Language and Syntax

Elegant is an adjective that is often used to describe the Python language. The word elegant is defined as "pleasingly graceful and stylish in appearance or manner." *Uncomplicated* and *powerful* could also be great words to assist in the description of this language. It is a fact that Python is an elegant language that lets one create powerful applications in an uncomplicated manner. The ability to make reading and writing complex software easier is the objective of all programming languages, and Python does just that.

While we've easily defined the goal of programming languages in a broad sense in paragraph one, we have left out one main advantage of learning the Python programming language: Python has been extended to run on the Java platform, and so it can run anywhere with a JVM. There are also C and .NET versions of Python with multiplatform support. So, Python can run nearly everywhere. In this book, we focus on Jython, the language implementation that takes the elegance, power, and ease of Python and runs it on the JVM.

The Java platform is an asset to the Jython language much like the C libraries are for Python. Jython is able to run just about everywhere, which gives lots of flexibility when deciding how to implement an application. Not only does the Java platform allow for flexibility with regards to application deployment, but it also offers a vast library containing thousands of APIs that are available for use by Jython. Add in the maturity of the Java platform and it becomes easy to see why Jython is such an attractive programming language. The goal, if you will, of any programming language is to grant its developers the same experience that Jython does. Simply put, learning Jython will be an asset to any developer.

As I've mentioned, the Jython language implementation takes Python and runs it on the JVM, but it does much more than that. Once you have experienced the power of programming on the Java platform, it will be difficult to move away from it. Learning Jython not only allows you to run on the JVM, but it also allows you to learn a new way to harness the power of the platform. The language increases productivity as it has an easily understood syntax that reads almost as if it were pseudocode. It also adds dynamic abilities that are not available in the Java language itself.

In this chapter you will learn how to install and configure your environment, and you will also get an overview of those features that the Python language has to offer. This chapter is not intended to delve so deep into the concepts of syntax as to bore you, but rather to give you a quick and informative introduction to the syntax so that you will know the basics and learn the language as you move on through the book. It will also allow you the chance to compare some Java examples with those which are written in Python so you can see some of the advantages this language has to offer.

By the time you have completed this chapter, you should know the basic structure and organization that Python code should follow. You'll know how to use basic language concepts such as defining variables, using reserved words, and performing basic tasks. It will give you a taste of using statements and expressions. As every great program contains comments, you'll

learn how to document single lines of code as well as entire code blocks. As you move through the book, you will use this chapter as a reference to the basics. This chapter will not cover each feature in completion, but it will give you enough basic knowledge to start using the Python language.

The Difference between Jython and Python

Jython is an implementation of the Python language for the Java platform. Throughout this book, you will be learning how to use the Python language, and along the way we will show you where the Jython implementation differs from CPython, which is the canonical implementation of Python written in the C language. It is important to note that the Python language syntax remains consistent throughout the different implementations. At the time of this writing, there are three mainstream implementations of Python. These implementations are: CPython, Jython for the Java platform, and IronPython for the .NET platform. At the time of this writing, CPython is the most prevalent of the implementations. Therefore if you see the word Python somewhere, it could well be referring to that implementation.

This book will reference the Python language in sections regarding the language syntax or functionality that is inherent to the language itself. However, the book will reference the name Jython when discussing functionality and techniques that are specific to the Java platform implementation. No doubt about it, this book will go in-depth to cover the key features of Jython and you'll learn concepts that only adhere the Jython implementation. Along the way, you will learn how to program in Python and advanced techniques.

Developers from all languages and backgrounds will benefit from this book. Whether you are interested in learning Python for the first time or discovering Jython techniques and advanced concepts, this book is a good fit. Java developers and those who are new to the Python language will find specific interest in reading through Part I of this book as it will teach the Python language from the basics to more advanced concepts. Seasoned Python developers will probably find more interest in Part II and Part III as they focus more on the Jython implementation specifics. Often in this reference, you will see Java code compared with Python code.

Installing and Configuring Jython

Before we delve into the basics of the language, we'll learn how to obtain Jython and configure it for your environment. To get started, you will need to obtain a copy of Jython from the official website www.jython.org. Because this book focuses on release 2.5.x, it would be best to visit the site now and download the most recent version of that release. You will see that there are previous releases that are available to you, but they do not contain many of the features which have been included in the 2.5.x series.

Jython implementation maintains consistent features which match those in the Python language for each version. For example, if you download the Jython 2.2.1 release, it will include all of the features that the Python 2.2 release contains. Similarly, when using the 2.5 release you will

have access to the same features which are included in Python 2.5. There are also some extra pieces included with the 2.5 release which are specific to Jython. We'll discuss more about these extra features throughout the book.

Please grab a copy of the most recent version of the Jython 2.5 release. You will see that the release is packaged as a cross-platform executable JAR file. Right away, you can see the obvious advantage of running on the Java platform. . .one installer that works for various platforms. It doesn't get much easier than that! In order to install the Jython language, you will need to have Java 5 or greater installed on your machine. If you do not have Java 5 or greater then you'd better go and grab that from www.java.com and install it before trying to initiate the Jython installer.

You can initiate the Jython installer by simply double-clicking on the JAR file. It will run you through a series of standard installation questions. At one point you will need to determine which features you'd like to install. If you are interested in looking through the source code for Jython, or possibly developing code for the project then you should choose the "All" option to install everything. . .including source. However, for most Jython developers and especially for those who are just beginning to learn the language, I would recommend choosing the "Standard" installation option. Once you've chosen your options and supplied an installation path then you will be off to the races.

In order to run Jython, you will need to invoke the jython.bat executable file on Windows or the jython.sh file on *NIX machines and Mac OS X. That being said, you'll have to traverse into the directory that you've installed Jython where you will find the file. It would be best to place this directory within your PATH environment variable on either Windows, *NIX, or OS X machines so that you can fire up Jython from within any directory on your machine. Once you've done this then you should be able to open up a terminal or command prompt and type "jython" then hit enter to invoke the interactive interpreter. This is where our journey begins! The Jython interactive interpreter is a great place to evaluate code and learn the language. It is a real-time testing environment that allows you to type code and instantly see the result. As you are reading through this chapter, I recommend you open up the Jython interpreter and follow along with the code examples.

Identifiers and Declaring Variables

Every programming language needs to contain the ability to capture or calculate values and store them. Python is no exception, and doing so is quite easy. Defining variables in Python is very similar to other languages such as Java, but there are a few differences that you need to note.

To define a variable in the Python language, you simply name it using an identifier. An identifier is a name that is used to identify an object. The language treats the variable name as a label that points to a value. It does not give any type for the value. Therefore, this allows any variable to hold any type of data. It also allows the ability of having one variable contain of different data types throughout the life cycle of a program. So a variable that is originally assigned with an

integer, can later contain a String. Identifiers in Python can consist of any ordering of letters, numbers, or underscores. However, an identifier must always begin with a non-numeric character value. We can use identifiers to name any type of variable, block, or object in Python. As with most other programming languages, once an identifier is defined, it can be referenced elsewhere in the program.

Once declared, a variable is untyped and can take any value. This is one difference between using a statically typed language such as Java, and using dynamic languages like Python. In Java, you need to declare the type of variable which you are creating, and you do not in Python. It may not sound like very much at first, but this ability can lead to some extraordinary results. Consider the following two listings, lets define a value 'x' below and we'll give it a value of zero.

Listing 1-1. Java – Declare Variable

```
int x = 0;
```

Listing 1-2. Python – Declare Variable

```
x = 0
```

As you see, we did not have to give a type to this variable. We simply choose a name and assign it a value. Since we do not need to declare a type for the variable, we can change it to a different value and type later in the program.

Listing 1-3.

```
x = 'Hello Jython'
```

We've just changed the value of the variable 'x' from a numeric value to a String without any consequences. What really occurred is that we created a new variable 'Hello Jython' and assigned it to the identifier 'x', which in turn lost its reference to 0. This is a key to the dynamic language philosophy. . .change should not be difficult.

Let us take what we know so far and apply it to some simple calculations. Based upon the definition of a variable in Python, we can assign an integer value to a variable, and change it to a float at a later point. For instance:

Listing 1-4.

```
>>> x = 6
>>> y = 3.14
>>> x = x * y
>>> print x
18.84
```

In the previous example, we've demonstrated that we can dynamically change the type of any given variable by simply performing a calculation upon it. In other languages such as Java, we would have had to begin by assigning a float type to the 'x' variable so that we could later change its value to a float. Not here, Python allows us to bypass type constriction and gives us an easy way to do it.

Reserved Words

There are a few more rules to creating identifiers that we must follow in order to adhere to the Python language standard. Certain words are not to be used as identifiers as the Python language reserves them for performing a specific role within our programs. These words which cannot be used are known as reserved words. If we try to use one of these reserved words as an identifier, we will see a SyntaxError thrown as Python wants these reserved words as its own.

There are no symbols allowed in identifiers. Yes, that means the Perl developers will have to get used to defining variables without the $.

Table 1-1 lists all of the Python language reserved words:

Table 1-1. Reserved Words

and	assert	break	class	continue
def	del	elif	else	except
exec	finally	for	from	global
or	pass	print	raise	return
try	while	with	yield	

It is important to take care when naming variables so that you do not choose a name that matches one of the module names from the standard library.

Coding Structure

Another key factor in which Python differs from other languages is its coding structure. Back in the day, we had to develop programs based upon a very strict structure such that certain pieces must begin and end within certain punctuations. Python uses indentation rather than punctuation to define the structure of code. Unlike languages such as Java that use brackets to open or close a code block, Python uses spacing as to make code easier to read and also limit unnecessary symbols in your code. It strictly enforces ordered and organized code but it lets the programmer define the rules for indentation, although a standard of four characters exists.

For instance, let's jump ahead and look at a simple 'if' statement. Although you may not yet be familiar with this construct, I think you will agree that it is easy to determine the outcome. Take a look at the following block of code written in Java first, and then we'll compare it to the Python equivalent.

Listing 1-5. Java if-statement

```
x = 100;
if(x > 0){
System.out.println("Wow, this is Java");
} else {
System.out.println("Java likes curly braces");
}
```

Now, let's look at a similar block of code written in Python.

Listing 1-6. Python if-statement

```
x = 100
if x > 0:
    print 'Wow, this is elegant'
else:
    print 'Organization is the key'
```

Okay, this is cheesy but we will go through it nonetheless as it is demonstrating a couple of key points to the Python language. As you see, the Python program evaluates if the value of the variable 'x' is greater than zero. If so, it will print 'Wow, this is elegant.' Otherwise, it will print 'Organization is the key.' Look at the indentation which is used within the 'if' block. This particular block of code uses four spaces to indent the 'print' statement from the initial line of the block. Likewise, the 'else' jumps back to the first space of the line and its corresponding implementation is also indented by four spaces. This technique must be adhered to throughout an entire Python application. By doing so, we gain a couple of major benefits: easy-to-read code and no need to use curly braces. Most other programming languages such as Java use a bracket "[" or curly brace "{" to open and close a block of code. There is no need to do so when using Python as the spacing takes care of this for you. Less code = easier to read and maintain. It is also worth noting that the Java code in the example could have been written on one line, or worse, but we chose to format it nicely.

Python ensures that each block of code adheres to its defined spacing strategy in a consistent manner. What is the defined spacing strategy? You decide. As long as the first line of a code block is out-dented by at least one space, the rest of the block can maintain a consistent indentation, which makes code easy to read. Many argue that it is the structuring technique that Python adheres to which makes them so easy to read. No doubt, adhering to a standard spacing throughout an application makes for organization. As mentioned previously, the Python standard spacing technique is to use four characters for indentation. If you adhere to these standards then your code will be easy to read and maintain in the future. Your brain seems hard-wired to adhering to some form of indentation, so Python and your brain are wired up the same way.

Operators

The operators that are used by Python are very similar to those used in other languages...straightforward and easy to use. As with any other language, you have your normal operators such as +, -, *, and /, which are available for performing calculations. As you can see from the following examples, there is no special trick to using any of these operators.

Listing 1-7. Performing Integer-based Operations

```
>>> x = 9
>>> y = 2
>>> x + y
11
>>> x - y
7
>>> x * y
18
>>> x / y
4
```

Perhaps the most important thing to note with calculations is that if you are performing calculations based on integer values then you will receive a rounded result. If you are performing calculations based upon floats then you will receive float results, and so on.

Listing 1-8. Performing Float-based Operations

```
>>> x = 9.0
>>> y = 2.0
>>> x + y
11.0
>>> x - y
7.0
>>> x * y
18.0
>>> x / y
4.5
```

It is important to note this distinction because as you can see from the differences in the results of the division (/) operations in Listings 1-7 and 1-8, we have rounding on the integer values and not on the float. A good rule of thumb is that if your application requires precise calculations to be defined, then it is best to use float values for all of your numeric variables, or else you will run into a rounding issue. In Python 2.5 and earlier, integer division always rounds down, producing the floor as the result. In Python 2.2, the // operator was introduced which is another way to obtain the floor result when dividing integers or floats. This operator was introduced as a segue way for changing integer division in future releases so that the result would be a *true* division. In Chapter 3, we'll discuss division using a technique that always performs *true* division.

Expressions

Expressions are just what they sound like. They are a piece of Python code that can be evaluated and produces a value. Expressions are not instructions to the interpreter, but rather a combination of values and operators that are evaluated. If we wish to perform a calculation based upon two variables or numeric values then we are producing an expression.

Listing 1-9. Examples of Expressions

```
>>> x + y
>>> x - y
>>> x * y
>>> x / y
```

The examples of expressions that are shown above are very simplistic. Expressions can be made to be very complex and perform powerful computations. They can be combined together to produce complex results.

Functions

Oftentimes it is nice to take suites of code that perform specific tasks and extract them into their own unit of functionality so that the code can be reused in numerous places without retyping each time. A common way to define a reusable piece of code is to create a function. Functions are named portions of code that perform that usually perform one or more tasks and return a value. In order to define a function we use the *def* statement.

The *def* statement will become second nature for usage throughout any Python programmer's life. The *def *statement is used to define a function. Here is a simple piece of pseudocode that shows how to use it.

Listing 1-10.

```
def my_function_name(parameter_list):
    implementation
```

The pseudocode above demonstrates how one would use the *def* statement, and how to construct a simple function. As you can see, *def* precedes the function name and parameter list when defining a function.

Listing 1-11.

```
>>> def my_simple_function():
... print 'This is a really basic function'
...
>>> my_simple_function()
This is a really basic function
```

This example is about the most basic form of function that can be created. As you can see, the function contains one line of code which is a print statement. We will discuss the print statement in more detail later in this chapter; however, all you need to know now is that it is used to print some text to the screen. In this case, we print a simple message whenever the function is called.

Functions can accept parameters, or other program variables, that can be used within the context of the function to perform some task and return a value.

Listing 1-12.

```
>>> def multiply_nums(x, y):
... return x * y
...
>>> multiply_nums(25, 7)
175
```

As seen above, parameters are simply variables that are assigned when the function is called. Specifically, we assign 25 to *x* and 7 to *y* in the example. The function then takes *x* and *y,* performs a calculation and returns the result.

Functions in Python are just like other variables and they be passed around as parameters to other functions if needed. Here we show a basic example of passing one function to another function. We'll pass the *multiply_nums* function into the function below and then use it to perform some calculations.

Listing 1-13.

```
>>> def perform_math(oper):
... return oper(5, 6)
...
>>> perform_math(multiply_nums)
30
```

Although this example is very basic, you can see that another function can be passed as a parameter and then used within another function. For more detail on using *def* and functions, please take a look at Chapter 4, which is all about functions.

Classes

Python is an object-oriented programming language. which means that everything in the language is an object of some type. Much like building blocks are used for constructing buildings, each object in Python can be put together to build pieces of programs or entire programs. This section will give you a brief introduction to Python classes, which are one of the keys to object orientation in this language.

Classes are defined using the *class* keyword. Classes can contain functions, methods, and variables. Methods are just like functions in that the *def* keyword is used to create them, and they accept parameters. The only difference is that methods take a parameter known as *self* that refers to the object to which the method belongs. Classes contain what is known as an initializer method, and it is called automatically when a class is instantiated. Let's take a look at a simple example and then explain it.

Listing 1-14. Simple Python Class

```
>>> class my_object:
... def __init__(self, x, y):
... self.x = x
... self.y = y
...
... def mult(self):
... print self.x * self.y
...
... def add(self):
... print self.x + self.y
...
>>> obj1 = my_object(7, 8)
>>> obj1.mult()
56
>>> obj1.add()
15
```

In this class example, we define a class named *my_object*. The class accepts two parameters, *x* and *y*. A class initializer method is named *__init__()*, and it is used to initialize any values that may be used in the class. An initializer also defines what values can be passed to a class in order to create an object. You can see that each method and function within the class accepts the *self* argument. The *self* argument is used to refer to the object itself, this is how the class shares variables and such. The *self* keyword is similar to *this* in Java code. The *x* and *y* variables in the example are named *self.x* and *self.y* in the initializer, that means that they will be available for use throughout the entire class. While working with code within the object, you can refer to these variables as *self.x*and *self.y*. If you create the object and assign a name to it such as *obj1*, then you can refer to these same variables as *obj1.x* and *obj1.y*.

As you can see, the class is called by passing the values 7 and 8 to it. These values are then assigned to *x* and *y* within the class initializer method. We assign the class object to an identifier that we call *obj1*. The *obj1* identifier now holds a reference to *my_object()* with the values we've passed it. The *obj1*identifier can now be used to call methods and functions that are defined within the class.

For more information on classes, please see Chapter 6, which covers object orientation in Python. Classes are very powerful and the fundamental building blocks for making larger programs.

Statements

When we refer to statements, we are really referring to a line of code that contains an instruction that does something. A statement tells the Python interpreter to perform a task. Ultimately, programs are made up of a combination of expressions and statements. In this section, we will take a tour of statement keywords and learn how they can be used.

Let's start out by listing each of these different statement keywords, and then we will go into more detail about how to use each of them with different examples. I will not cover every statement keyword in this section as some of them are better left for later in the chapter or the book, but you should have a good idea of how to code an action which performs a task after reading through this section. While this section will provide implementation details about the different statements, you should refer to later chapters to find advanced uses of these features.

Table 1-2. Statement Keywords

if-elif-else	for
while	continue
break	try-except-finally
assert	def
print	del
raise	import

Now that we've taken a look at each of these keywords, it is time to look at each of them in detail. It is important to remember that you cannot use any of these keywords for variable names.

if-elif-else Statement

The if statement simply performs an evaluation on an expression and does different things depending on whether it is *True* or *False*. If the expression evaluates to *True* then one set of statements will be executed, and if it evaluates to *False* a different set of statements will be executed. If statements are quite often used for branching code into one direction or another based upon certain values which have been calculated or provided in the code.

Pseudocode would be as follows:

Listing 1-15.

```
if <an expression to test>:
    perform an action
else:
    perform a different action
```

Any number of *if/else* statements can be linked together in order to create a logical code branch. When there are multiple expressions to be evaluated in the same statement, then the *elif* statement can be used to link these expressions together*. *Note that each set of*

*statements within an *if-elif-*else statement must be indented with the conditional statement out-dented and the resulting set of statements indented. Remember, a consistent indentation must be followed throughout the course of the program. The *if* statement is a good example of how well the consistent use of indention helps readability of a program. If you are coding in Java for example, you can space the code however you'd like as long as you use the curly braces to enclose the statement. This can lead to code that is very hard to read…the indentation which Python requires really shines through here.

Listing 1-16. Example of if statement

```
>>> x = 3
>>> y = 2
>>> if x == y:
... print 'x is equal to y'
... elif x > y:
... print 'x is greater than y'
... else:
... print 'x is less than y'
...
x is greater than y
```

While the code is simple, it demonstrates that using an *if* statement can result in branching code logic.

print Statement

The *print* statement is used to display program output onto the screen (you've already seen it in action several times). It can be used for displaying messages, which are printed from within a program, and also for printing values, which may have been calculated. In order to display variable values within a print statement, we need to learn how to use some of the formatting options that are available to Python. This section will cover the basics of using the print statement along with how to display values by formatting your strings of text.

In the Java language, we need to make a call to the System library in order to print something to the command line. In Python, this can be done with the use of the *print* statement. The most basic use of the *print* statement is to display a line of text. In order to do so, you simply enclose the text that you want to display within single or double quotes. Take a look at the following example written in Java, and compare it to the example immediately following which is rewritten in Python. I think you'll see why the *print* statement in Python makes life a bit easier.

Listing 1-17. Java Print Output Example

```
System.out.println("This text will be printed to the command line");
```

Listing 1-18. Python Print Output Example

```
print 'This text will be printed to the command line'
```

As you can see from this example, printing a line of text in Python is very straightforward. We can also print variable values to the screen using the *print*statement.

Listing 1-19.

```
>>> my_value = 'I love programming in Jython'
>>> print my_value
I love programming in Jython
```

Once again, very straightforward in terms of printing values of variables. Simply place the variable within a print statement. We can also use this technique in order to append the values of variables to a line of text. In order to do so, just place the concatenation operator (+) in between the String of text which you would like to append to, and the variable you'd like to append.

Listing 1-20.

```
>>> print 'I like programming in Java, but ' + my_value
I like programming in Java, but I love programming in Jython
```

This is great and all, but really not useful if you'd like to properly format your text or work with *int* values. After all, the Python parser is treating the (+) operator as a concatenation operator in this case...not as an addition operator. Python bases the result of the (+) operator on the type of the first operand. If you try to append a numeric value to a String you will end up with an error.

Listing 1-21.

```
>>> z = 10
>>> print 'I am a fan of the number: ' + z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

As you can see from this example, Python does not like this trick very much. So in order to perform this task correctly we will need to use some of the aforementioned Python formatting options. This is easy and powerful to do, and it allows one to place any content or value into a print statement. Before you see an example, let's take a look at some of the formatting operators and how to choose the one that you need.

- %s - String
- %d - Decimal
- %f - Float

If you wish to include the contents of a variable or the result of an expression in your *print *statement, you'll use the following syntax:

Listing 1-22.

```
print 'String of text goes here %d %s %f' % (decimalValue, stringValue,
floatValue)
```

In the pseudocode above (if we can really have pseudocode for print statements), we wish to print the string of text, which is contained within the single quotes, but also have the values of the variables contained where the formatting operators are located. Each of the formatting operators, which are included in the string of text, will be replaced with the corresponding values from those variables at the end of the print statement. The % symbol between the line of text and the list of variables tells Python that it should expect the variables to follow, and that the value of these variables should be placed within the string of text in their corresponding positions.

Listing 1-23.

```
>>> string_value = 'hello world'
>>> float_value = 3.998
>>> decimal_value = 5
>>> print 'Here is a test of the print statement using the values: %d, %s,
and %f' % (decimal_value, string_value, float_value)
Here is a test of the print statement using the values: 5, hello world, and
3.998000
```

As you can see this is quite easy to use and very flexible. The next example shows that we also have the option of using expressions as opposed to variables within our statement.

Listing 1-24.

```
>>> x = 1
>>> y = 2
>>> print 'The value of x + y is: %d' % (x + y)
The value of x + y is: 3
```

The formatting operator that is used determines how the output looks, it does not matter what type of input is passed to the operator. For instance, we could pass an integer or float to %s and it would print just fine, but it will in effect be turned into a string in its exact format. If we pass an integer or float to %d or %f, it will be formatted properly to represent a decimal or float respectively. Take a look at the following example to see the output for each of the different formatting operators.

Listing 1-25.

```
>>> x = 2.3456
>>> print '%s' % x
2.3456
>>> print '%d' % x
2
>>> print '%f' % x
2.345600
```

Another useful feature of the print statement is that it can be used for debugging purposes. If we simply need to find out the value of a variable during processing then it is easy to display using the *print *statement. Using this technique can often really assist in debugging and writing your code.

try-except-finally

The *try-except-finally *is the supported method for performing error handling within a Python application. The idea is that we try to run a piece of code and if it fails then it is caught and the error is handled in a proper fashion. We all know that if someone is using a program that displays an ugly long error message, it is not usually appreciated. Using the *try-except-finally *statement to properly catch and handle our errors can mitigate an ugly program dump.

This approach is the same concept that is used within many languages, including Java. There are a number of defined *error types* within the Python programming language and we can leverage these error types in order to facilitate the *try-except-finally *process. When one of the defined error types is caught, then a suite of code can be coded for handling the error, or can simply be logged, ignored, and so on. The main idea is to avoid those ugly error messages and handle them neatly by displaying a formatted error message or performing another process.

Listing 1-26.

```
>>> # Suppose we've calculated a value and assigned it to x
>>> x
8.97
>>> y = 0
>>> try:
... print 'The rocket trajectory is: %f' % (x/y)
... except:
... print 'Houston, we have a problem....

Houston, we have a problem.
```

If there is an exception that is caught within the block of code and we need a way to perform some cleanup tasks, we would place the cleanup code within the*finally* clause of the block. All code within the *finally* clause is always invoked before the exception is raised. The details of this topic can be read about more in Chapter 7. In the next section, we'll take a look at the raise statement, which we can use to raise exceptions at any point in our program.

raise Statement

As mentioned in the previous section, the *raise* statement is used to throw or "raise" an exception in Python. We know that a *try-*except clause is needed if Python decides to raise an exception, but what if you'd like to raise an exception of your own? You can place a *raise* statement anywhere that you wish to raise a specified exception. There are a number of defined exceptions within the language which can be raised. For instance, NameError is raised when a specific piece of code is undefined or has no name. For a complete list of exceptions in Python, please visit Chapter 7.

Listing 1-27.

```
>>> raise NameError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError
```

If you wish to specify your own message within a *raise* then you can do so by raising a generic Exception, and then specifying your message on the statement as follows.

Listing 1-28.

```
>>> raise Exception('Custom Exception')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Custom Exception
```

import Statement

A program can be made up of one or more suites of code. In order to save a program so that it can be used later, we place the code into files on our computer. Files that contain Python code should contain a *.py* suffix such as my_code.py and so forth. These files are known as modules in the Python world. The *import* statement is used much like it is in other languages, it brings external modules or code into a program so that it can be used. This statement is ultimately responsible for reuse of code in multiple locations. The *import* statement allows us to save code into a flat file or script, and then use it in an application at a later time.

If a class is stored in an external module that is named the same as the class itself, the *import* statement can be used to explicitly bring that class into an application. Similarly, if you wish to import only a specific identifier from another module into your current module, then the specific code can be named within using the syntax *from <<module>> import <<specific code>>. *Time to see some examples.

Listing 1-29.

```
# Import a module named TipCalculator
import TipCalculator
```

```
# Import a function tipCalculator from within a module called
ExternalModule.py
from ExternalModule import tipCalculator
```

When importing modules into your program, you must ensure that the module being imported does not conflict with another name in your current program. To import a module that is named the same as another identifier in your current program, you can use the *as* syntax. In the following example, let's assume that we have defined an external module with the name of *tipCalculator.py* and we want to use it's functionality in our current program. However, we already have a function named *tipCalculator()* within the current program. Therefore, we use the *as* syntax to refer to the *tipCalculator* module.

Listing 1-30.

```
import tipCalculator as tip
```

This section just touches the surface of importing and working with external modules. For a more detailed discussion, please visit Chapter 7 which covers this topic specifically.

Iteration

The Python language has several iteration structures which are used to traverse through a series of items in a list, database records, or any other type of collection. A list in Python is a container that holds objects or values and can be indexed. For instance, we create a list of numbers in the following example. We then obtain the second element in the list by using the index value of 1 (indexing starts at zero, so the first element of the list is my_numbers[0]).

Listing 1-31.

```
>>> my_numbers = [1, 2, 3, 4, 5]
>>> my_numbers
[1, 2, 3, 4, 5]
>>> my_numbers[1]
2
```

For more information on lists, please see Chapter 2 that goes into detail about lists and other containers that can be used in Python.

The most commonly used iteration structure within the language is probably the *for *loop, which is known for its easy syntax and practical usage.

Listing 1-32.

```
>>> for value in my_numbers:
... print value
...
```

```
1
2
3
4
5
```

However, the *while* loop still plays an important role in iteration, especially when you are not dealing with collections of data, but rather working with conditional expressions. In this simple example, we use a *while* loop to iterate over the contents of *my_numbers*. Note that the *len()* function just returns the number of elements that are contained in the list.

Listing 1-33.

```
>>> x = 0
>>> while x < len(my_numbers):
... print my_numbers[x]
... x = x + 1
...
1
2
3
4
5
```

This section will take you though each of these two iteration structures and touch upon the basics of using them. The* while* loop is relatively basic in usage, whereas there are many different implementations and choices when using the *for* loop. I will only touch upon the *for* loop from a high-level perspective in this introductory chapter, but if you wish to go more in-depth then please visit Chapter 3.

While Loop

The *while* loop construct is used in order to iterate through code based upon a provided conditional statement. As long as the condition is true, then the loop will continue to process. Once the condition evaluates to false, the looping ends. The pseudocode for *while* loop logic reads as follows:

```
while True
    perform operation
```

The loop begins with the declaration of the *while* and conditional expression, and it ends once the conditional has been met and the expression is *True*. The expression is checked at the beginning of each looping sequence, so normally some value that is contained within the expression is changed within the suite of statements inside the loop. Eventually the value is changed in such a way that it makes the expression evaluate to False, otherwise an infinite loop would occur. Keep in mind that we need to indent each of the lines of code that exist within

the *while* loop. This not only helps the code to maintain readability, but it also allows Python to do away with the curly braces!

Listing 1-34. Example of a Java While Loop

```
int x = 9;
int y = 2;
int z = x - y;
while (y < x){
System.out.println("y is " + z + " less than x");
y = y++;
}
```

Now, let's see the same code written in Python.

Listing 1-35. Example of a Python While Loop

```
>>> x = 9
>>> y = 2
>>> while y < x:
...     print 'y is %d less than x' % (x-y)
...     y += 1
...
y is 7 less than x
y is 6 less than x
y is 5 less than x
y is 4 less than x
y is 3 less than x
y is 2 less than x
y is 1 less than x
```

In this example, you can see that the conditional *y < x *is evaluated each time the loop passes. Along the way, we increment the value of *y* by one each time we iterate, so that eventually *y *is no longer less than *x* and the loop ends.

For Loop

We will lightly touch upon *for* loops in this chapter, but you can delve deeper into the topic in chapter two or three when lists, dictionaries, tuples, and ranges are discussed. For now, you should know that a *for* loop is used to iterate through a defined set of values. The *for* loop is very useful for performing iteration through values because this is a concept which is used in just about any application. For instance, if you retrieve a list of database values, you can use a *for*loop to iterate through them and print each one out.

The pseudocode to *for* loop logic is as follows:

```
for each value in this defined set:
    perform suite of operations
```

As you can see with the pseudocode, I've indented in a similar fashion to the way in which the other expression constructs are indented. This uniform indentation practice is consistent throughout the Python programming language. We'll compare the *for* loop in Java to the Python syntax below so that you can see how the latter makes code more concise.

Listing 1-36. Example of Java For Loop

```
for (x = 0; x <= 10; x++){
    System.out.println(x);
}
```

Now, the same code implemented in Python:

Listing 1-37. Example of Python For Loop

```
>>> for x in range(10):
...     print x
...
0
1
2
3
4
5
6
7
8
9
```

In this example, we use a construct which has not yet been discussed. A range is a built-in function for Python which simply provides a range from one particular value to another. In the example, we pass the value 10 into the range which gives us all values between 0 and 10, inclusive of the zero at the front and exclusive at the end. We see this in the resulting print out after the expression.

Basic Keyboard Input

The Python language has a couple of built-in functions to take input from the keyboard as to facilitate the process of writing applications that allow user input. Namely, *raw_input()*, and *input()* can be used to prompt and accept user input from the command-line. Not only is this useful for creating command-line applications and scripts, but it also comes in handy for writing small tests into your applications.

The *raw_input()* function accepts keyboard entry and converts it to a string, stripping the trailing newline character. Similarly, the *input() *function accepts keyboard entry as *raw_input()*, but it then evaluates it as an expression. The *input()* function should be used with caution as it expects a valid Python expression to be entered. It will raise a *SyntaxError* if this is not the case. Using *input()* could result in a security concern as it basically allows your user to run arbitrary

Python code at will. It is best to steer clear of using *input()* in most cases and just stick to using raw_input. Let's take a look at using each of these functions in some basic examples.

Listing 1-38. Using raw_input() and input()

```
# The text within the function is optional, and it is used as a prompt to the
user
>>> name = raw_input("Enter Your Name:")
Enter Your Name:Josh
>>> print name
Josh
# Use the input function to evaluate an expression entered in by the user
>>> val = input ('Please provide an expression: ')
Please provide an expression: 9 * 3
>>> val
27
# The input function raises an error if an expression is not provided
>>> val = input ('Please provide an expression: ')
Please provide an expression: My Name is Josh
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
My Name is Josh
^
SyntaxError: invalid syntax
```

There will be examples provided later in the book for different ways of using the *raw_input()* function. Now let's take a look at some of the other Python statements that have not yet been covered in this chapter.

Other Python Statements

There are some other Python statements that can be used within applications as well, but they are probably better meant to be discussed within a later chapter as they provide more advanced functionality. The following is a listing of other Python statements which you will read more about later on:

exec - Execute Python code in a dynamic fashion

global — References a variable a global (Chapter 4)

with - New feature in 2.5 using __future__

class - Create or define a new class object (Chapter 6)

yield — Used with generators, returns a value (Chapter 4)

Documenting Code

Code documentation: an annoyingly important part of every application developer's life. Although many of us despise code documentation, it must exist for any application that is going to be used for production purposes. Not only is proper code documentation a must for manageability and long-term understanding of Python code fragments, but it also plays an important role in debugging some code as we will see in some examples below.

Sometimes we wish to document an entire function or class, and other times we wish to document only a line or two. Whatever the case, Python provides a way to do it in a rather unobtrusive manner. Much like many of the other programming languages that exist today, we can begin a comment on any part of any code line. We can also comment spanning multiple lines if we wish. Just on a personal note, we rather like the Python documentation symbol (#) or hash, as it provides for clear-cut readability. There are not many places in code that you will use the (#) symbol unless you are trying to perform some documentation. Many other languages use symbols such as (/) which can make code harder to read as those symbols are evident in many other non-documenting pieces of code. Okay, it is time to get off my soap box on Python and get down to business.

In order to document a line of code, you simply start the document or comment with a (#) symbol. This symbol can be placed anywhere on the line and whatever follows it is ignored by the Python compiler and treated as a comment or documentation. Whatever precedes the symbol will be parsed as expected.

Listing 1-39.

```
>>> # This is a line of documentation
>>> x = 0 # This is also documentation
>>> y = 20
>>> print x + y
20
```

As you can see, the Python parser ignores everything after the #, so we can easily document or comment as needed.

One can easily document multiple lines of code using the # symbol as well by placing the hash at the start of each line. It nicely marks a particular block as documentation. However, Python also provides a multi-line comment using the triple-quote (‘‘‘) designation at the beginning and end of a comment. This type of multi-line comment is also referred to as a doc string and it is only to be used at the start of a module, class, or function. While string literals can be placed elsewhere in code, they will not be treated as docstrings unless used at the start of the code. Let's take a look at these two instances of multi-line documentation in the examples that follow.

Listing 1-40. Multiple Lines of Documentation Beginning With #

```
# This function is used in order to provide the square
# of any value which is passed in.  The result will be
# passed back to the calling code.
```

```
def square_val(value):
    return value * value
...
>>> print square_val(3)
9
```

*Listing 1-41. Multiple Lines of Documentation **Enclosed in Triple Quotes (‘’’)*

```
def tip_calc(value, pct):
''' This function is used as a tip calculator based on a percentage
    which is passed in as well as the value of the total amount.
    In
    this function, the first parameter is to be the total amount of a
    bill for which we will calculate the tip based upon the second
    parameter as a percentage '''
    return value * (pct * .01)
...
>>> print tip_calc(75,15)
11.25
```

Okay, as we can see, both of these documentation methods can be used to get the task of documenting or comment code done. In Listing 1-40, we used multiple lines of documentation beginning with the # symbol in order to document the *square_val *function. In Listing 1-41, we use the triple-quote method in order to span multiple lines of documentation. Both of them appear to work as defined. However, the second option provides a greater purpose as it allows one to document specific named code blocks and retrieve that documentation by calling the *help(function)* function. For instance, if we wish to find out what the *square_val *code does, we need to visit the code and either read the multi-line comment or simply parse the code. However, if we wish to find out what the tip_calc function does, we can call the help(tip_calc) function and the multi-line comment will be returned to us. This provides a great tool to use for finding out what code does without actually visiting the code itself.

Listing 1-42. Printing the Documentation for the tip_calc Function

```
>>> help(tip_calc)
Help on function tip_calc in module __main__:

tip_calc(value, pct)
    This function is used as a tip calculator based on a percentage
    which is passed in as well as the value of the total amount. In
    this function, the first parameter is to be the total amount of a
    bill for which we will calculate the tip based upon the second
    parameter as a percentage
```

These examples and short explanations should give you a pretty good feel for the power of documentation that is provided by the Python language. As you can see, using the multi-line triple-quote method is very suitable for documenting classes or functions. Commenting with the # symbol provides a great way to organize comments within source and also for documenting those lines of code which may be “not so easy” to understand.

Python Help

Getting help when using the Jython interpreter is quite easy. Built into the interactive interpreter is an excellent *help()* option which provides information on any module, keyword, or topic available to the Python language. By calling the *help()* function without passing in the name of a function, the Python help system is invoked. While making use of the *help()* system, you can either use the interactive help which is invoked within the interpreter by simply typing *help()*, or as we have seen previously you can obtain the docstring for a specific object by typing *help(object)*.

It should be noted that while using the help system in the interactive mode, there is a plethora of information available at your fingertips. If you would like to see for yourself, simply start the Jython interactive interpreter and type *help()*. After you are inside the interactive help, you can exit at any time by typing *quit*. In order to obtain a listing of modules, keywords, or topics you just type either "*modules*," "*keywords*,"* *or "*topics*", and you will be provided with a complete listing. You will also receive help for using the interactive help system. . .or maybe this should be referred to as *meta-help*!

Although the Jython interactive help system is great, you may still need further assistance. There are a large number of books published on the Python language that will be sure to help you out. Make sure that you are referencing a book that provides you with information for the specific Python release that you are using as each version contains some differences. As mentioned previously in the chapter, the Jython version number contains is consistent with its CPython counterpart. Therefore, each feature that is available within CPython 2.5, for instance, should be available within Jython 2.5 and so on.

Summary

This chapter has covered lots of basic Python programming material. It should have provided a basic foundation for the fundamentals of programming in Python. This chapter shall be used to reflect upon while delving deeper into the language throughout the remainder of this book.

We began by discussing some of the differences between CPython and Jython. There are many good reasons to run Python on the JVM, including the availability of great Java libraries and excellent deployment targets. Once we learned how to install and configure Jython, we dove into the Python language. We learned about the declaration of variables and explained the dynamic tendencies of the language. We then went on to present the reserved words of the language and then discussed the coding structure which must be adhered to when developing a Python application. After that, we discussed operators and expressions. We learned that expressions are generally pieces of code that are evaluated to produce a value. We took a brief tour of Python functions as to cover their basic syntax and usage. Functions are a fundamental part of the language and most Python developers use functions in every program. A short section introducing classes followed, it is important to know the basics of classes early even though there is much more to learn in Chapter 6. We took a look at statements and learned that they consist of instructions that allow us to perform different tasks within our applications. Each

of the Python statements were discussed and examples were given. Iteration constructs were then discussed so that we could begin to use our statements and program looping tasks.

Following the language overview, we took a brief look at using keyboard input. This is a feature for many programs, and it is important to know for building basic programs. We then learned a bit about documentation, it is an important part of any application and Python makes it easy to do. Not only did we learn how to document lines of code, but also documenting entire modules, functions and classes. We touched briefly on the Python help() system as it can be a handy feature to use while learning the language. It can also be useful for advanced programmers who need to look up a topic that they may be a bit rusty on.

Throughout the rest of the book, you will learn more in-depth and advanced uses of the topics that we've discussed in this chapter. You will also learn concepts and techniques that you'll be able to utilize in your own programs to make them more powerful and easy to maintain.

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