Part I

Dipping Your Toe into Python

1

Programming Basics and Strings

This chapter is a gentle introduction to the practice of programming in Python. Python is a very rich language with many features, so it is important to learn to walk before you learn to run. Chapters 1 through 3 provide a basic introduction to common programming ideas, explained in easily digestible paragraphs with simple examples.

If you are already an experienced programmer interested in Python, you may want to read this chapter quickly and take note of the examples, but until Chapter 3 you will be reading material with which you've probably already gained some familiarity in another language.

If you are a novice programmer, by the end of this chapter you will learn the following:

- ❑ Some guiding principles for programming
- ❑ Directions for your first interactions with a programming language — Python.

The exercises at the end of the chapter provide hands-on experience with the basic information that you have learned.

How Programming is Different from Using a Computer

The first thing you need to understand about computers when you're programming is that you control the computer. Sometimes the computer doesn't do what you expect, but even when it doesn't do what you want the first time, it should do the same thing the second and third time — until you take charge and change the program.

The trend in personal computers has been away from reliability and toward software being built on top of other, unreliable, software. The results that you live with might have you believing that computers are malicious and arbitrary beasts, existing to taunt you with unbearable amounts of extra work and various harassments while you're already trying to accomplish something. However, after you've learned how to program, you gain an understanding of how this situation has come to pass, and perhaps you'll find that you can do better than some of the programmers whose software you've used.

Note that programming in a language like Python, an *interpreted* language, means that you are not going to need to know a whole lot about computer hardware, memory, or long sequences of 0s and 1s. You are going to write in text form like you are used to reading and writing but in a different and simpler language. Python is the language, and like English or any other language(s) you speak, it makes sense to other people who already speak the language. Learning a programming language can be even easier, however, because programming languages aren't intended for discussions, debates, phone calls, plays, movies, or any kind of casual interaction. They're intended for giving instructions and ensuring that those instructions are followed. Computers have been fashioned into incredibly flexible tools that have found a use in almost every business and task that people have found themselves doing, but they are still built from fundamentally understandable and controllable pieces.

Programming is Consistency

In spite of the complexity involved in covering all of the disciplines into which computers have crept, the basic computer is still relatively simple in principle. The internal mechanisms that define how a computer works haven't changed a lot since the 1950s when transistors were first used in computers.

In all that time, this core simplicity has meant that computers can, and should, be held to a high standard of consistency. What this means to you, as the programmer, is that anytime you tell a computer to metaphorically jump, you must tell it how high and where to land, and it will perform that jump — over and over again for as long as you specify. The program should not arbitrarily stop working or change how it works without you facilitating the change.

Programming is Control

Programming a computer is very different from creating a program, as the word applies to people in real life. In real life, you ask people to do things, and sometimes you have to struggle mightily to ensure that your wishes are carried out — for example, if you plan a party for 30 people and assign two of them to bring the chips and dip and they bring the drinks instead, it is out of your control.

With computers that problem doesn't exist. The computer does exactly what you tell it to do. As you can imagine, this means that you must pay some attention to detail to ensure that the computer does just what you want it to do.

One of the goals of Python is to program in *blocks* that enable you to think about larger and larger projects by building each project as pieces that behave in well-understood ways. This is a key goal of a programming style known as *object-oriented programming*. The guiding principle of this style is that you can create reliable pieces that still work when you piece them together, that are understandable, and that are useful. This gives you, the programmer, control over how the parts of your programs run, while enabling you to extend your program as the problems you're solving evolve.

Programming Copes with Change

Programs are run on computers that handle real-world problems; and in the real world, plans and circumstances frequently change. Because of these shifting circumstances, programmers rarely get the opportunity to create perfectly crafted, useful, and flexible programs. Usually, you can achieve only two of these goals. The changes that you will have to deal with should give you some perspective and lead you to program cautiously. With sufficient caution, you can create programs that know when they're being asked to exceed their capabilities, and they can fail gracefully by notifying their users that they've stopped. In the best cases, you can create programs that explain what failed and why. Python offers especially useful features that enable you to describe what conditions may have occurred that prevented your program from working.

What All That Means Together

Taken together, these beginning principles mean that you're going to be introduced to programming as a way of telling a computer what tasks you want it to do, in an environment where you are in control. You will be aware that sometimes accidents can happen and that these mistakes can be accommodated through mechanisms that offer you some discretion regarding how these conditions will be handled, including recovering from problems and continuing to work.

The First Steps

The absolute first step you need to take before you can begin programming in Python is to download and install Python version 3.1. Navigate to `www.python.org/download` and choose the newest version of Python. You will be taken to a page with instructions on how to download the appropriate version for your computer. For instance, if you are running Windows, it may say Windows x86 MSI Installer (3.0).

Programs are written in a form called *source code*. Source code contains the instructions that the language follows, and when the source code is read and processed, the instructions that you've put in there become the actions that the computer takes.

Just as authors and editors have specialized tools for writing for magazines, books, or online publications, programmers also need specialized tools. As a starting Python programmer, the right tool for the job is the Python IDLE GUI (graphical user interface).

Once the download is finished, double-click it to run the program. Your best bet is to accept the default prompts Python offers you. This process may take a few minutes, depending on your system.

After setup is complete, you will want to test to make sure it is installed properly. Click the Windows Start menu and go to All Programs. You will see Python 3.0 in the menu. Choose IDLE (Python GUI) and wait for the program to load.

Once IDLE launches, type in "Test, test, testing" and press the Enter key. If Python is running correctly, it should return the value

```
'Test, test, testing'
```

in blue letters and with single quotes (I'll get more into this soon). Congratulations — you have successfully installed Python and are well on your way to becoming a programming guru.

Installing Python 3.1 on Non-Windows Systems

If you are the proud owner of a Mac and are running Mac OS X, you are in luck; it comes with Python installed. Unfortunately, it may not be the most up-to-date version. For security and compatibility purposes, I would suggest logging on to `www.python.org/download/mac`. Check to see that your Mac OS X version is the right version for the Python you are installing.

If you have a Linux computer, you may also already have Python installed, but again, it may be an earlier version. I would once more suggest you go to the Python website to find the latest version (and of course, the one appropriate to your system). The website `www.python.org/download` should have instructions on how to download the right version for your computer.

Using the Python Shell

Before starting to write programs, you'll need to learn how to experiment with the Python shell. For now, you can think of the Python shell as a way to peer within running Python code. It places you inside of a running instance of Python, into which you can feed programming code; at the same time, Python will do what you have asked it to do and will show you a little bit about how it responds to its environment. Because running programs often have a *context* — things that you as the programmer have tailored to your needs — it is an advantage to have the shell because it lets you experiment with the context you have created.

Now that you have installed Python version 3.1, you can begin to experiment with the shell's basic behavior. For starters, type in some text:

```
>>>"Hello World. You will never see this."
```

Note that typing the previous sentence into the shell didn't actually do anything; nothing was changed in the Python environment. Instead, the sentence was evaluated by Python, to determine what, if anything, you wanted Python to do. In this case, you merely wanted it to read the text.

Although Python didn't technically do anything with your words, it did give some indication that it read them. Python indicated this by displaying the text you entered (known as a *string*) in quotes. A *string* is a data type, and each data type is displayed differently by Python. As you progress through this book, you will see the different ways Python displays each one.

Beginning to Use Python — Strings

At this point, you should feel free to experiment with using the shell's basic behavior. Type some text, in quotes; for starters, you could type the following:

```
>>> "This text really won't do anything"
"This text really won't do anything"
```

You should notice one thing immediately: After you entered a quote ("), the Python shell changed the color of everything up to the quote that completed the sentence. Of course, the preceding text is absolutely true. It did nothing: It didn't change your Python environment; it was merely *evaluated* by the running Python instance, in case it did determine that in fact you'd told it to do something. In this case, you've asked it only to read the text you wrote, but doing this doesn't constitute a change to the environment.

However, you can see that Python indicated that it saw what you entered. It showed you the text you entered, and it displayed it in the manner it will always display a string — in quotes. As you learn about other *data types*, you'll find that Python has a way of displaying each one differently.

What is a String?

A *string* is one of several data types that exist within the Python language. A data type, as the name implies, is a category that a particular type of data fits into. Every type of data you enter into a computer is segregated into one of these data types, whether they be numbers or letters, as is the case in this scenario. Giving data a type allows the computer to determine how to handle the data. For instance, if you want the program to show the mathematical equation 1+1 on a screen, you have to tell it that it is text. Otherwise, the program will interpret the data as a mathematical equation and evaluate it accordingly.

You'll get more into the different data types and how important it is to define them in a later chapter. For now however, know that a string is a data type that consists of any character, be it a letter, number, symbol, or punctuation mark. Therefore, the following are all examples of strings:

“Hello, how are you?”

“1+1”

“I ate 4 bananas”

“!@#$%^&*()”

Why the Quotes?

When you type a string into Python, you do so by preceding it with quotes. Whether these quotes are single ('), double("), or triple(""") depends on what you are trying to accomplish. For the most part, you will use single quotes, because it requires less effort (you do not need to hold down the Shift key to create them). Note, however, that they are interchangeable with the double and even triple quotes.

Try typing in some strings. After you type in each sentence, press the Enter key to allow Python to evaluate your statement.

Try It Out Entering Strings with Different Quotes

Enter the following strings, keeping in mind the type of quotes (single or double) and the ends of lines (use the Enter key when you see that the end of a line has been reached):

```
>>> "This is a string using a double quote"
'This a string using a double quote'
>>> 'This is a string with a single quote'
'This is a string with a single quote'
>>> """This string has three quotes
look at what it can do!"""
'This string has three quotes\nlook at what it can do!'
>>>
```

In the preceding examples, although the sentences may look different to the human eye, the computer is interpreting them all the same way: that is, as a string. There is a true purpose to having three different quoting methods, which is described next.

Why Three Types of Quotes?

The reasoning behind having three types of quotes is fairly simple. Let's say that you want to use a contraction in your sentence, as I have just done. If you type a sentence such as "I can't believe it's not butter" into the shell, nothing much happens, but when you actually try to get the program to *use* that string in any way, you will get an error message. To show you what I mean, the following section introduces you to the `print()` function.

Using the print() Function

A function in Python (and every other programming language) is a tool developers use to save time and make their programs more efficient. Instead of writing the same code over and over again, they store it in a function, and then call upon that function when they need it. Don't worry too much about functions at the moment; they are covered in greater detail later on. For now, it is enough to know what the term means and how it relates to programming.

The `print()` function is used whenever you want to print text to the screen. Try the following example in your Python shell:

```
>>> print("Hello World!")
```

When you press Enter, you should see the following:

```
Hello World!
```

You will want to note several things here. First, as you were entering in the `print()` function, a pop-up as shown in Figure 1-1 appeared, showing you the various options available to you within the function:

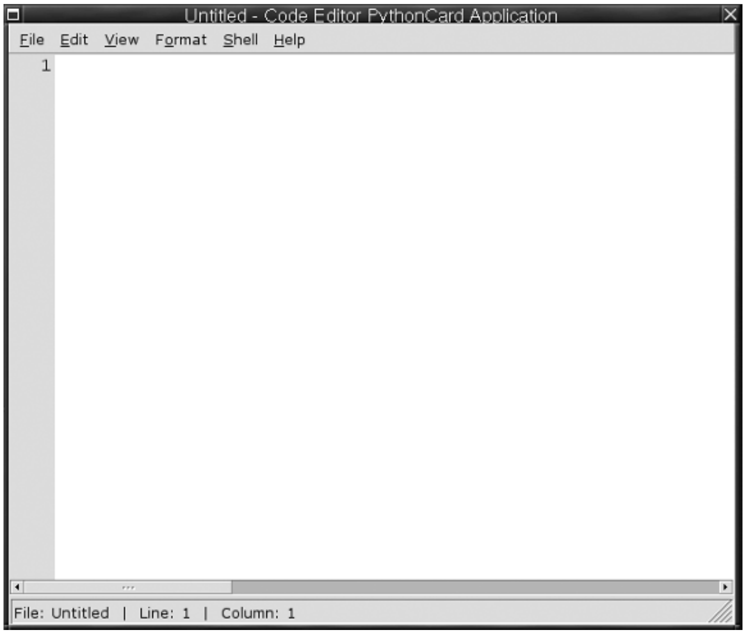
Untitled - Code Editor PythonCard Application
File Edit View Format Shell Help
File: Untitled | Line: 1 | Column: 1

Figure 1-1

Second, the text once more appeared in blue on the next line, but this time without quotation marks around it. This is because unlike in the previous examples, Python actually *did* something with the data.

Congratulations, you just wrote your first program!

Understanding Different Quotes

Now that you know how to use the `print()` function, you can begin to work with the different types of quotes discussed earlier in this chapter. Try the examples from earlier:

```
>>> print('This is a string using a single quote!')
This is a string using a single quote!
>>>print("This is a string using a double quote!")
This is a string using a double quote!
>>>print("""This string has three quotes!
Look at what it can do!""")
This string has three quotes
Look at what it can do!
```

In this example, you see that the single quote (') and double quote (") are interchangeable *in those instances.* However, when you want to work with a contraction, such as *don't*, or if you want to quote someone quoting something, observe what happens:

```
>>>print("I said, "Don't do it")
```

When you press Enter to execute the function, you will get the error message: `SyntaxError: invalid syntax (<pyshell#10>, line 1)`. I know what you are thinking — "What happened? I thought double and single quotes are interchangeable." Well, they are for the most part. However, when you try to mix them, it can often end up in a syntax error, meaning that your code has been entered incorrectly, and Python doesn't know what the heck you are trying to say.

What really happens here is that Python sees your first double quote and interprets that as the beginning of your string. When it encounters the double quote before the word *Don't*, it sees it as the end of the string. Therefore, the letters *on* make no sense to Python, because they are not part of the string. The string doesn't begin again until you get to the single quote before the *t*.

There is a simple solution to this, known as an escape. Retry the preceding code, adding an escape character to this string:

```
>>>print("I said, \"Don't do it")
I said, "Don't do it
```

This time, your code worked. When Python saw the backslash (`\`), or escape character, it knew to treat the double quote as a character, and not as a data type indicator. As you may have noticed, however, there is still one last problem with this line of code. See the missing double quote at the end of your results? To get Python to print the double quote at the end of the sentence, you simply add another escape character and a second double quote, like so:

```
>>>print("I said, \"Don't do it\"")
I said, "Don't do it"
```

Finally, let's take a moment to discuss the triple quote. You briefly saw its usage earlier. In that example, you saw that the triple quote allows you to write some text on multiple lines, without being processed until you close it with another triple quote. This technique is useful if you have a large amount of data that you do not wish to print on one line, or if you want to create line breaks within your code. Here, in the next example, you write a poem using this method:

```
>>>print("""Roses are red
Violets are blue
I just printed multiples lines
And you did too!""")
Roses are red
Violets are blue
I just printed multiple lines
And you did too!
```

There is another way to print text on multiple lines using the newline (`\n`) escape character, which is the most common of all the escape characters. I'll show it to you here briefly, and come back to discuss it in more depth in a later chapter. Try this code out:

```
>>>print("Roses are red \n Violets are blue \n
I just printed multiple
lines \n And you did too!")
Roses are red
Violets are blue
I just printed multiple lines
And you did too!
```

As you can see, the results are the same. Which you use is up to you, but the newline escape is probably more efficient and easier to read.

Putting Two Strings Together

There comes a time in every programmer's life when they have to combine two or more strings together. This is known as *concatenation*. For example, let's say that you have a database consisting of employees' first and last names. You may, at some point, wish to print these out as one whole record, instead of as two. In Python, each of these items can be treated as one, as shown here:

```
>>>"John"
'John'
>>>"Everyman"
'Everyman'
```

Try It Out Using + to Combine Strings

You can use several different methods to join distinct strings together. The first is by using the mathematical approach:

```
>>> "John" + "Everyman"
'JohnEveryman'
```

You could also just skip the + symbol altogether and do it this way:

```
>>>'JohnEveryman'
JohnEveryman
```

As you can see from these examples, both strings were combined; however, Python read the statement literally, and as such, there is no space between the two strings (remember: Python now views them as one string, not two!). So how do you fix this? You can fix it in two simple ways. The first involves adding a space after the first string, in this manner:

```
>>>"John " "Everyman"
John Everyman
```

I do not recommend this approach, however, because it can be difficult to ascertain that you added a space to the end of *John* if you ever need to read the code later in the future, say, when you are bleary-eyed and its four in the morning. The other approach is to simply use a separator, like so:

```
>>>"John" + " " + "Everyman"
John Everyman
```

Other reasons exist why you should use this method instead of simply typing in a space that have to do with database storage, but that is covered Chapter 14. Note that you can make any separator you like:

```
>>>"John" + "." + "Everyman"
John.Everyman
```

Joining Strings with the Print() Function

By default, the `print()` function is a considerate fellow that inserts the space for you when you print more than one string in a sentence. As you will see, there is no need to use a space separator. Instead, you just separate every string with a comma (,):

```
>>>print("John" , "Everyman")
John Everyman
```

Putting Strings Together in Different Ways

Another way to specify strings is to use a *format specifier*. It works by putting in a special sequence of characters that Python will interpret as a placeholder for a value that will be provided by you. This may initially seem like it's too complex to be useful, but format specifiers also enable you to control what the displayed information looks like, as well as a number of other useful tricks.

Try It Out **Using a Format Specifier to Populate a String**

In the simplest case, you can do the same thing with your friend, John Q.:

```
>>> "John Q. %s" % ("Public")
'John Q. Public'
```

How It Works

The `%s` is known as a format specifier, specifically for strings. As the discussion on data types continues throughout this book, you take a look at several more, each specific to its given data type. Every specifier acts as a placeholder for that type in the string; and after the string, the `%` sign outside of the string indicates that after it, all of the values to be inserted into the format specifier will be presented there to be used in the string.

You may notice the parentheses. This tells the string that it should expect to see a sequence that contains the values to be used by the string to populate its format specifiers.

A simpler way to think of it is to imagine that the `%s` is a storage bin that holds the value in the parentheses. If you want to do more than one value, you would simply add another format specifier, in this manner:

```
>>>"John %s%s" % ("Every" , "Man")
John Everyman
```

These sequences are an integral part of programming in Python, and as such, they are covered in greater detail later in this book. For now, just know that every format specification in a string has to have an element that matches it in the sequence that is provided to it. Each item in the sequence are strings that must be separated by commas.

So why do they call it a format specifier if you store data in it? The reason is that it has multiple functions; being a container is only one of them. The following example shows you how to not only store data with the format specifier, but specify how that data will be displayed as well.

Try It Out More String Formatting

In this example, you tell the format specifier how many characters to expect. Try the following code and watch what happens:

```
>>> "%s %s %10s" % ("John" , "Every", "Man")
'John Every        Man'
>>> "%-5s %s %10s" % ("John" , "Every", "Man")
John     Every          Man
```

How It Works

In the first line of code, the word *Man* appears far away from the other words; this is because in your last format specifier, you added a 10, so it is expecting a string with ten characters. When it does not find ten (it only finds three . . . M-a-n) it pads space in between with seven spaces.

In the second line of code you entered, you will notice that the word *Every* is spaced differently. This occurs for the same reason as before — only this time, it occurred to the left, instead of the right. Whenever you right a negative (–) in your format specifier, the format occurs to the left of the word. If there is just a number with no negative, it occurs to the right.

Summary

In this chapter you learned how to install Python, and how to work with the Python GUI (IDLE), which is a program written in Python for the express purpose of editing Python programs. In addition to editing files, this "shell" allows you to experiment with simple programming statements in the Python language.

Among the things you learned to do within the shell are the basics of handling strings, including string concatenation, as well as how to format strings with format specifiers, and even storing strings within that same `%s` format specifier. In addition, you learned to work with multiple styles of quotes, including the single, double, and triple, and found out what the `\n` newline escape character was for.

Finally, you learned your very first function, `print()`, and wrote your first program, the Hello World standby, which is a time-honored tradition among programmers; it's similar to learning "Smoke on the Water" if you play guitar — it's the first thing you'll ever learn.

The key things to take away from this chapter are:

❑ Programming is consistency. All programs are created with a specific use in mind, and your user will expect the program not only to live up to that usage, but to work in exactly the same manner each and every time. If the user clicks a button and a print dialog box pops up, this button should always work in this manner.

❑ Programming is control. As a programmer, you control the actions your application can and cannot take. Even aspects of the program that seem random to the casual observer are, in fact, controlled by the parameters that you create.

- Programming copes with changes. Through repeated tests, you can ensure that your program responds appropriately to the user, even when they ask the program to do something you did not develop it to do.
- Strings are a data type, or simply put, a category of data. These strings allow you to interact with the user in a plethora of ways, such as printing text to the window, accepting text from the user, and so forth. A string can consist of any letter, number, or special character.
- The `print()` function allows you to print text to the user's screen. It follows the syntax: `print("Here is some text").`

Exercises

1. In the Python shell, type the string, `"print Rock a by baby,\n\ton the tree top,\t\ when the wind blows\n\t\t\t the cradle will drop."` Feel free to experiment with the number of `\n` and `\t` escape sequences to see how this affects what gets displayed on your screen. You can even try changing their placement. What do you think you are likely to see?
2. In the Python shell, use the same string indicated in Exercise 1, but this time, display it using the `print()` function. Once more, try differing the number of `\n` and `\t` escape sequences. How do you think it will differ?