Beginning Python: From Novice to Professional, Second Edition

ISBN-13 (pbk): 978-1-59059-982-2

ISBN-10 (pbk): 1-59059-982-9

ISBN-13 (electronic): 978-1-4302-0634-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Frank Pohlmann
Technical Reviewers: Gregg Bolinger, Richard Taylor
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editor: Marilyn Smith
Associate Production Director: Kari Brooks-Copony
Production Editor: Liz Berry
Compositor: Pat Christenson
Proofreader: April Eddy
Indexer: John Collin
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 1

Instant Hacking: The Basics

It's time to start hacking.[1] In this chapter, you learn how to take control of your computer by speaking a language it understands: Python. Nothing here is particularly difficult, so if you know the basic principles of how your computer works, you should be able to follow the examples and try them out yourself. I'll go through the basics, startiwng with the excruciatingly simple, but because Python is such a powerful language, you'll soon be able to do pretty advanced things.

First, I show you how to get the software you need. Then I tell you a bit about algorithms and their main components. Throughout these sections, there are numerous small examples (most of them using only simple arithmetic) that you can try out in the Python interactive interpreter (covered in the section "The Interactive Interpreter" in this chapter). You learn about variables, functions, and modules, and after handling these topics, I show you how to write and run larger programs. Finally, I deal with strings, an important aspect of almost any Python program.

Installing Python

Before you can start programming, you need some new software. What follows is a short description of how to download and install Python. If you want to jump into the installation process without detailed guidance, you can simply visit `http://www.python.org/download` to get the most recent version of Python.

Windows

To install Python on a Windows machine, follow these steps:

1. Open a web browser and go to `http://www.python.org`.
2. Click the Download link.
3. You should see several links here, with names such as Python 2.5.*x* and Python 2.5.*x* Windows installer. Click the Windows installer link to download the installer file. (If you're running on an Itanium or AMD machine, you need to choose the appropriate installer.)

1. *Hacking* is not the same as *cracking*, which is a term describing computer crime. The two are often confused. Hacking basically means "having fun while programming." For more information, see Eric Raymond's article "How to Become a Hacker" at `http://www.catb.org/~esr/faqs/hacker-howto.html`.

■ **Note** If you can't find the link mentioned in step 3, click the link with the highest version among those with names like Python 2.5.*x*. For Python 2.5, you could simply go to `http://www.python.org/2.5`. Follow the instructions for Windows users. This will entail downloading a file called `python-2.5.x.msi` (or something similar), where 2.5.*x* should be the version number of the newest release.

4. Store the Windows Installer file somewhere on your computer, such as `C:\download\python-2.5.x.msi`. (Just create a directory where you can find it later.)
5. Run the downloaded file by double-clicking it in Windows Explorer. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the installation is finished, and you're ready to roll!

Assuming that the installation went well, you now have a new program in your Windows Start menu. Run the Python Integrated Development Environment (IDLE) by selecting Start ➤ Programs ➤ Python[2] ➤ IDLE (Python GUI).

You should now see a window that looks like the one shown in Figure 1-1. If you feel a bit lost, simply select Help ➤ IDLE Help from the menu to get a simple description of the various menu items and basic usage. For more documentation on IDLE, check out `http://www.python.org/idle`. (Here you will also find more information on running IDLE on platforms other than Windows.) If you press F1, or select Help ➤ Python Docs from the menu, you will get the full Python documentation. (The document there of most use to you will probably be the Library Reference.) All the documentation is searchable.

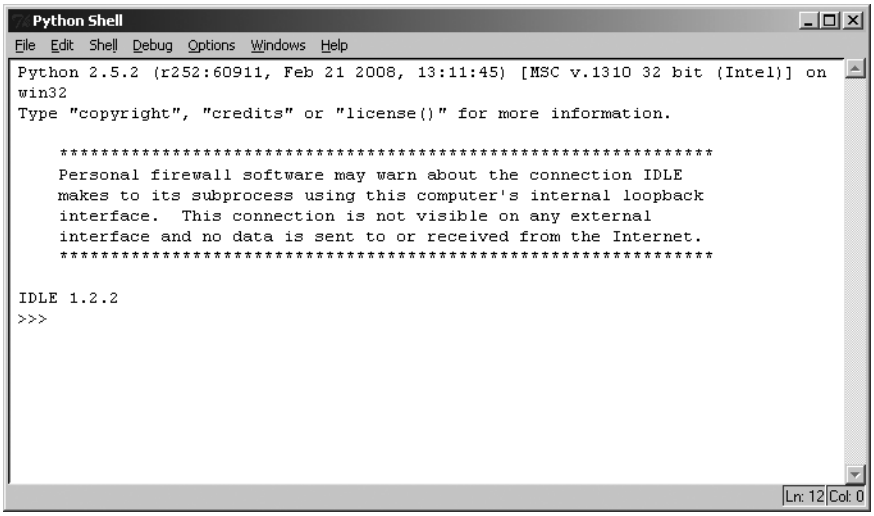

Figure 1-1. *The IDLE interactive Python shell*

2. This menu option will probably include your version number, as in Python 2.5.

Once you have the IDLE interactive Python shell running, you can continue with the section "The Interactive Interpreter," later in this chapter.

WINDOWS INSTALLER

Python for Microsoft Windows is distributed as a Windows Installer file, and requires that your Windows version supports Windows Installer 2.0 (or later). If you don't have Windows Installer, it can be downloaded freely for Windows 95, 98, ME, NT 4.0, and 2000. Windows XP and later versions of Windows already have Windows Installer, and many older machines will, too. There are download instructions for the Installer on the Python download page.

Alternatively, you could go to the Microsoft download site, `http://www.microsoft.com/downloads`, and search for "Windows Installer" (or simply select it from the download menu). Choose the most recent version for your platform and follow the download and installation instructions.

If you're uncertain about whether you have Windows Installer, simply try executing step 5 of the previous installation instructions: double-click the MSI file. If you get the install wizard, everything is okay. See `http://www.python.org/2.5/msi.html` for advanced features of the Windows Installer related to Python installation.

Linux and UNIX

In most Linux and UNIX installations (including Mac OS X), a Python interpreter will already be present. You can check whether this is the case for you by running the `python` command at the prompt, as follows:

```
$ python
```

Running this command should start the interactive Python interpreter, with output similar to the following:

```
Python 2.5.1 (r251:54869, Apr 18 2007, 22:08:04)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

■ **Note** To exit the interactive interpreter, use Ctrl-D (press the Ctrl key and while keeping that depressed, press D).

If there is no Python interpreter installed, you will probably get an error message similar to the following:

```
bash: python: command not found
```

In that case, you need to install Python yourself, as described in the following sections.

Using a Package Manager

Several package systems and installation mechanisms exist for Linux. If you're running a Linux system with some form of package manager, chances are you can get Python through it.

■ **Note** You will probably need to have administrator privileges (a root account) in order to install Python using a package manager in Linux.

For example, if you're running Debian Linux, you should be able to install Python with the following command:

```
$ apt-get install python
```

If you're running Gentoo Linux, you should be able to use Portage, like this:

```
$ emerge python
```

In both cases, `$` is, of course, the bash prompt.

■ **Note** Many other package managers out there have automatic download capabilities, including Yum, Synaptic (specific to Ubuntu Linux), and other Debian-style managers. You should probably be able to get recent versions of Python through these.

Compiling from Sources

If you don't have a package manager, or would rather not use it, you can compile Python yourself. This may be the method of choice if you are on a UNIX box but you don't have root access (installation privileges). This method is very flexible, and enables you to install Python wherever you want, including in your own home directory. To compile and install Python, follow these steps:

1. Go to the download page (refer to steps 1 and 2 in the instructions for installing Python on a Windows system).
2. Follow the instructions for downloading the sources.
3. Download the file with the extension `.tgz`. Store it in a temporary location. Assuming that you want to install Python in your home directory, you may want to put it in a directory such as `~/python`. Enter this directory (e.g., using `cd ~/python`).
4. Unpack the archive with the command `tar -xzvf Python-2.5.tgz` (where `2.5` is the version number of the downloaded source code). If your version of `tar` doesn't support the `z` option, you may want to uncompress the archive with `gunzip` first, and then use `tar -xvf` afterward. If there is something wrong with the archive, try downloading it again. Sometimes errors occur during download.

5. Enter the unpacked directory:

```
$ cd Python-2.5
```

Now you should be able to execute the following commands:

```
./configure --prefix=$(pwd)
make
make install
```

You should end up with an executable file called `python` in the current directory. (If this doesn't work, consult the `README` file included in the distribution.) Put the current directory in your `PATH` environment variable, and you're ready to rock.

To find out about the other configuration directives, execute this command:

```
./configure --help
```

Macintosh

If you're using a Macintosh with a recent version of Mac OS X, you'll have a version of Python installed already. Just open the Terminal application and enter the command `python` to start it. Even if you would like to install a newer version of Python, you should leave this one alone, as it is used in several parts of the operating system. You could use either MacPorts (`http://macports.org`) or Fink (`http://finkproject.org`), or you could use the distribution from the Python web site, by following these steps:

1. Go to the standard download page (see steps 1 and 2 from the Windows instructions earlier in this chapter).
2. Follow the link for the Mac OS X installer. There should also be a link to the MacPython download page, which has more information. The MacPython page also has versions of Python for older versions of the Mac OS.
3. Once you've downloaded the installer `.dmg` file, it will probably mount automatically. If not, simply double-click it. In the mounted disk image, you'll find an installer package (`.mpkg`) file. If you double-click this, the installation wizard will open, which will take you through the necessary steps.

Other Distributions

You now have the standard Python distribution installed. Unless you have a particular interest in alternative solutions, that should be all you need. If you are curious (and, perhaps, feeling a bit courageous), read on.

Several Python distributions are available in addition to the official one. The most well-known of these is probably ActivePython, which is available for Linux, Windows, Mac OS X, and several UNIX varieties. A slightly less well-known but quite interesting distribution is Stackless Python. These distributions are based on the standard implementation of Python, written in the C programming language. Two distributions that take a different approach are Jython and IronPython. If you're interested in development environments other than IDLE, Table 1-1 lists some options.

Table 1-1. *Some Integrated Development Environments (IDEs) for Python*

Environment	Description	Web Site
IDLE	The standard Python environment	`http://www.python.org/idle`
Pythonwin	Windows-oriented environment	`http://www.python.org/download/windows`
ActivePython	Feature-packed; contains Pythonwin IDE	`http://www.activestate.com`
Komodo	Commercial IDE	`http://www.activestate.com`[3]
Wingware	Commercial IDE	`http://www.wingware.com`
BlackAdder	Commercial IDE and (Qt) GUI builder	`http://www.thekompany.com`
Boa Constructor	Free IDE and GUI builder	`http://boa-constructor.sf.net`
Anjuta	Versatile IDE for Linux/UNIX	`http://anjuta.sf.net`
Arachno Python	Commercial IDE	`http://www.python-ide.com`
Code Crusader	Commercial IDE	`http://www.newplanetsoftware.com`
Code Forge	Commercial IDE	`http://www.codeforge.com`
Eclipse	Popular, flexible, open source IDE	`http://www.eclipse.org`
eric	Free IDE using Qt	`http://eric-ide.sf.net`
KDevelop	Cross-language IDE for KDE	`http://www.kdevelop.org`
VisualWx	Free GUI builder	`http://visualwx.altervista.org`
wxDesigner	Commercial GUI builder	`http://www.roebling.de`
wxGlade	Free GUI builder	`http://wxglade.sf.net`

ActivePython is a Python distribution from ActiveState (`http://www.activestate.com`). At its core, it's the same as the standard Python distribution for Windows. The main difference is that it includes a lot of extra goodies (modules) that are available separately. It's definitely worth a look if you are running Windows.

3. Komodo has been made open source, so free versions are also available.

Stackless Python is a reimplementation of Python, based on the original code, but with some important internal changes. To a beginning user, these differences won't matter much, and one of the more standard distributions would probably be more useful. The main advantages of Stackless Python are that it allows deeper levels of recursion and more efficient multithreading. As mentioned, both of these are rather advanced features, not needed by the average user. You can get Stackless Python from `http://www.stackless.com`.

Jython (`http://www.jython.org`) and IronPython (`http://www.codeplex.com/IronPython`) are different—they're versions of Python implemented in other languages. Jython is implemented in Java, targeting the Java Virtual Machine, and IronPython is implemented in C#, targeting the .NET and MONO implementations of the common language runtime (CLR). At the time of writing, Jython is quite stable, but lagging behind Python—the current Jython version is 2.2, while Python is at 2.5. There are significant differences in these two versions of the language. IronPython is still rather young, but it is quite usable, and it is reported to be faster than standard Python on some benchmarks.

Keeping in Touch and Up-to-Date

The Python language evolves continuously. To find out more about recent releases and relevant tools, the `python.org` web site is an invaluable asset. To find out what's new in a given release, go to the page for the given release, such as `http://python.org/2.5` for release 2.5. There you will also find a link to Andrew Kuchling's in-depth description of what's new for the release, with a URL such as `http://python.org/doc/2.5/whatsnew` for release 2.5. If there have been new releases since this book went to press, you can use these web pages to check out any new features.

■ **Tip** For a summary of what's changed in the more radically new release 3.0, see `http://docs.python.org/dev/3.0/whatsnew/3.0.html`.

If you want to keep up with newly released third-party modules or software for Python, check out the Python email list `python-announce-list`; for general discussions about Python, try `python-list`, but be warned: this list gets a *lot* of traffic. Both of these lists are available at `http://mail.python.org`. If you're a Usenet user, these two lists are also available as the newsgroups `comp.lang.python.announce` and `comp.lang.python`, respectively. If you're totally lost, you could try the `python-help` list (available from the same place as the two other lists) or simply email `help@python.org`. Before you do, you really ought to see if your question is a frequently asked one, by consulting the Python FAQ, at `http://python.org/doc/faq`, or by performing a quick Web search.

The Interactive Interpreter

When you start up Python, you get a prompt similar to the following:

```
Python 2.5.1 (r251:54869, Apr 18 2007, 22:08:04)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

■ **Note** The exact appearance of the interpreter and its error messages will depend on which version you are using.

This might not seem very interesting, but believe me—it is. This is your gateway to hackerdom—your first step in taking control of your computer. In more pragmatic terms, it's an interactive Python interpreter. Just to see if it's working, try the following:

```
>>> print "Hello, world!"
```

When you press the Enter key, the following output appears:

```
Hello, world!
>>>
```

■ **Note** If you are familiar with other computer languages, you may be used to terminating every line with a semicolon. There is no need to do so in Python. A line is a line, more or less. You may add a semicolon if you like, but it won't have any effect (unless more code follows on the same line), and it is not a common thing to do.

What happened here? The `>>>` thingy is the prompt. You can write something in this space, like `print "Hello, world!"`. If you press Enter, the Python interpreter prints out the string "Hello, world!" and you get a new prompt below that.

■ **Note** The term "printing" in this context refers to writing text to the screen, not producing hard copies with a printer.

What if you write something completely different? Try it out:

```
>>> The Spanish Inquisition
SyntaxError: invalid syntax
>>>
```

Obviously, the interpreter didn't understand that.[4] (If you are running an interpreter other than IDLE, such as the command-line version for Linux, the error message will be slightly

4. After all, no one expects the Spanish Inquisition . . .

dfferent.) The interpreter also indicates what's wrong: it will emphasize the word *Spanish* by giving it a red background (or, in the command-line version, by using a caret, ^).

If you feel like it, play around with the interpreter some more. For some guidance, try entering the command `help` at the prompt and pressing Enter. As mentioned, you can press F1 for help about IDLE. Otherwise, let's press on. After all, the interpreter isn't much fun when you don't know what to tell it, is it?

Algo . . . What?

Before you start programming in earnest, I'll try to give you an idea of what computer programming is. Simply put, it's telling a computer what to do. Computers can do a lot of things, but they aren't very good at thinking for themselves. They really need to be spoon-fed the details. You need to feed the computer an algorithm in some language it understands. *Algorithm* is just a fancy word for a procedure or recipe—a detailed description of how to do something. Consider the following:

```
SPAM with SPAM, SPAM, Eggs, and SPAM:
First, take some SPAM.
Then add some SPAM, SPAM, and eggs.
If a particularly spicy SPAM is desired, add some SPAM.
Cook until done - Check every 10 minutes.
```

This recipe may not be very interesting, but how it's constructed is. It consists of a series of instructions to be followed in order. Some of the instructions may be done directly ("take some SPAM"), while some require some deliberation ("If a particularly spicy SPAM is desired"), and others must be repeated several times ("Check every 10 minutes.")

Recipes and algorithms consist of ingredients (objects, things), and instructions (statements). In this example, SPAM and eggs were the ingredients, while the instructions consisted of adding SPAM, cooking for a given length of time, and so on. Let's start with some reasonably simple Python ingredients and see what you can do with them.

Numbers and Expressions

The interactive Python interpreter can be used as a powerful calculator. Try the following:

```
>>> 2 + 2
```

This should give you the answer 4. That wasn't too hard. Well, what about this:

```
>>> 53672 + 235253
288925
```

Still not impressed? Admittedly, this is pretty standard stuff. (I'll assume that you've used a calculator enough to know the difference between `1+2*3` and `(1+2)*3`.) All the usual arithmetic operators work as expected—almost. There is one potential trap here, and that is integer division (in Python versions prior to 3.0):

```
>>> 1/2
0
```

What happened here? One integer (a nonfractional number) was divided by another, and the result was rounded down to give an integer result. This behavior can be useful at times, but often (if not most of the time), you need ordinary division. What do you do to get that? There are two possible solutions: use real numbers (numbers with decimal points) rather than integers, or tell Python to change how division works.

Real numbers are called *floats* (or *floating-point numbers*) in Python. If either one of the numbers in a division is a float, the result will be, too:

```
>>> 1.0 / 2.0
0.5

>>> 1/2.0
0.5
>>> 1.0/2
0.5

>>> 1/2.
0.5
```

If you would rather have Python do proper division, you could add the following statement to the beginning of your program (writing full programs is described later) or simply execute it in the interactive interpreter:

```
>>> from __future__ import division
```

■ **Note** In case it's not entirely clear, the `future` in the instruction is surrounded by two underscores on both sides: `__future__`.

Another alternative, if you're running Python from the command line (e.g., on a Linux machine), is to supply the command-line switch `-Qnew`. In either case, division will suddenly make a bit more sense:

```
>>> 1 / 2
0.5
```

Of course, the single slash can no longer be used for the kind of integer division shown earlier. A separate operator will do this for you—the double slash:

```
>>> 1 // 2
0
```

The double slash consistently performs integer division, even with floats:

```
>>> 1.0 // 2.0
0.0
```

There is a more thorough explanation of the `__future__` stuff in the section "Back to the `__future__`," later in this chapter.

Now you've seen the basic arithmetic operators (addition, subtraction, multiplication, and division), but one more operator is quite useful at times:

```
>>> 1 % 2
1
```

This is the remainder (modulus) operator. `x % y` gives the remainder of `x` divided by `y`. Here are a few more examples:

```
>>> 10 / 3
3
>>> 10 % 3
1
>>> 9 / 3
3
>>> 9 % 3
0
>>> 2.75 % 0.5
0.25
```

Here `10/3` is `3` because the result is rounded down. But 3 × 3 is 9, so you get a remainder of 1. When you divide 9 by 3, the result is exactly 3, with no rounding. Therefore, the remainder is 0. This may be useful if you want to check something "every 10 minutes" as in the recipe earlier in the chapter. You can simply check whether `minute % 10` is 0. (For a description on how to do this, see the sidebar "Sneak Peek: The `if` Statement," later in this chapter.) As you can see from the final example, the remainder operator works just fine with floats as well.

The last operator is the exponentiation (or power) operator:

```
>>> 2 ** 3
8
>>> -3 ** 2
-9
>>> (-3) ** 2
9
```

Note that the exponentiation operator binds tighter than the negation (unary minus), so `-3**2` is in fact the same as `-(3**2)`. If you want to calculate `(-3)**2`, you must say so explicitly.

Large Integers

Python can handle really large integers:

```
>>> 10000000000000000000
10000000000000000000L
```

What happened here? The number suddenly got an `L` tacked onto the end.

■ **Note** If you're using a version of Python older than 2.2, you get the following behavior:

```
>>> 10000000000000000000
OverflowError: integer literal too large
```

The newer versions of Python are more flexible when dealing with big numbers.

Ordinary integers can't be larger than 2147483647 (or smaller than –2147483648). If you want really big numbers, you must use longs. A *long* (or *long integer*) is written just like an ordinary integer but with an `L` at the end. (You can, in theory, use a lowercase `l` as well, but that looks all too much like the digit 1, so I'd advise against it.)

In the previous example, Python converted the integer to a long, but you can do that yourself, too. Let's try that big number again:

```
>>> 10000000000000000000L
10000000000000000000L
```

Of course, this is only useful in old versions of Python that aren't capable of figuring this stuff out.

Well, can you do math with these monster numbers, too? Sure thing. Consider the following:

```
>>> 19871639871639816391 86L * 198763981726391826L + 23
3949766264320055676130001437 84791693659L
```

As you can see, you can mix long integers and plain integers as you like. In all likelihood, you won't have to worry about the difference between longs and ints unless you're doing type checking, as described in Chapter 7—and that's something you should almost never do.

Hexadecimals and Octals

To conclude this section, I should mention that hexadecimal numbers are written like this:

```
>>> 0xAF
175
```

and octal numbers like this:

```
>>> 010
8
```

The first digit in both of these is zero. (If you don't know what this is all about, just close your eyes and skip to the next section—you're not missing anything important.)

■ **Note** For a summary of Python's numeric types and operators, see Appendix B.

Variables

Another concept that might be familiar to you is *variables*. If math makes you queasy, don't worry: variables in Python are easy to understand. A variable is basically a name that represents (or refers to) some value. For example, you might want the name `x` to represent `3`. To make it so, simply execute the following:

```
>>> x = 3
```

This is called an *assignment*. We assign the value `3` to the variable `x`. Another way of putting this is to say that we bind the variable `x` to the value (or object) `3`. After you've assigned a value to a variable, you can use the variable in expressions:

```
>>> x * 2
6
```

Note that you need to assign a value to a variable before you use it. After all, it doesn't make any sense to use a variable if it doesn't represent a value, does it?

■ **Note** Variable names can consist of letters, digits, and underscore characters (_). A variable can't begin with a digit, so `Plan9` is a valid variable name, whereas `9Plan` is not.

Statements

Until now we've been working (almost) exclusively with expressions, the ingredients of the recipe. But what about statements—the instructions?

In fact, I've cheated. I've introduced two types of statements already: the `print` statement, and assignments. So, what's the difference between a statement and an expression? Well, an expression *is* something, while a statement *does* something (or, rather, tells the computer to do something). For example, `2*2` *is* `4`, whereas `print 2*2` *prints* `4`. What's the difference? After all, they behave very similarly. Consider the following:

```
>>> 2*2
4
>>> print 2*2
4
```

As long as you execute this in the interactive interpreter, the results are similar, but that is only because the interpreter always prints out the values of all expressions (using the same representation as `repr`—see the section "String Representations, str and repr" later in this chapter). That is not true of Python in general. Later in this chapter, you'll see how to make programs that run without this interactive prompt, and simply putting an expression such as `2*2` in your program won't do anything interesting.[5] Putting `print 2*2` in there, on the other hand, will print out `4`.

---

5. In case you're wondering—yes, it *does* do something. It calculates the product of 2 and 2. However, the result isn't kept anywhere or shown to the user; it has no *side effects*, beyond the calculation itself.

■ **Note** In Python 3.0, `print` is a function, which means you need to write `print(42)` instead of `print 42`, for example. Other than that, it works more or less like the statement, as described here.

The difference between statements and expressions may be more obvious when dealing with assignments. Because they are not expressions, they have no values that can be printed out by the interactive interpreter:

```
>>> x = 3
>>>
```

As you can see, you get a new prompt immediately. Something has changed, however; `x` is now bound to the value `3`.

This is a defining quality of statements in general: they change things. For example, assignments change variables, and `print` statements change how your screen looks.

Assignments are, perhaps, the most important type of statement in any programming language, although it may be difficult to grasp their importance right now. Variables may just seem like temporary "storage" (like the pots and pans of a cooking recipe), but the real power of variables is that you don't need to know what values they hold in order to manipulate them.[6] For example, you know that `x * y` evaluates to the product of `x` and `y`, even though you may have no knowledge of what `x` and `y` are. So, you may write programs that use variables in various ways without knowing the values they will eventually hold (or refer to) when the program is run.

Getting Input from the User

You've seen that you can write programs with variables without knowing their values. Of course, the interpreter must know the values eventually. So how can it be that we don't? The interpreter knows only what we tell it, right? Not necessarily.

You may have written a program, and someone else may use it. You cannot predict what values users will supply to the program. Let's take a look at the useful function `input`. (I'll have more to say about functions in a minute.)

```
>>> input("The meaning of life: ")
The meaning of life: 42
42
```

What happens here is that the first line (`input(...)`) is executed in the interactive interpreter. It prints out the string `"The meaning of life: "` as a new prompt. I type `42` and press

6. Note the quotes around storage. Values aren't stored in variables—they're stored in some murky depths of computer memory, and are referred to by variables. As will become abundantly clear as you read on, more than one variable can refer to the same value.

Enter. The resulting value of `input` is that very number, which is automatically printed out in the last line. That may not seem very useful, but look at the following:

```
>>> x = input("x: ")
x: 34
>>> y = input("y: ")
y: 42
>>> print x * y
1428
```

Here, the statements at the Python prompts (`>>>`) could be part of a finished program, and the values entered (`34` and `42`) would be supplied by some user. Your program would then print out the value `1428`, which is the product of the two. And you didn't have to know these values when you wrote the program, right?

■ **Note** This is much more useful when you save your programs in a separate file so other users can execute them. You learn to do that later in this chapter, in the section "Saving and Executing Your Programs."

SNEAK PEEK: THE IF STATEMENT

To make things a bit more fun, I'll give you a sneak peek of something you aren't really supposed to learn about until Chapter 5: the `if` statement. The `if` statement lets you perform an action (another statement) if a given condition is true. One type of condition is an equality test, using the equality operator `==`. Yes, it's a *double* equality sign. The single one is used for assignments, remember?

You simply put this condition after the word `if` and then separate it from the following statement with a colon:

```
>>> if 1 == 2: print 'One equals two'
...
>>> if 1 == 1: print 'One equals one'
...
One equals one
>>>
```

As you can see, nothing happens when the condition is false. When it is true, however, the following statement (in this case, a `print` statement) is executed. Note also that when using `if` statements in the interactive interpreter, you need to press Enter twice before it is executed. (The reason for this will become clear in Chapter 5—don't worry about it for now.)

So, if the variable `time` is bound to the current time in minutes, you could check whether you're "on the hour" with the following statement:

```
if time % 60 == 0: print 'On the hour!'
```

Functions

In the section on numbers and expressions, I used the exponentiation operator (**) to calculate powers. The fact is that you can use a *function* instead, called `pow`:

```
>>> 2**3
8
>>> pow(2,3)
8
```

A function is like a little program that you can use to perform a specific action. Python has a lot of functions that can do many wonderful things. In fact, you can make your own functions, too (more about that later); therefore, we often refer to standard functions such as `pow` as *built-in* functions.

Using a function as I did in the preceding example is called *calling* the function. You supply it with *parameters* (in this case, `2` and `3`) and it *returns* a value to you. Because it returns a value, a function call is simply another type of *expression*, like the arithmetic expressions discussed earlier in this chapter.[7] In fact, you can combine function calls and operators to create more complicated expressions:

```
>>> 10 + pow(2, 3*5)/3.0
10932.666666666666
```

■ **Note** The exact number of decimals may vary depending on which version of Python you are using.

Several built-in functions can be used in numeric expressions like this. For example, `abs` gives the absolute value of a number, and `round` rounds floating-point numbers to the nearest integer:

```
>>> abs(-10)
10
>>> 1/2
0
>>> round(1.0/2.0)
1.0
```

Notice the difference between the two last expressions. Integer division always rounds down, whereas `round` rounds to the nearest integer. But what if you want to round a given number down? For example, you might know that a person is 32.9 years old, but you would like to round that down to 32 because she isn't really 33 yet. Python has a function for this (called `floor`)—it just isn't available directly. As is the case with many useful functions, it is found in a *module*.

7. Function calls can also be used as statements if you simply ignore the return value.

Modules

You may think of modules as extensions that can be imported into Python to extend its capabilities. You import modules with a special command called (naturally enough) `import`. The function mentioned in the previous section, `floor`, is in a module called `math`:

```
>>> import math
>>> math.floor(32.9)
32.0
```

Notice how this works: we import a module with `import`, and then use the functions from that module by writing `module.function`.

If you want the age to be an integer (`32`) and not a float (`32.0`), you can use the function `int`:[8]

```
>>> int(math.floor(32.9))
32
```

■ **Note** Similar functions exist to convert to other types (for example, `long` and `float`). In fact, these aren't completely normal functions—they're *type objects*. I'll have more to say about types later. The opposite of `floor` is `ceil` (short for "ceiling"), which finds the smallest integral value larger than or equal to the given number.

If you are sure that you won't import more than one function with a given name (from different modules), you might not want to write the module name each time you call the function. Then you can use a variant of the `import` command:

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

After using `from module import function`, you can use the function without its module prefix.

■ **Tip** You may, in fact, use variables to refer to functions (and most other things in Python). For example, by performing the assignment `foo = math.sqrt`, you can start using `foo` to calculate square roots; for example, `foo(4)` yields `2.0`.

---

8. The `int` function/type will actually round down while converting to an integer, so when converting to an integer, using `math.floor` is superfluous; you could simply use `int(32.9)`.

cmath and Complex Numbers

The `sqrt` function is used to calculate the square root of a number. Let's see what happens if we supply it with a negative number:

```
>>> from math import sqrt
>>> sqrt(-1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in ?
    sqrt(-1)
ValueError: math domain error
```

or, on some platforms:

```
>>> sqrt(-1)
nan
```

■ **Note** nan is simply a special value meaning "not a number."

Well, that's reasonable. You can't take the square root of a negative number—or can you? Indeed you can. The square root of a negative number is an imaginary number. (This is a standard mathematical concept—if you find it a bit too mind-bending, feel free to skip ahead.) So why couldn't `sqrt` deal with it? Because it deals only with floats, and imaginary numbers (and complex numbers, the sum of real and imaginary numbers) are something completely different—which is why they are covered by a different module, `cmath` (for complex math):

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

Notice that I didn't use `from ... import ...` here. If I had, I would have lost my ordinary `sqrt`. Name clashes like these can be sneaky, so unless you really want to use the `from` version, you should probably stick with a plain `import`.

The value `1j` is an imaginary number. These numbers are written with a trailing `j` (or `J`), just like longs use `L`. Without delving into the theory of complex numbers, let me just show a final example of how you can use them:

```
>>> (1+3j) * (9+4j)
(-3+31j)
```

As you can see, the support for complex numbers is built into the language.

■ **Note** There is no separate type for imaginary numbers in Python. They are treated as complex numbers whose real component is zero.

Back to the __future__

It has been rumored that Guido van Rossum (Python's creator) has a time machine, because quite often when people request features in the language, the features have already been implemented. Of course, we aren't all allowed into this time machine, but Guido has been kind enough to build a part of it into Python, in the form of the magic module `__future__`. From it, we can import features that will be standard in Python in the future but that aren't part of the language yet. You saw this in the section about numbers and expressions, and you'll be bumping into it from time to time throughout this book.

Saving and Executing Your Programs

The interactive interpreter is one of Python's great strengths. It makes it possible to test solutions and to experiment with the language in real time. If you want to know how something works, just try it! However, everything you write in the interactive interpreter is lost when you quit. What you really want to do is write programs that both you and other people can run. In this section, you learn how to do just that.

First of all, you need a text editor, preferably one intended for programming. (If you use something like Microsoft Word, which I don't really recommend, be sure to save your code as plain text.) If you are already using IDLE, you're in luck. With IDLE, you can simply create a new editor window with File ➤ New Window. Another window appears, without an interactive prompt. Whew!

Start by entering the following:

```
print "Hello, world!"
```

Now select File ➤ Save to save your program (which is, in fact, a plain text file). Be sure to put it somewhere where you can find it later. You might want to create a directory where you put all your Python projects, such as `C:\python` in Windows. In a UNIX environment, you might use a directory like `~/python`. Give your file any reasonable name, such as `hello.py`. The `.py` ending is important.

■ **Note** If you followed the installation instructions earlier in this chapter, you may have put your Python installation in `~/python` already, but because that has a subdirectory of its own (such as `~/python/Python-2.5/`), this shouldn't cause any problems. If you would rather put your own programs somewhere else, feel free to use a directory such as `~/my_python_programs`.

Got that? Don't close the window with your program in it. If you did, just open it again (File ➤ Open). Now you can run it with Edit ➤ Run script, or by pressing Ctrl+F5. (If you aren't using IDLE, see the next section about running your programs from the command prompt.)

What happens? `Hello, world!` is printed in the interpreter window, which is exactly what we wanted. The interpreter prompt may be gone (depending on the version you're using), but you can get it back by pressing Enter (in the interpreter window).

Let's extend our script to the following:

```
name = raw_input("What is your name? ")
print "Hello, " + name + "!"
```

■ **Note** Don't worry about the difference between `input` and `raw_input`—I'll get to that.

If you run this (remember to save it first), you should see the following prompt in the interpreter window:

```
What is your name?
```

Enter your name (for example, `Gumby`) and press Enter. You should get something like this:

```
Hello, Gumby!
```

Fun, isn't it?

Running Your Python Scripts from a Command Prompt

Actually, there are several ways to run your programs. First, let's assume that you have a DOS window or a UNIX shell prompt before you, and that the directory containing the Python executable (called `python.exe` in Windows, and `python` in UNIX) or the directory *containing* the executable (in Windows) has been put in your `PATH` environment variable.[9] Also, let's assume that your script from the previous section (`hello.py`) is in the current directory. Then you can execute your script with the following command in Windows:

```
C:\>python hello.py
```

or UNIX:

```
$ python hello.py
```

As you can see, the command is the same. Only the system prompt changes.

■ **Note** If you don't want to mess with environment variables, you can simply specify the full path of the Python interpreter. In Windows, you might do something like this:

```
C:\>C:\Python25\python hello.py
```

Making Your Scripts Behave Like Normal Programs

Sometimes you want to execute a Python program (also called a *script*) the same way you execute other programs (such as your web browser or text editor), rather than explicitly using the

9. If you don't understand this sentence, you should perhaps skip the section. You don't really need it.

Python interpreter. In UNIX, there is a standard way of doing this: have the first line of your script begin with the character sequence `#!` (called *pound bang* or *shebang*) followed by the absolute path to the program that interprets the script (in our case Python). Even if you didn't quite understand that, just put the following in the first line of your script if you want it to run easily on UNIX:

```
#!/usr/bin/env python
```

This should run the script, regardless of where the Python binary is located.

■ **Note** In some operating systems if you install a recent version of Python (e.g., 2.5) you will still have an old one lying around (e.g.,1.5.2), which is needed by some system programs (so you can't uninstall it). In such cases, the `/usr/bin/env` trick is not a good idea, as you will probably end up with your programs being executed by the old Python. Instead, you should find the exact location of your new Python executable (probably called `python` or `python2`) and use the full path in the pound bang line, like this:

```
#!/usr/bin/python2
```

The exact path may vary from system to system.

Before you can actually run your script, you must make it executable:

```
$ chmod a+x hello.py
```

Now it can be run like this (assuming that you have the current directory in your path):

```
$ hello.py
```

If this doesn't work, try using `./hello.py` instead, which will work even if the current directory (`.`) is not part of your execution path.

If you like, you can rename your file and remove the `py` suffix to make it look more like a normal program.

What About Double-Clicking?

In Windows, the suffix (`.py`) is the key to making your script behave like a program. Try double-clicking the file `hello.py` you saved in the previous section. If Python was installed correctly, a DOS window appears with the prompt "What is your name?" Cool, huh?[10] (You'll see how to make your programs look better, with buttons, menus, and so on, later.)

There is one problem with running your program like this, however. Once you've entered your name, the program window closes before you can read the result. The window closes when the program is finished. Try changing the script by adding the following line at the end:

```
raw_input("Press <enter>")
```

10. This behavior depends on your operating system and the installed Python interpreter. If you've saved the file using IDLE in Mac OS X, for example, double-clicking the file will simply open it in the IDLE code editor.

Now, after running the program and entering your name, you should have a DOS window with the following contents:

```
What is your name? Gumby
Hello, Gumby!
Press <enter>
```

Once you press the Enter key, the window closes (because the program is finished). Just as a teaser, rename your file `hello.pyw`. (This is Windows-specific.) Double-click it as before. What happens? Nothing! How can that be? I will tell you later in the book—I promise.

Comments

The hash sign (#) is a bit special in Python. When you put it in your code, everything to the right of it is ignored (which is why the Python interpreter didn't choke on the `/usr/bin/env` stuff used earlier). Here is an example:

```
# Print the circumference of the circle:
print 2 * pi * radius
```

The first line here is called a *comment*, which can be useful in making programs easier to understand—both for other people and for yourself when you come back to old code. It has been said that the first commandment of programmers is "Thou Shalt Comment" (although some less charitable programmers swear by the motto "If it was hard to write, it should be hard to read"). Make sure your comments say significant things and don't simply restate what is already obvious from the code. Useless, redundant comments may be worse than none. For example, in the following, a comment isn't really called for:

```
# Get the user's name:
user_name = raw_input("What is your name?")
```

It's always a good idea to make your code readable on its own as well, even without the comments. Luckily, Python is an excellent language for writing readable programs.

Strings

Now what was all that `raw_input` and `"Hello, " + name + "!"` stuff about? Let's tackle the `"Hello"` part first and leave `raw_input` for later.

The first program in this chapter was simply

```
print "Hello, world!"
```

It is customary to begin with a program like this in programming tutorials. The problem is that I haven't really explained how it works yet. You know the basics of the `print` statement (I'll have more to say about that later), but what is `"Hello, world!"`? It's called a *string* (as in "a string of characters"). Strings are found in almost every useful, real-world Python program and have many uses. Their main use is to represent bits of text, such as the exclamation "Hello, world!"

Single-Quoted Strings and Escaping Quotes

Strings are values, just as numbers are:

```
>>> "Hello, world!"
'Hello, world!'
```

There is one thing that may be a bit surprising about this example, though: when Python printed out our string, it used single quotes, whereas we used double quotes. What's the difference? Actually, there is no difference:

```
>>> 'Hello, world!'
'Hello, world!'
```

Here, we use single quotes, and the result is the same. So why allow both? Because in some cases it may be useful:

```
>>> "Let's go!"
"Let's go!"
>>> '"Hello, world!" she said'
'"Hello, world!" she said'
```

In the preceding code, the first string contains a single quote (or an apostrophe, as we should perhaps call it in this context), and therefore we can't use single quotes to enclose the string. If we did, the interpreter would complain (and rightly so):

```
>>> 'Let's go!'
SyntaxError: invalid syntax
```

Here, the string is `'Let'`, and Python doesn't quite know what to do with the following `s` (or the rest of the line, for that matter).

In the second string, we use double quotes as part of our sentence. Therefore, we have to use single quotes to enclose our string, for the same reasons as stated previously. Or, actually we don't *have* to. It's just convenient. An alternative is to use the backslash character (`\`) to escape the quotes in the string, like this:

```
>>> 'Let\'s go!'
"Let's go!"
```

Python understands that the middle single quote is a character *in* the string and not the *end* of the string. (Even so, Python chooses to use double quotes when printing out the string.) The same works with double quotes, as you might expect:

```
>>> "\"Hello, world!\" she said"
'"Hello, world!" she said'
```

Escaping quotes like this can be useful, and sometimes necessary. For example, what would you do without the backslash if your string contained both single and double quotes, as in the string `'Let\'s say "Hello, world!"'`?

■ **Note** Tired of backslashes? As you will see later in this chapter, you can avoid most of them by using long strings and raw strings (which can be combined).

Concatenating Strings

Just to keep whipping this slightly tortured example, let me show you another way of writing the same string:

```
>>> "Let's say " '"Hello, world!"'
'Let\'s say "Hello, world!"'
```

I've simply written two strings, one after the other, and Python automatically concatenates them (makes them into one string). This mechanism isn't used very often, but it can be useful at times. However, it works only when you actually write both strings at the same time, directly following one another:

```
>>> x = "Hello, "
>>> y = "world!"
>>> x y
SyntaxError: invalid syntax
```

In other words, this is just a special way of writing strings, not a general method of concatenating them. How, then, do you concatenate strings? Just like you add numbers:

```
>>> "Hello, " + "world!"
'Hello, world!'
>>> x = "Hello, "
>>> y = "world!"
>>> x + y
'Hello, world!'
```

String Representations, str and repr

Throughout these examples, you have probably noticed that all the strings printed out by Python are still quoted. That's because it prints out the value as it might be written in Python code, not how you would like it to look for the user. If you use `print`, however, the result is different:

```
>>> "Hello, world!"
'Hello, world!'
>>> 10000L
10000L
>>> print "Hello, world!"
Hello, world!
>>> print 10000L
10000
```

As you can see, the long integer `10000L` is simply the number 10000 and should be written that way when presented to the user. But when you want to know what value a variable refers to, you may be interested in whether it's a normal integer or a long, for example.

What is actually going on here is that values are converted to strings through two different mechanisms. You can use both mechanisms yourself, through the functions `str` and `repr`. `str` simply converts a value into a string in some reasonable fashion that will probably be understood by a user, for example.[11] `repr` creates a string that is a representation of the value as a legal Python expression. Here are a few examples:

```
>>> print repr("Hello, world!")
'Hello, world!'
>>> print repr(10000L)
10000L
>>> print str("Hello, world!")
Hello, world!
>>> print str(10000L)
10000
```

A synonym for `repr(x)` is `` `x` `` (here, you use backticks, not single quotes). This can be useful when you want to print out a sentence containing a number:

```
>>> temp = 42
>>> print "The temperature is " + temp
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in ?
    print "The temperature is " + temp
TypeError: cannot add type "int" to string
>>> print "The temperature is " + `temp`
The temperature is 42
```

■ **Note** Backticks are removed in Python 3.0, so even though you may find backticks in old code, you should probably stick with `repr` yourself.

The first `print` statement doesn't work because you can't add a string to a number. The second one, however, works because I have converted `temp` to the string `"42"` by using the backticks. (I could have just as well used `repr`, which means the same thing, but may be a bit clearer. Actually, in this case, I could also have used `str`. Don't worry too much about this right now.)

In short, `str`, `repr`, and backticks are three ways of converting a Python value to a string. The function `str` makes it look good, while `repr` (and the backticks) tries to make the resulting string a legal Python expression.

---

11. Actually, `str` is a type, just like `int` and `long`. `repr`, however, is simply a function.

input vs. raw_input

Now you know what `"Hello, " + name + "!"` means. But what about `raw_input`? Isn't `input` good enough? Let's try it. Enter the following in a separate script file:

```
name = input("What is your name? ")
print "Hello, " + name + "!"
```

This is a perfectly valid program, but as you will soon see, it's a bit impractical. Let's try to run it:

```
What is your name? Gumby
Traceback (most recent call last):
  File "C:/python/test.py", line 2, in ?
    name = input("What is your name? ")
  File "<string>", line 0, in ?
NameError: name 'Gumby' is not defined
```

The problem is that `input` assumes that what you enter is a valid Python expression (it's more or less the inverse of `repr`). If you write your name as a string, that's no problem:

```
What is your name? "Gumby"
Hello, Gumby!
```

However, it's just a bit too much to ask that users write their name in quotes like this. Therefore, we use `raw_input`, which treats all input as raw data and puts it into a string:

```
>>> input("Enter a number: ")
Enter a number: 3
3
>>> raw_input("Enter a number: ")
Enter a number: 3
'3'
```

Unless you have a special need for `input`, you should probably use `raw_input`.

Long Strings, Raw Strings, and Unicode

Before ending this chapter, I want to tell you about a few other ways of writing strings. These alternate string syntaxes can be useful when you have strings that span several lines or contain various special characters.

Long Strings

If you want to write a really long string, one that spans several lines, you can use triple quotes instead of ordinary quotes:

```
print '''This is a very long string.
It continues here.
And it's not over yet.
"Hello, world!"
Still here.'''
```

You can also use triple double quotes, `"""like this"""`. Note that because of the distinctive enclosing quotes, both single and double quotes are allowed inside, without being backslash-escaped.

■ **Tip** Ordinary strings can also span several lines. If the last character on a line is a backslash, the line break itself is "escaped" and ignored. For example:

```
print "Hello, \
world!"
```

would print out `Hello, world!`. The same goes for expressions and statements in general:

```
>>> 1 + 2 + \
       4 + 5
12
>>> print \
      'Hello, world'
Hello, world
```

Raw Strings

Raw strings aren't too picky about backslashes, which can be very useful sometimes.[12] In ordinary strings, the backslash has a special role: it *escapes* things, letting you put things into your string that you couldn't normally write directly. For example, a new line is written `\n`, and can be put into a string like this:

```
>>> print 'Hello,\nworld!'
Hello,
world!
```

This is normally just dandy, but in some cases, it's not what you want. What if you wanted the string to include a backslash followed by an `n`? You might want to put the DOS pathname `C:\nowhere` into a string:

```
>>> path = 'C:\nowhere'
>>> path
'C:\nowhere'
```

This looks correct, until you print it and discover the flaw:

```
>>> print path
C:
owhere
```

12. Raw strings can be especially useful when writing regular expressions. More about those in Chapter 10.

Not exactly what we were after, is it? So what do we do? We can escape the backslash itself:

```
>>> print 'C:\\nowhere'
C:\nowhere
```

This is just fine. But for long paths, you wind up with a lot of backslashes:

```
path = 'C:\\Program Files\\fnord\\foo\\bar\\baz\\frozz\\bozz'
```

Raw strings are useful in such cases. They don't treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it:

```
>>> print r'C:\nowhere'
C:\nowhere
>>> print r'C:\Program Files\fnord\foo\bar\baz\frozz\bozz'
C:\Program Files\fnord\foo\bar\baz\frozz\bozz
```

As you can see, raw strings are prefixed with an `r`. It would seem that you can put anything inside a raw string, and that is almost true. Quotes must be escaped as usual, although that means that you get a backslash in your final string, too:

```
>>> print r'Let\'s go!'
Let\'s go!
```

The one thing you can't have in a raw string is a lone, final backslash. In other words, the last character in a raw string cannot be a backslash unless you escape it (and then the backslash you use to escape it will be part of the string, too). Given the previous example, that ought to be obvious. If the last character (before the final quote) is an unescaped backslash, Python won't know whether or not to end the string:

```
>>> print r"This is illegal\"
SyntaxError: invalid token
```

Okay, so it's reasonable, but what if you want the last character in your raw string to be a backslash? (Perhaps it's the end of a DOS path, for example.) Well, I've given you a whole bag of tricks in this section that should help you solve that problem, but basically you need to put the backslash in a separate string. A simple way of doing that is the following:

```
>>> print r'C:\Program Files\foo\bar' '\\'
C:\Program Files\foo\bar\
```

Note that you can use both single and double quotes with raw strings. Even triple-quoted strings can be raw.

Unicode Strings

The final type of string constant is the *Unicode string* (or Unicode *object*—they don't really belong to the same type as strings). If you don't know what Unicode is, you probably don't need to know about this. (If you want to find out more about it, you can go to the Unicode web site, `www.unicode.org`.) Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters,

including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following:

```
>>> u'Hello, world!'
u'Hello, world!'
```

As you can see, Unicode strings use the prefix `u`, just as raw strings use the prefix `r`.

■ **Note** In Python 3.0, all strings will be Unicode strings.

A Quick Summary

This chapter covered quite a bit of material. Let's take a look at what you've learned before moving on.

Algorithms: An algorithm is a recipe telling you exactly how to perform a task. When you program a computer, you are essentially describing an algorithm in a language the computer can understand, such as Python. Such a machine-friendly description is called a program, and it mainly consists of expressions and statements.

Expressions: An expression is a part of a computer program that represents a value. For example, `2+2` is an expression, representing the value `4`. Simple expressions are built from *literal values* (such as `2` or `"Hello"`) by using *operators* (such as `+` or `%`) and *functions* (such as `pow`). More complicated expressions can be created by combining simpler expressions (e.g., `(2+2)*(3-1)`). Expressions may also contain *variables*.

Variables: A variable is a name that represents a value. New values may be assigned to variables through *assignments* such as `x = 2`. An assignment is a kind of *statement*.

Statements: A statement is an instruction that tells the computer to *do* something. That may involve changing variables (through assignments), printing things to the screen (such as `print "Hello, world!"`), importing modules, or a host of other stuff.

Functions: Functions in Python work just like functions in mathematics: they may take some arguments, and they return a result. (They may actually do lots of interesting stuff before returning, as you will find out when you learn to write your own functions in Chapter 6.)

Modules: Modules are extensions that can be imported into Python to extend its capabilities. For example, several useful mathematical functions are available in the `math` module.

Programs: You have looked at the practicalities of writing, saving, and running Python programs.

Strings: Strings are really simple—they are just pieces of text. And yet there is a lot to know about them. In this chapter, you've seen many ways to write them, and in Chapter 3 you learn many ways of using them.

New Functions in This Chapter

Function	Description
`abs(number)`	Returns the absolute value of a number
`cmath.sqrt(number)`	Returns the square root; works with negative numbers
`float(object)`	Converts a string or number to a floating-point number
`help()`	Offers interactive help
`input(prompt)`	Gets input from the user
`int(object)`	Converts a string or number to an integer
`long(object)`	Converts a string or number to a long integer
`math.ceil(number)`	Returns the ceiling of a number as a float
`math.floor(number)`	Returns the floor of a number as a float
`math.sqrt(number)`	Returns the square root; doesn't work with negative numbers
`pow(x, y[, z])`	Returns `x` to the power of `y` (modulo `z`)
`raw_input(prompt)`	Gets input from the user, as a string
`repr(object)`	Returns a string representation of a value
`round(number[, ndigits])`	Rounds a number to a given precision
`str(object)`	Converts a value to a string

What Now?

Now that you know the basics of expressions, let's move on to something a bit more advanced: data structures. Instead of dealing with simple values (such as numbers), you'll see how to bunch them together in more complex structures, such as lists and dictionaries. In addition, you'll take another close look at strings. In Chapter 5, you learn more about statements, and after that you'll be ready to write some really nifty programs.