # Write each dictionary element to the file
        for player in player_dict:
            playerfile.write('%s:%s\n' %
(player.strip(),player_dict[player].strip()))
        # Close file
        playerfile.close()
```

This example is packed full of concepts that have been discussed throughout the first three chapters of the book. As stated previously, the concept is to create and manage a list of sport players and their relative positions. The example starts by entering a while() loop that runs the program until the user enters the exit command. Next, the program checks to see if the 'players.txt' file exists. If it does, then the program prompts the user to enter a code to determine the next action to be taken. However, if the file does not exist then the user is forced to create at least one player/position pair in the file.

Continuing on, the program allows the user to enter as many player/position pairs as needed, or exit the program at any time. If the user chooses to manage the player/position list, the program simply opens the 'players.txt' file, uses a for() loop to iterate over each entry within the file. A dictionary is populated with the current player in each iteration of the loop. Once the loop has completed, the file is closed and the dictionary is iterated and printed. Exiting the program forces the else() clause to be invoked, which iterates over each player in the dictionary and writes them to the file.

Unfortunately, this program is quite simplistic and some features could not be implemented without knowledge of functions (Chapter 4) or classes (Chapter 6). A good practice would be to revisit this program once those topics have been covered and simplify as well as add additional functionality.

Summary

All programs are constructed out of statements and expressions. In this chapter we covered details of creating expressions and using them. Expressions can be composed of any number of mathematical operators and comparisons. In this chapter we discussed the basics of using mathematical operators in our programs. The __future__ division topic introduced us to using features from the __future__. We then delved into comparisons and comparison operators.

We ended this short chapter by discussing proper program flow and properly learned about the if statement as well as how to construct different types of loops in Python.

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