Introduction

Welcome to Python 101! I wrote this book to help you learn Python. It is not meant to be an exhaustive reference book. Instead, the object is to get you acquainted with the building blocks of Python so that you can actually write something useful yourself. A lot of programming textbooks only teach you the language, but do not go much beyond that. I will endeavour to not only get you up to speed on the basics, but also to show you how to create useful programs. Now you may be wondering why just learning the basics isn't enough. In my experience, when I get finished reading an introductory text, I want to then create something, but I don't know how! I've got the learning, but not the glue to get from point A to point B. I think it's important to not only teach you the basics, but also cover intermediate material.

Thus, this book will be split into four parts:

- Part one will cover Python's basics
- Part two will be on a small subset of Python's Standard Library
- Part three will be a series of small tutorials
- Part four will cover Python packaging and distribution

Let me spend a few moments explaining what each part has to offer. In part one, we will cover the following:

- Python types (strings, lists, dicts, etc)
- Conditional statements
- Loops
- List and dictionary comprehensions
- Exception Handling
- File I/O
- Functions and Classes

Part two will talk about some of Python's standard library. The standard library is what comes pre-packaged with Python. It is made up of modules that you can import to get added functionality. For example, you can import the **math** module to gain some high level math functions. I will be cherry picking the modules I use the most as a day-to-day professional and explaining how they work. The reason I think this is a good idea is that they are common, every day modules that I think you will benefit knowing about at the beginning of your Python education. This section will also cover various ways to install 3rd party modules. Finally, I will cover how to create your own modules and packages and why you'd want to do that in the first place. Here are some of the modules we will be covering:

- csv
- ConfigParser
- logging
- os
- smtplib / email
- subprocess
- sys
- thread / queues
- time / datetime

Part three will be made up of small tutorials that will help you to learn how to use Python in a practical way. In this way, you will learn how to create Python programs that can actually do something useful! You can take the knowledge in these tutorials to create your own scripts. Ideas for further enhancements to these mini-applications will be provided at the end of each tutorial so you will have something that you can try out on your own. Here are a few of the 3rd party packages that we'll be covering:

- pip and easy_install
- configobj
- lxml
- requests
- virtualenv
- pylint / pychecker
- SQLAlchemy

Part four is going to cover how to take your code and give it to your friends, family and the world! You will learn the following:

- How to turn your reusable scripts into Python "eggs", "wheels" and more
- How to upload your creation to the Python Package Index (PyPI)
- How to create binary executables so you can run your application without Python
- How to create an installer for your application

The chapters and sections may not all be the same length. I plan to cover each topic well, but not every topic will require the same page count.

A Brief History of Python

I think it helps to know the background of the Python programming language. Python was created in the late 1980s. Everyone agrees that its creator is Guido van Rossum when he wrote it as a successor to the ABC programming language that he was using. Guido named the language after one of his favorite comedy acts: Monty Python. The language wasn't released until 1991 and it has grown a lot in terms of the number of included modules and packages included. At the time of this writing, there are two major versions of Python: the 2.x series and the 3.x (sometimes known as Python 3000) . The 3.x series is not backwards compatible with 2.x because the idea when creating 3.x was to get rid of some of the idiosyncrasies in the original. The current versions are 2.7.6 and 3.3.3. Most of the features in 3.x have been backported to 2.x; however, 3.x is getting the majority of Python's current development, so it is the version of the future.

Some people think Python is just for writing little scripts to glue together "real" code, like C++ or Haskell. However you will find Python to be useful in almost any situation. Python is used by lots of big name companies such as Google, NASA, LinkedIn, Industrial Light & Magic, and many others. Python is used not only on the backend, but also on the front. In case you're new to the computer science field, backend programming is the stuff that's behind the scenes; things like database processing, document generation, etc. Frontend processing is the pretty stuff most users are familiar with, such as web pages or desktop user interfaces. For example, there are some really nice Python GUI toolkits such as wxPython, PySide, and Kivy. There are also several web frameworks like Django, Pyramid, and Flask. You might find it surprising to know that Django is used for Instagram and Pinterest. If you have used these or many other websites, then you have used something that's powered by Python without even realizing it!

About the Author

You may be wondering about who I am and why I might be knowledgeable enough about Python to write about it, so I thought I'd give you a little information about myself. I started programming in Python in the Spring of 2006 for a job. My first assignment was to port Windows login scripts from Kixtart to Python. My second project was to port VBA code (basically a GUI on top of Microsoft Office products) to Python, which is how I first got started in wxPython. I've been using Python ever since, doing a variation of backend programming and desktop front end user interfaces.

I realized that one way for me to remember how to do certain things in Python was to write about them and that's how my Python blog came about: http://www.blog.pythonlibrary.org/. As I wrote, I would receive feedback from my readers and I ended up expanding the blog to include tips, tutorials, Python news, and Python book reviews. I work regularly with Packt Publishing as a technical reviewer, which means that I get to try to check for errors in the books before they're published. I also have written for the Developer Zone (DZone) and i-programmer websites as well as the Python Software Foundation. In November 2013, DZone published **The Essential Core Python Cheat Sheet** that I co-authored.

Conventions

As with most technical books, this one includes a few conventions that you need to be aware of. New topics and terminology will be in **bold**. You will also see some examples that look like the following:

```
>>> myString = "Welcome to Python!"
```

The **>>>** is a Python prompt symbol. You will see this in the Python **interpreter** and in **IDLE**. You will learn more about each of these in the first chapter. Other code examples will be shown in a similar manner, but without the **>>>**. Here is an example of a code block that includes line numbers:

```
1 def foo(bar):
2     print("This is function that doesn't do much!")
3     return None
```

Requirements

You will need a working Python installation. The examples should work in either Python 2.x or 3.x unless specifically marked otherwise. Most Linux and Mac machines come with Python already installed. However, if you happen to find yourself without Python, you can go download a copy from http://python.org/download/. There are up-to-date installation instructions on their website, so I won't include any installation instructions in this book. Any additional requirements will be explained later on in the book.

Reader Feedback

I welcome feedback about my writings. If you'd like to let me know what you thought of the book, you can send comments to the following address:

comments@pythonlibrary.org

Errata

I try my best not to publish errors in my writings, but it happens from time to time. If you happen to see an error in this book, feel free to let me know by emailing me at the following:

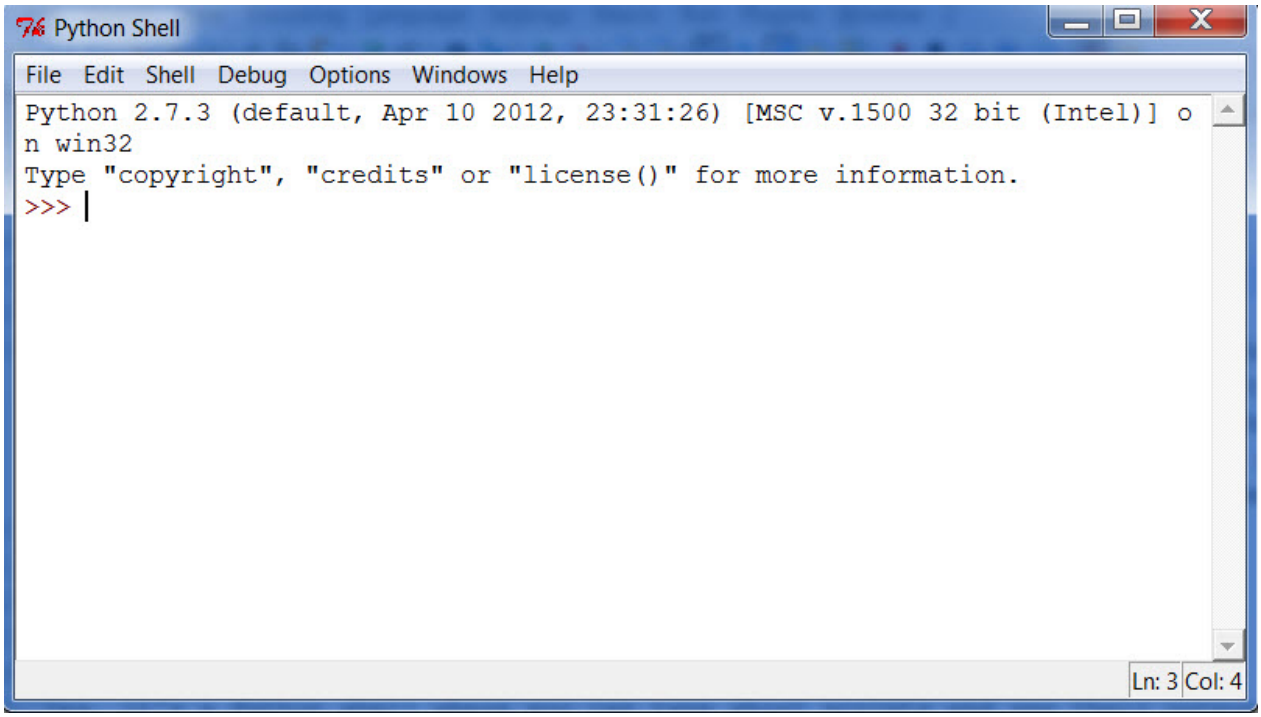
errata@pythonlibrary.org

Chapter 1 - IDLE Programming

Using IDLE

Python comes with its own code editor: **IDLE**. There is some unconfirmed lore that the name for IDLE comes from Eric Idle, an actor in *Monty Python*. I have no idea if that's true or not, but it would make sense in this context as it appears to be a pun on the acronym IDE or Integrate Development Environment. An IDE is an editor for programmers that provides color highlighting of key words in the language, auto-complete, a debugger and lots of other fun things. You can find an IDE for most popular languages and a number of IDEs will work with multiple languages. IDLE is kind of a lite IDE, but it does have all those items mentioned. It allows the programmer to write Python and debug their code quite easily. The reason I call it "lite" is the debugger is very basic and it's missing other features that programmers who have a background using products like *Visual Studio* will miss. You might also like to know that IDLE was created using Tkinter, a Python GUI toolkit that comes with Python.

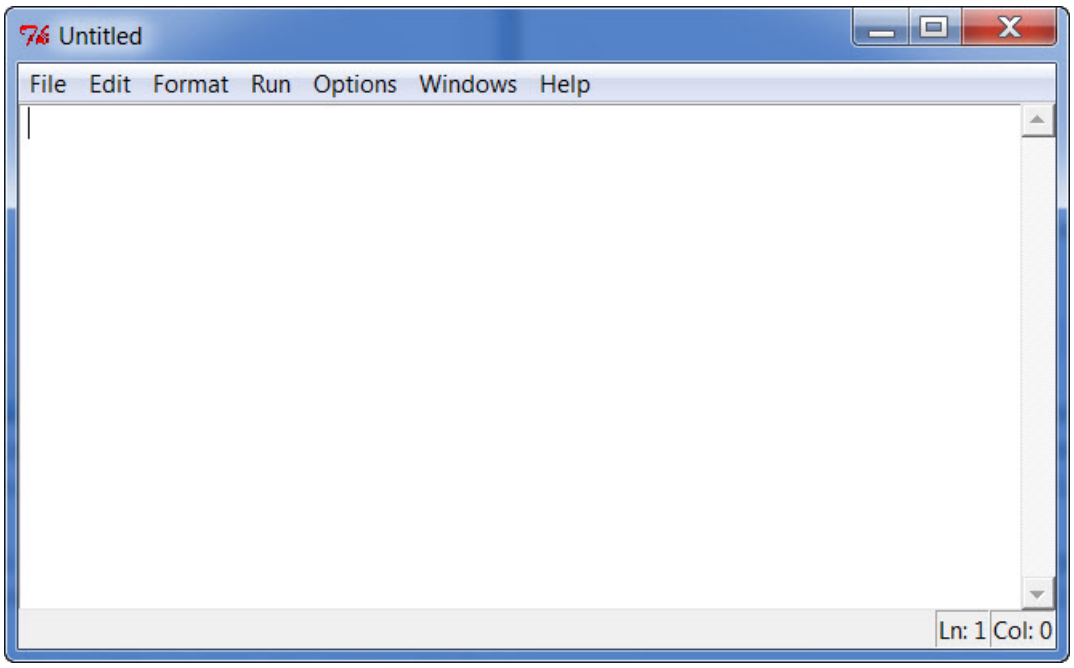
Open up IDLE and you'll see something like this:



Yes, it's a Python shell where you can type short scripts and see their output immediately and even interact with code in real time. There is no compiling of the code as Python is an interpretive language and runs in the Python interpreter. Let's write your first program now. Type the following after the command prompt (>>>) in IDLE:

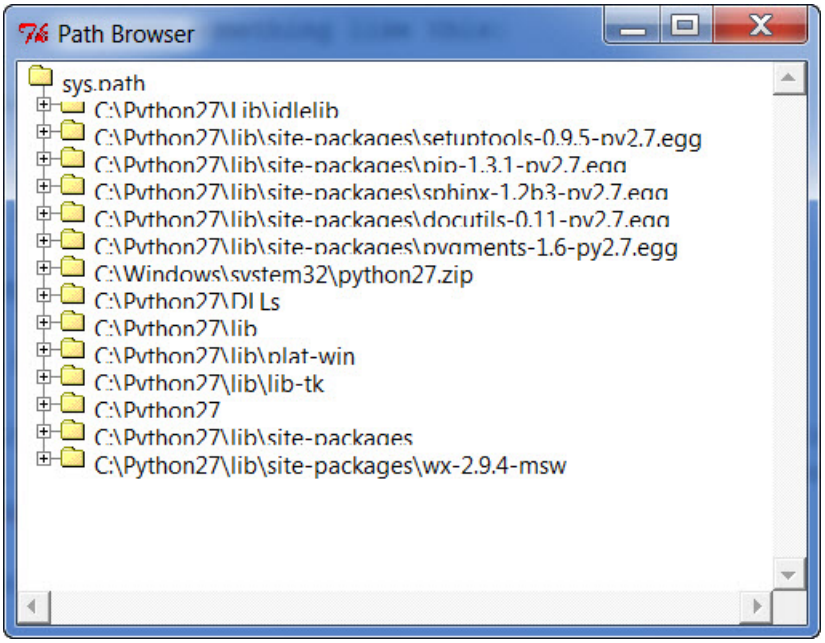
```
print "Hello from Python!"
```

You have just written your first program! All your program does is write a string to the screen, but you'll find that very helpful later on. If you want to save your code into a file, go to the File menu and choose New Window (or press CTRL+N). Now you can type in your program and save it here. The primary benefit of using the Python shell is that you can experiment with small snippets to see how your code will behave before you put the code into a real program. The code editor screen looks a little different than the IDLE screenshot above:



Now we'll spend a little time looking at IDLE's other useful features. In the **File** menu, you'll find a **Path Browser** which is useful for figuring out where Python looks for module imports. You see, Python first looks in the same directory as the script that is running to see if the file it needs to import is there. Then it checks a predefined list of other locations. You can actually add and remove locations as well.

Python comes with lots of modules and packages that you can import to add new features. For example, you can import the math module for all kinds of good math functions, like square roots, cosines, etcetera. The Path Browser will show you where these files are located on your hard drive, if you have imported anything. My Path Browser looks like this:



Next there's a **Class Browser** that will help you navigate your code. This is actually something that won't be very useful to you right now, but will be in the future. You'll find it helpful when you have lots of lines of code in a single file as it will give you a "tree-like" interface for your code. Note that you won't be able to load the Class Browser unless you have actually saved your program.

The **Edit** menu has your typical features, such as Copy, Cut, Paste, Undo, Redo and Select All. It also contains various ways to search your code and do a search and replace. Finally, the Edit menu has some menu items that will Show you various things, such as highlighting parentheses or displaying the auto-complete list.

The **Format** menu has lots of useful functionality. It has some helpful items for indenting and dedenting your code, as well as commenting out your code. I find that pretty helpful when I'm testing my code. Commenting out your code can be very helpful. One way it can be helpful is when you have a lot of code and you need to find out why it's not working correctly. Commenting out portions of it and re-running the script can help you figure out where you went wrong. You just go along slowly uncommenting out stuff until you hit your bug. Which reminds me; you may have noticed that the main IDLE screen has a Debugger menu. That is nice for debugging, but only in the Shell window. Sadly you cannot use the debugger in your main editing menu. If you need a more versatile debugger, you should either find a different IDE or try Python's debugger found in the pdb library.

What are Comments?

A comment is a way to leave un-runnable code that documents what you are doing in your code. Every programming language uses a different symbol to demarcate where a comment starts and ends. What do comments look like in Python though? A comment is anything that begins with an octothorpe (i.e. a hash or pound sign). The following is an example of some comments in action:

```
# This is a comment before some code
print "Hello from Python!"
print "Winter is coming" # this is an in-line comment
```

You can write comments on a line all by themselves or following a statement, like the second **print** statement above. The Python interpreter ignores comments, so you can write anything you want in them. Most programmers I have met don't use comments very much. However, I highly recommend using comments liberally not just for yourself, but for anyone else who might have to maintain or enhance your code in the future. I have found my own comments useful when I come back to a script that I wrote 6 months ago and I have found myself working with code that didn't have comments and wishing that it did so I could figure it out faster.

Examples of good comments would include explanations about complex code statements, or adding an explanation for acronyms in your code. Sometimes you'll need to leave a comment to explain why you did something a certain way because it's just not obvious.

Now we need to get back to going over the menu options of IDLE:

The **Run** menu has a couple of handy options. You can use it to bring up the Python Shell, check your code for errors, or run your code. The Options menu doesn't have very many items. It does have a Configure option that allows you to change a the code highlighting colors, fonts and key shortcuts. Other than that, you get a Code Context option that is helpful in that it puts an overlay in the editing window which will show you which class or function you're currently in. We will be explaining functions and classes near the end of Part I. You will find this feature is useful whenever you have a lot of code in a function and the name has scrolled off the top of the screen. With this option enabled, that doesn't happen.

The **Windows** menu shows you a list of currently open Windows and allows you to switch between them.

Last but not least is the **Help** menu where you can learn about IDLE, get help with IDLE itself or load up a local copy of the Python documentation. The documentation will explain how each piece of Python works and is pretty exhaustive in its coverage. The Help menu is probably the most helpful in that you can get access to the docs even when you're not connected to the internet. You can search the documentation,

find HOWTOs, read about any of the builtin libraries, and learn so much your head will probably start spinning.

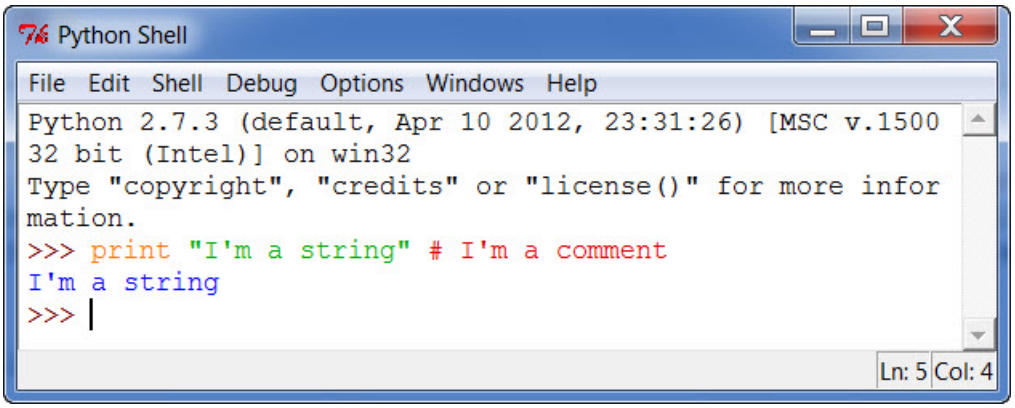
Other Tips

When you see code examples in the following chapters, you can write and run them in IDLE. I wrote all my programs in IDLE for the first couple of years of my Python programming life and I was pretty happy with it. There are lots of free Python IDEs out there though and several IDEs that you have to pay for. If you want to go cheap, you might want to take a look at Eclipse+PyDev, Editra or even Notepad++. For a paid IDE, I would recommend WingWare's IDE or possibly PyCharm. They have many more features such as integration with code repositories, better debuggers, refactoring help, etc.

In this book, we will be using IDLE in our examples because it comes with Python and will provide a common test bed. I still think IDLE has the best, most consistent code highlighting of any IDE I have used. Code highlighting is important in my mind in that it helps prevent me from using one of Python's key words (or build-ins) for a variable name. In case you're wondering, here is a list of those key words:

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

Let's see what happens as type out a few things in Python:



As you can see, IDLE color coded everything. A key word is orange, a string of text is in green, a comment is in red and the output from the print statement is in blue.

Wrapping Up

In this chapter we learned how to use Python's integrated development environment, IDLE. We also learned what comments are and how to use them. At this point, you should be familiar enough with IDLE to use it in the rest of this book. That means, we are ready to move on and start learning about Python's various data types. We will start with Strings in the following chapter.

Chapter 2 - All About Strings

There are several data types in Python. The main data types that you'll probably see the most are string, integer, float, list, dict and tuple. In this chapter, we'll cover the string data type. You'll be surprised how many things you can do with strings in Python right out of the box. There's also a string module that you can import to access even more functionality, but we won't be looking at that in this chapter. Instead, we will be covering the following topics:

- How to create strings
- String concatenation
- String methods
- String slicing
- String substitution

How to Create a String

Strings are usually created in three ways. You can use single, double or triple quotes. Let's take a look!

```
>>> my_string = "Welcome to Python!"
>>> another_string = 'The bright red fox jumped the fence.'
>>> a_long_string = '''This is a
multi-line string. It covers more than
one line'''
```

The triple quoted line can be done with three single quotes or three double quotes. Either way, they allow the programmer to write strings over multiple lines. If you print it out, you will notice that the output retains the line breaks. If you need to use single quotes in your string, then wrap it in double quotes. See the following example.

```
>>> my_string = "I'm a Python programmer!"
>>> otherString = 'The word "python" usually refers to a snake'
>>> tripleString = """Here's another way to embed "quotes" in a string"""
```

The code above demonstrates how you could put single quotes or double quotes into a string. There's actually one other way to create a string and that is by using the **str** method. Here's how it works:

```
>>> my_number = 123
>>> my_string = str(my_number)
```

If you type the code above into your interpreter, you'll find that you have transformed the integer value into a string and assigned the string to the variable *my_string*. This is known as **casting**. You can cast some data types into other data types, like numbers into strings. But you'll also find that you can't always do the reverse, such as casting a string like 'ABC' into an integer. If you do that, you'll end up with an error like the one in the following example:

```
>>> int('ABC')
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
ValueError: invalid literal for int() with base 10: 'ABC'
```

We will look at exception handling in a later chapter, but as you may have guessed from the message, this means that you cannot convert a literal into an integer. However, if you had done

```
>>> x = int("123")
```

then that would have worked fine.

It should be noted that a string is one of Python immutable types. What this means is that you cannot change a string's content after creation. Let's try to change one to see what happens:

```
>>> my_string = "abc"
>>> my_string[0] = "d"
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: 'str' object does not support item assignment
```

Here we try to change the first character from an "a" to a "d"; however this raises a TypeError that stops us from doing so. Now you may think that by assigning a new string to the same variable that you've changed the string. Let's see if that's true:

```
>>> my_string = "abc"
>>> id(my_string)
19397208
>>> my_string = "def"
>>> id(my_string)
25558288
>>> my_string = my_string + "ghi"
>>> id(my_string)
31345312
```

By checking the id of the object, we can determine that any time we assign a new value to the variable, its identity changes.

String Concatenation

Concatenation is a big word that means to combine or add two things together. In this case, we want to know how to add two strings together. As you might suspect, this operation is very easy in Python:

```
>>> string_one = "My dog ate "
>>> string_two = "my homework!"
>>> string_three = string_one + string_two
```

The '+' operator concatenates the two strings into one.

String Methods

A string is an object in Python. In fact, everything in Python is an object. However, you're not really ready for that. If you want to know more about how Python is an object oriented programming language, then you'll need to skip to that chapter. In the meantime, it's enough to know that strings have their very own methods built into them. For example, let's say you have the following string:

```
>>> my_string = "This is a string!"
```

Now you want to cause this string to be entirely in uppercase. To do that, all you need to do is call its **upper()** method, like this:

```
>>> my_string.upper()
>>> print my_string
```

If you have your interpreter open, you can also do the same thing like this:

```
>>> "This is a string!".upper()
```

There are many other string methods. For example, if you wanted everything to be lowercase, you would use the **lower()** method. If you wanted to remove all the leading and trailing white space, you would use **strip()**. To get a list of all the string methods, type the following command into your interpreter:

```
>>> dir(my_string)
```

You should end up seeing something like the following:

> ['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

You can safely ignore the methods that begin and end with double-underscores, such as __add__. They are not used in every day Python coding. Focus on the other ones instead. If you'd like to know what one of them does, just ask for help. For example, say you want to learn what **capitalize** is for. To find out, you would type

```
>>> help(my_string.capitalize)
```

This would return the following information:

> Help on built-in function capitalize:
>
> **capitalize(...)**
>
> S.capitalize() -> string
>
> Return a copy of the string S with only its first character capitalized.

You have just learned a little bit about a topic called **introspection**. Python allows easy introspection of all its objects, which makes it very easy to use. Basically, introspection allows you to ask Python about itself. In an earlier section, you learned about casting. You may have wondered how to tell what type the variable was (i.e. an int or a string). You can ask Python to tell you that!

```
>>> type(my_string)
    <type 'str'>
```

As you can see, the my_string variable is of type str!

String Slicing

One subject that you'll find yourself doing a lot of in the real world is string slicing. I have been surprised how often I have needed to know how to do this in my day-to-day job. Let's take a look at how slicing works with the following string:

```
>>> my_string = "I like Python!"
```

Each character in a string can be accessed using slicing. For example, if I want to grab just the first character, I could do this:

```
>>> my_string[0:1]
```

This grabs the first character in the string up to, but **not** including, the 2nd character. Yes, Python is zero-based. It's a little easier to understand if we map out each character's position in a table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
I		l	i	k	e		P	y	t	h	o	n	!

Thus we have a string that is 14 characters long, starting at zero and going through thirteen. Let's do a few more examples to get these concepts into our heads better.

```
>>> my_string[:1]
'I'
>>> my_string[0:12]
'I like Pytho'
>>> my_string[0:13]
'I like Python'
>>> my_string[0:14]
'I like Python!'
>>> my_string[0:-5]
'I like Py'
>>> my_string[:]
'I like Python!'
>>> my_string[2:]
'like Python!'
```

As you can see from these examples, we can do a slice by just specifying the beginning of the slice (i.e. my_string[2:]), the ending of the slice (i.e. my_string[:1]) or both (i.e. my_string[0:13]). We can even use negative values that start at the end of the string. So the example where we did my_string[0:-5] starts at zero, but ends 5 characters before the end of the string.

You may be wondering where you would use this. I find myself using it for parsing fixed width records in files or occasionally for parsing complicated file names that follow a very specific naming convention. I have also used it in parsing out values from binary-type files. Any job where you need to do text file processing will be made easier if you understand slicing and how to use it effectively.

String Formatting

String formatting (AKA substitution) is the topic of substituting values into a base string. Most of the time, you will be inserting strings within strings; however you will also find yourself inserting integers and floats into strings quite often as well. There are two different ways to accomplish this task. We'll start with the old way of doing things and then move on to the new.

Ye Olde Way of Substituting Strings

The easiest way to learn how to do this is to see a few examples. So here we go:

```
>>> my_string = "I like %s" % "Python"
>>> my_string
'I like Python'
```

```
>>> var = "cookies"
>>> newString = "I like %s" % var
>>> newString
'I like cookies'
>>> another_string = "I like %s and %s" % ("Python", var)
>>> another_string
'I like Python and cookies'
```

As you've probably guessed, the **%s** is the important piece in the code above. It tells Python that you may be inserting text soon. If you follow the string with a percent sign and another string or variable, then Python will attempt to insert it into the string. You can insert multiple strings by putting multiple instances of **%s** inside your string. You'll see that in the last example. Just note that when you insert more than one string, you have to enclose the strings that you're going to insert with parentheses.

Now let's see what happens if we don't insert enough strings:

```
>>> another_string = "I like %s and %s" % "Python"
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: not enough arguments for format string
```

Oops! We didn't pass enough arguments to format the string! If you look carefully at the example above, you'll notice it has two instances of %s, so to insert strings into it, you have to pass it the same number of strings! Now we're ready to learn about inserting integers and floats. Let's take a look!

```
>>> my_string = "%i + %i = %i" % (1,2,3)
>>> my_string
'1 + 2 = 3'
>>> float_string = "%f" % (1.23)
>>> float_string
'1.230000'
>>> float_string2 = "%.2f" % (1.23)
>>> float_string2
'1.23'
>>> float_string3 = "%.2f" % (1.237)
>>> float_string3
'1.24'
```

The first example above is pretty obvious. We create a string that accept three arguments and we pass them in. Just in case you hadn't figured it out yet, no, Python isn't actually doing any addition in that first example. For the second example, we pass in a float. Note that the output includes a lot of extra zeroes. We don't want that, so we tell Python to limit it to two decimal places in the 3rd example ("%.2f"). The last example shows you that Python will do some rounding for you if you pass it a float that's greater than two decimal places.

Now let's see what happens if we pass it bad data:

```
>>> int_float_err = "%i + %f" % ("1", "2.00")
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: %d format: a number is required, not str
```

In this example, we pass it two strings instead of an integer and a float. This raises a TypeError and tells us that Python was expecting a number. This refers to not passing an integer, so let's fix that and see if that fixes the issue:

```
>>> int_float_err = "%i + %f" % (1, "2.00")
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: float argument required, not str
```

Nope. We get the same error, but a different message that tells us we should have passed a float. As you can see, Python gives us pretty good information about what went wrong and how to fix it. If you fix the inputs appropriately, then you should be able to get this example to run.

Let's move on to the new method of string formatting!

Templates and the New String Formatting Methodology

This new method was actually added back in Python 2.4 as string templates, but was added as a regular string method via the **format** method in Python 2.6. So it's not really a new method, just newer. Anyway, let's start with templates!

```
>>> print "%(lang)s is fun!" % {"lang":"Python"}
Python is fun!
```

This probably looks pretty weird, but basically we just changed our **%s** into **%(lang)s**, which is basically the %s with a variable inside it. The second part is actually called a Python dictionary that we will be studying in the next section. Basically it's a key:value pair, so when Python sees the key "lang" in the string AND in the key of the dictionary that is passed in, it replaces that key with its value. Let's look at some more samples:

```
>>> print "%(value)s %(value)s %(value)s !" % {"value":"SPAM"}
SPAM SPAM SPAM !
>>> print "%(x)i + %(y)i = %(z)i" % {"x":1, "y":2}
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
KeyError: 'z'
>>> print "%(x)i + %(y)i = %(z)i" % {"x":1, "y":2, "z":3}
1 + 2 = 3
```

In the first example, you'll notice that we only passed in one value, but it was inserted 3 times! This is one of the advantages of using templates. The second example has an issue in that we forgot to pass in key, namely the "z" key. The third example rectifies this issue and shows the result. Now let's look at how we can do something similar with the string's format method!

```
>>> "Python is as simple as {0}, {1}, {2}".format("a", "b", "c")
'Python is as simple as a, b, c'
>>> "Python is as simple as {1}, {0}, {2}".format("a", "b", "c")
'Python is as simple as b, a, c'
>>> xy = {"x":0, "y":10}
>>> print "Graph a point at where x={x} and y={y}".format(**xy)
Graph a point at where x=0 and y=10
```

In the first two examples, you can see how we can pass items positionally. If we rearrange the order, we get a slightly different output. The last example uses a dictionary like we were using in the templates above. However, we have to extract the dictionary using the double asterisk to get it to work correctly here.

There are lots of other things you can do with strings, such as specifying a width, aligning the text, converting to different bases and much more. Be sure to take a look at some of the references below for more information.

- Python's official documentation on the str type
- String Formatting
- More on String Formatting
- String Services
- Python's documentation on unicode

Wrapping Up

We have covered a lot in this chapter. Let's review:

First we learned how to create strings themselves, then we moved on to the topic of string concatenation. After that we looked at some of the methods that the string object gives us. Next we looked at string slicing and we finished up by learning about string substitution.

In the next chapter, we will look at three more of Python's built-in data types: lists, tuples and dictionaries. Let's get to it!