PART I
Introducing IronPython

- **CHAPTER 1:** Discovering IronPython
- **CHAPTER 2:** Understanding the IronPython Basics

1

Discovering IronPython

WHAT'S IN THIS CHAPTER?

- Understanding why you want to add IronPython to your developer toolbox
- Obtaining and installing IronPython on your machine
- Understanding some underlying basics of how IronPython works
- Using IronPython at the console and within a window
- Designing and building a simple application

IronPython: It sounds like some kind of metal snake infesting your computer, but it isn't. IronPython is the .NET version of the open source Python language (`http://www.python.org/`). Python is a dynamic language that can greatly enhance your programming experience, help you create applications in less time, and make the applications you create significantly more responsive to user needs. Of course, you've heard these promises before from other languages. This chapter helps you understand how IronPython delivers on these promises in specific situations. The smart developer soon learns that every language serves specific needs and might not work well in others. So this chapter isn't here to blow smoke at you — once you complete it, you'll understand the strengths and weaknesses of IronPython.

Of course, you'll need to obtain a copy of IronPython before you can use it because Visual Studio doesn't include IronPython as part of the default installation. This chapter helps you get IronPython installed on your system and tells you about some options you may want to install as well.

Once you have IronPython installed, you'll want to know a little about how it works. This chapter won't make you an IronPython guru who's familiar with every nuance of the underlying structural elements, but it will give you a good overview that will make the rest of the book a lot easier to understand. You'll put your new-found knowledge to the test by performing a few tasks at the IronPython console and within the IronPython windowed environment.

Finally, this chapter takes you through the process of creating a simple application. No, this isn't going to be the next great Windows application. It will be a little better than Hello World, but not much. The idea is to get you started doing something useful with IronPython. Don't worry; the examples will become a lot more interesting as the book progresses.

AN OVERVIEW OF IRONPYTHON

It surprises many developers to discover that computer languages are for humans, not for computers. A computer couldn't care less about which language you use, because it's all bits and bytes in the end anyway. Consequently, when you decide to learn another computer language, it really does pay to know what that language will do for you, the developer. Otherwise, there really isn't a point in making the effort.

One phrase you often hear when discussing Python (and by extension, IronPython) is "batteries included." Python has an immense standard library that addresses everything from working with ZIP files to interacting with the file system. You'll discover the details of working with the Standard Library in Chapter 6. For now, it's important to know that the Standard Library has a lot to offer and you may very well be able to build many of your applications without ever thinking about the .NET Framework.

As previously mentioned, IronPython is a .NET version of the Python language. For a .NET developer, using IronPython has the advantage of letting you create extensions using .NET (see Chapters 16 and 17 for details). In addition, you have full access to the .NET Framework (see Chapter 7 for details). You can work with IronPython and other .NET languages that you already know, which means that you can use the right tool for every job. However, IronPython has a few differences from the CPython implementation that everyone else uses (see Appendix A for details), which means that you can occasionally run into some odd compatibility problems when using IronPython. As with most things in life, advantages usually come with a few disadvantages.

You'll see Python appear in many guises when you begin using it. The original implementation of Python is CPython and that's the implementation that most developers target. In fact, you'll often see IronPython compared and contrasted with CPython throughout this book. It's important to remember that all these implementations attempt to achieve the same goal — full support of the Python standard. In most cases, all you really need to worry about is the IronPython implementation, unless you plan to use third-party libraries written for another Python implementation. This book helps you understand the use of CPython extensions in Appendix B.

There are some basic reasons that you want to use IronPython (or Python for that matter). The most important reason is that IronPython is a dynamic language, which means that it performs many tasks during run time, rather than compile time. Using a dynamic language means that your code has advantages of static languages, such as Visual Basic, in that it can more easily adapt to changing

environmental conditions. (You'll discover many other dynamic language advantages as the chapter progresses.) Unfortunately, you often pay for runtime flexibility with poorer performance — there's always a tradeoff between flexibility and performance.

Performance is a combination of three factors: speed, reliability, and security. When an application has a performance hit, it means a decrease in any of these three factors. When working with IronPython, there is a decrease in speed because the interpreter must compile code at run time, rather than at compile time. This speed decrease is partially offset by an improvement in reliability because IronPython applications are so flexible.

Dynamic languages provide a number of benefits such as the ability to enter several statements and execute them immediately to obtain feedback. Using a dynamic language also provides easier refactoring and code modification because you don't have to change static definitions throughout your code. It's even possible to call functions you haven't implemented yet and add an implementation later in the code when it's needed. Don't get the idea that dynamic languages are new. In fact, dynamic languages have been around for a very long time. Examples of other dynamic languages include the following:

- LISP (List Processing)
- Smalltalk
- JavaScript
- PHP
- Ruby
- ColdFusion
- Lua
- Cobra
- Groovy

Developers also assign a number of advantages specifically to the Python language (and IronPython's implementation of it). Whether these features truly are advantages to you depends on your perspective and experience. Many people do agree that Python provides these features:

- Support for the Windows, Linux/Unix, and Mac OS X platforms
- Managed support using both Java and .NET
- Considerable object-oriented programming (OOP) functionality that is easy to understand and use
- The capability to look within the code — .NET developers will know this as a strong form of reflection

- An extensive array of standard libraries
- Full library support using hierarchical packages (a concept that is already familiar to every .NET developer)
- Robust third-party libraries that support just about every need
- Support for writing both extensions and modules in both C and C++
- Support for writing extensions and modules using third-party solutions for both .NET (IronPython) and Java (Jython)
- Modular application development
- Error handling through exceptions (another concept familiar to any .NET developer)
- High-level dynamic data types
- Ease of embedding within applications as a scripting solution
- Procedural code that is relatively easy and natural to write
- Ease of reading and a clear syntax

All these features translate into increased developer productivity, which is something that dynamic languages as a whole supposedly provide (productivity is one of these issues that is hard to nail down and even harder to prove unless you resort to metrics such as lines of code, which prove useless when comparing languages). In addition to the great features that Python provides, IronPython provides a few of its own. The following list provides a brief overview of these features:

- Full access to the .NET Framework
- Usability within Silverlight applications
- Interactive console with full dynamic compilation provided as part of the product
- Accessibility from within a browser (see `http://ironpython.codeplex.com/Wiki/View.aspx?title=SilverlightInteractiveSession` for details)
- Full extensibility using the .NET Framework
- Complete source code available (see `http://ironpython.codeplex.com/SourceControl/ListDownloadableCommits.aspx` for details)

One of the negatives of working with IronPython, versus Python (in the form of CPython), is that you lose support for multiple platforms — you only have direct access to Windows. You can get around this problem using Mono (`http://www.mono-project.com/Main_Page`), but it isn't a straightforward fix and many developers will find it cumbersome. (Chapter 19 tells you more about working with Mono — a valuable solution for some Windows versions as well, such as Windows Server 2008 Server Core.) Of course, there isn't any way to get around the lack of Java support — you simply choose one virtual machine or the other. Appendix A lists more IronPython differences from CPython, most of which will cause compatibility and other issues for you.

An interesting use of IronPython is as an application testing tool. In fact, some developers use IronPython exclusively for this purpose. Chapter 18 tells you more about this exciting use of IronPython and demonstrates that using IronPython for this purpose really does make application testing considerably easier.

Don't get the idea that IronPython is going to restrict your use of familiar technologies. You can still create a Windows Forms application (see Chapter 8) and interact with COM (see Chapter 9). It's even possible to create command line (console) applications (see Chapter 10) and work with the Internet (see Chapter 11) just as you always have. What IronPython provides is another way to view problems that you must address using your applications. As with most languages, what you're getting is another tool that lets you create solutions in the least amount of time and with the fewest bugs.

GETTING IRONPYTHON

Before you can use IronPython, you need to get a copy of your own, install it, and check to make sure it works. Theoretically, you might want to obtain the source code and build your own version of IronPython, but most developers simply download the binaries and begin working with IronPython right away. The first three sections that follow tell you what you need to work with IronPython, how to obtain the software, and how to install it. You'll definitely want to read these sections.

The final two sections are completely optional. In fact, you may want to skip them for now and come back to them after you complete more chapters in the book. The first optional section tells you how to build your own copy of IronPython from the source. The second optional section tells you about third-party libraries.

There's a huge base of third-party libraries for IronPython. Generally, you don't need to install any third-party libraries to use this book. Everything you need to work with IronPython is included with the download you get from the CodePlex Web site. The only time you might need to work with third-party libraries is in Part IV. You'll receive specific instructions in the Part IV chapters for any required third-party libraries, so you only need to read "Using Third-Party Libraries" if you plan to work with third-party libraries immediately.

Understanding the IronPython Requirements

As with any software, IronPython has basic system requirements you must meet before you can use it. It turns out that there are actually two versions of IronPython 2.6 — one for the .NET Framework 2.0, 3.0, and 3.5, and a second for the .NET Framework 4.0. Here are the requirements for the .NET Framework 2.0, 3.0, and 3.5 version.

- The .NET Framework 2.0, 3.0, and 3.5

- (Optional) Visual Studio 2005 or Visual Studio 2008 (your system must meet the prerequisites for this software)
- (Optional) .NET Framework 2.0 Software Development Kit (SDK)

You need only the optional requirements if you plan to build IronPython 2.6 from the source code. Here are the requirements for the .NET Framework 4.0 version (again, the optional requirements are there if you want to build IronPython from source code).

- The .NET Framework 4.0
- (Optional) Visual Studio 2010

Getting the Software

As with most open source software, you have a number of choices when it comes to downloading IronPython. For the sake of your sanity, the best choice when starting with IronPython is to download the binary version of the product from `http://ironpython.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=30315`. You'll see the Microsoft Installer (MSI) link right below the Recommended Download link as IronPython-2.6.msi. If you really must save the few seconds downloading the MSI version, select the IronPython-2.6-Bin.zip link instead.

It's also possible to compile IronPython from the source code. If you want to use this option, select the IronPython-2.6-Src.zip link. You must have a copy of Visual Studio installed on your system to use this option. The "Building the Binaries from Scratch" section of the chapter describes how to build a version from scratch, but this process truly isn't for the IronPython beginner and doesn't serve much of a purpose unless you plan to add your own enhancements.

Most developers will likely use the standard version of IronPython that works with the .NET Framework 3.5 and earlier. However, you might need some of the new features in the .NET Framework 4.0, such as the C# `dynamic` keyword, which is part of the Dynamic Language Runtime (DLR) (`http://dlr.codeplex.com/`). The section "Understanding the Dynamic Language Runtime" later in this chapter tells you more about this .NET 4.0 feature. You can obtain this version of IronPython at `http://ironpython.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=27320`. The examples in this book will work with either version of IronPython 2.6, except where noted (where I'm demonstrating how to work with the DLR).

Performing the Installation

This section assumes that you've downloaded the MSI file to make life easy for yourself. This procedure works equally well for either version of IronPython 2.6 so you can use it for a DLR install as well. The following instructions help you get IronPython installed on your machine.

1. Double-click the MSI file you downloaded from the CodePlex Web site. You'll see the usual Welcome page — click Next to get past it.

2. Read the licensing agreement, check I Accept the Terms in the License Agreement, and then click Next. You'll see the Custom Setup dialog box shown in Figure 1-1 where you can select the IronPython features you want to install. At a minimum, you must install the Runtime. The Documentation, Standard Library, and Tools features are also strongly recommended. This book assumes that you've installed all the features. However, you might want to install just those features you actually need for a production setup (you might not actually need the samples).

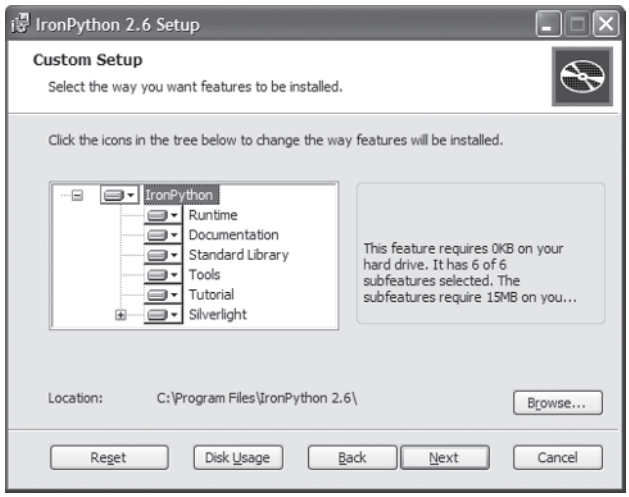

FIGURE 1-1: Choose the features you want to install.

When you perform a DLR installation, you'll see a Do Not NGen Installed Binaries option on the Custom Setup dialog box. Using the Native Image Generator (NGen) can greatly improve application performance, as described at `http://msdn.microsoft.com/en-us/magazine/cc163610.aspx`. *Earlier versions of IronPython didn't use NGen to build the libraries for you by default. You had to use a special command line to obtain the NGen feature (*`msiexec /qn /i "IronPython.msi" NGENDLLS=True`*). The default setup for IronPython 2.6 is to use NGen to build the libraries.*

However, using NGen also binds the binaries to the local machine, which may not be what you want when working in a dynamic environment. Consequently, if you plan to use DLR more than you plan to use other IronPython features, you might want to check the Do Not NGen Installed Binaries option.

3. Select the features you want to install. Click Next. You'll see a summary dialog box that simply states you're ready to install IronPython.
4. Click Install. MSI will begin the installation process. At some point, you'll see an installation completion screen.
5. Click Finish. You should be ready to begin working with IronPython at this point.

Building the Binaries from Scratch

You may eventually want to build the IronPython binaries from scratch. The normal reason to perform this task is to create a special version of IronPython that meets specific needs. A company may want to add extensions or special features to IronPython. Because you have the source code, it's acceptable to create a custom version of IronPython for yourself — one that contains any feature set you deem necessary to get your work completed. So have fun molding IronPython and then sharing your modifications with others. In order to perform this task, you must have a copy of Visual Studio (you must have Visual Studio 2010 to build a DLR version of IronPython). The following steps tell you how to build the IronPython 2.6 binaries from scratch.

1. Download the source code file, such as IronPython-2.6-Src.zip.
2. Extract the files into a folder. The example assumes that you extracted the files into the root directory of your hard drive into `\IronPython-2.6`.
3. Locate the `\IronPython-2.6\Src` directory and open the IronPython.sln solution using Visual Studio. Visual Studio will load the required files, and you'll see them in Solution Explorer, as shown in Figure 1-2. Figure 1-2 shows that IronPython consists of a number of projects — you must compile the entire solution to obtain a workable group of DLLs.
4. Make any required changes to the source code.
5. Choose Build ➪ Build Solution. Visual Studio creates the required DLLs, ready for testing.

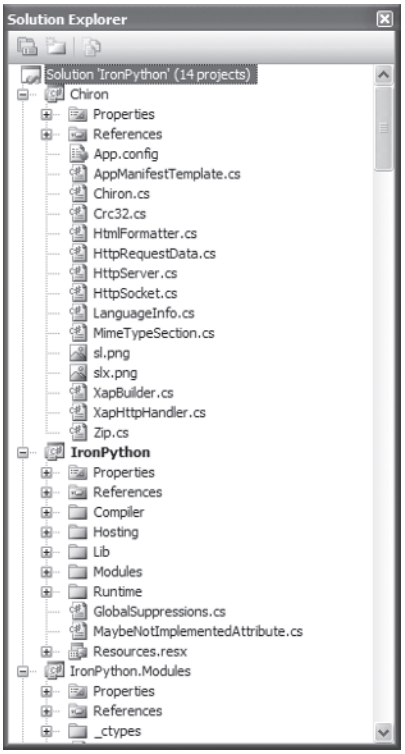

FIGURE 1-2: IronPython consists of multiple projects, so you must compile the entire solution.

Using Third-Party Libraries

Python is an extremely flexible language and enjoys strong third-party support. In fact, you can find lists of these libraries in various places on the Internet. Here are a few places to check:

- `http://code.google.com/appengine/docs/python/tools/libraries.html`
- `http://www.amaltas.org/show/third-party-python-libraries-and-frameworks.html`
- `http://dakrauth.com/blog/entry/third-party-python-libraries-interest/`

IronPython is a complex product. If you fail to compile the entire solution every time you make a change, you could end up with an unworkable group of DLLs due to interactions. It's important to build everything so that any changes propagate properly.

You should be able to use some third-party libraries with IronPython. At the time of this writing, you won't actually find any usable third-party libraries. However, you should check `http://www.ironpython.info/index.php/Third-Party_Library_Compatibility` from time-to-time to discover whether there are any third-party libraries that do work with IronPython. It's important to note that this list represents only tested libraries — you may find other third-party libraries that do work with the current version of IronPython.

UNDERSTANDING THE DYNAMIC LANGUAGE RUNTIME

IronPython is a dynamic language, yet the Common Language Runtime (CLR) is a static environment. While you can build a compiler that makes it possible to use a dynamic language with CLR, as was done for IronPython 1.0, you'll find that certain functionality is missing because CLR simply doesn't understand dynamic languages. Consequently, Microsoft started the Dynamic Language Runtime (DLR) project (see `http://dlr.codeplex.com/` for additional information). DLR sits on top of CLR and performs a level of interpretation that offers additional functionality for dynamic languages. By relying on DLR, IronPython gains access to the following support:

- Shared dynamic type support
- Shared hosted model
- Quick dynamic code generation
- Interaction with other dynamic languages
- Improved interaction with static languages such as C# and Visual Basic.NET (see Chapters 15, 16, and 17 for details)
- Shared sandbox security model and browse integration

DLR is now part of the .NET Framework 4.0. (In fact, you'll discover the details of this integration in Chapter 14.) Consequently, you can begin accessing these features immediately when using Visual Studio 2010 without having to install any additional support. Microsoft currently supports these languages using DLR:

- IronPython
- IronRuby
- JavaScript (EcmaScript 3.0)
- Visual Basic

Silverlight also provides support for DLR and there's even a special SDK for Silverlight DLR. You can discover more about this SDK at `http://silverlight.net/learn/dynamic-languages/`. The relevance of Silverlight support for this book is that you can now use IronPython as part of your Silverlight solution as described in Chapter 11. You can summarize the benefits of using DLR as follows:

- Makes it easier to port dynamic languages to the .NET Framework
- Lets you include dynamic features in static languages

- Creates an environment where sharing of objects and libraries between languages is possible
- Makes it possible to perform fast dynamic dispatch and invocation of objects

This section provides a good overview of DLR. You'll discover additional details about DLR as the book progresses. However, if you'd like to delve into some of the architectural details of DLR, check out the article at `http://msdn.microsoft.com/library/dd233052.aspx`.

USING THE IRONPYTHON CONSOLE

The IronPython console is the best place to begin working with IronPython. You can enter a few statements, test them out, and then work out additional details without too many consequences. In addition, because the console is interactive, you obtain immediate feedback, so you don't have to wait for a compile cycle to discover that something you're doing is completely wrong. In fact, even after you've mastered IronPython, you'll find that you use the console to try things out. Because IronPython is a dynamic language, you can try things without worrying about damaging an application. You can test things quickly using the console and then include them in your application. The following sections describe the IronPython console and how to use it. Expect to see the IronPython console in future chapters.

Opening and Using the Default Console

The IronPython console is an application provided with the default installation. You access it using the Start ➪ Programs ➪ IronPython 2.6 ➪ IronPython Console command. The console, shown in Figure 1-3, looks something like a command prompt, but it isn't.

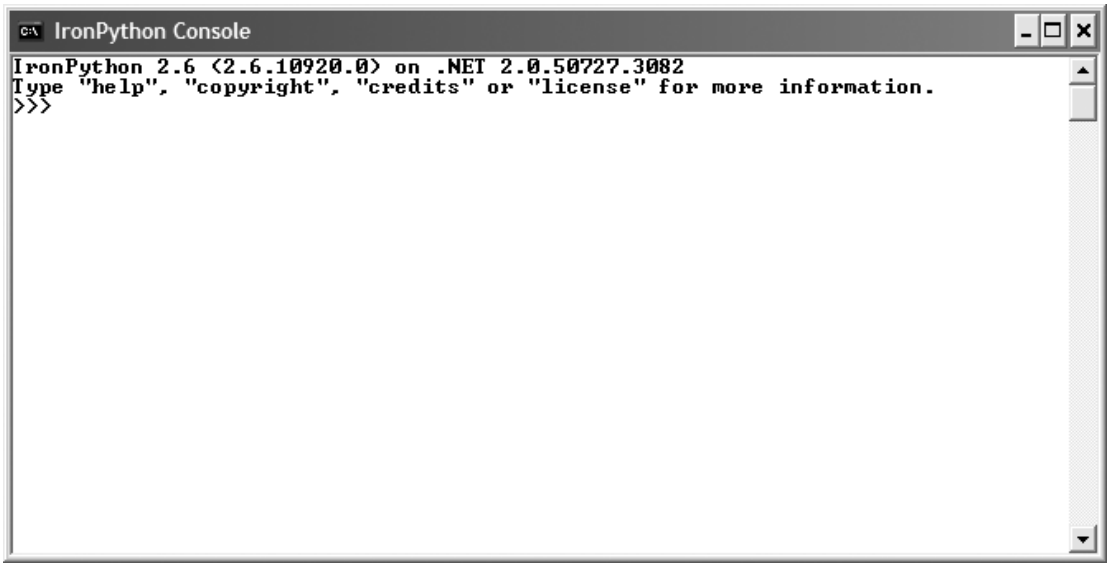

FIGURE 1-3: The IronPython console looks something like a command prompt.

Notice that the top of the window tells you which version of IronPython you're using and which version of the .NET Framework it's running on. This is important information because it helps you understand the IronPython environment and what limitations you have when working with IronPython. Below this first line, you'll see some commands that Microsoft thought you might find useful. The "Getting Help with Any Function" section of the chapter tells you more about the Help command.

To use the console, simply type the commands you want to issue. When you're done, IronPython will execute the commands and output any result you requested. A command need not be a function call or an object instantiation as it is in other languages. For example, type `2 + 2` right now and then press Enter. You'll see the result of this simple command, as shown in Figure 1-4.

FIGURE 1-4: IronPython is dynamic and the console is interactive.

Whenever you want to end a particular task, such as working with Help, press Enter a second time. The console will take you to the previous level of interaction.

Getting Help with Any Function

You can get help with any function in the console. If you simply type help and press Enter in the console, IronPython tells you how to request interactive help or help about a specific object. To begin interactive help, type `help()` and press Enter. You'll see the interactive help display shown in Figure 1-5.

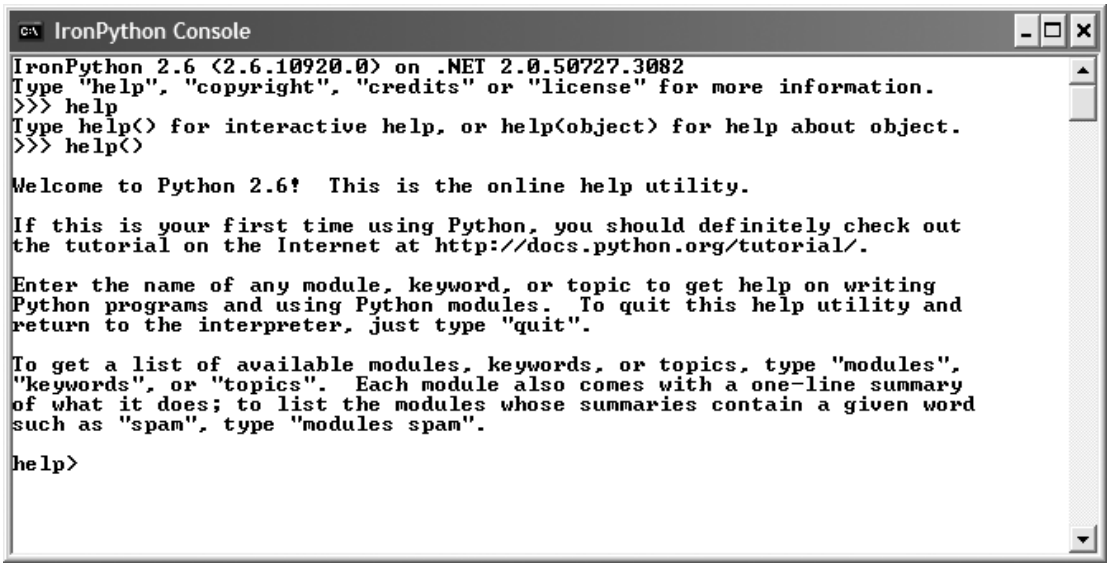

FIGURE 1-5: Interactive help lets you ask questions about IronPython.

Let's say you have no idea of what you want to find. Console help provides you with a list of words you can type to get general help. These terms are:

- Modules
- Keywords
- Topics

Type any of these terms and press Enter. You'll see a list of additional words you can type, as shown in Figure 1-6 for modules. Using this technique, you can drill down into help and locate anything you want. In fact, it's a good idea to spend some time in help just to see what's available. Even advanced developers can benefit from this approach — I personally follow this approach when I have time to increase my level of knowledge about all of the languages I use.

IronPython will constantly refer you to the online help for Python. So you might as well check it out now. You'll find a good Python tutorial at `http://docs.python.org/tutorial/`*. While you're at it, there's also a good IronPython-specific tutorial that comes with your installation. Simply choose Start ➪ Programs ➪ IronPython 2.6 ➪ IronPython Tutorial. Although these sources of help are useful, you'll get a much better start working through the examples in the book.*

You might know about the topic you want to find. For example, you might know that you want to print something to screen, but you don't quite know how to use print. In this case, type `help('print')` and press Enter. Figure 1-7 shows the results. You see complete documentation about the `print` keyword.

Understanding the IPY.EXE Command Line Syntax

When you open a console window, what you're actually doing is executing IPY.EXE, which is the IronPython interpreter. You don't have to open a console window to use IPY.EXE. In fact, you normally won't. It's possible to execute IronPython applications directly at the command line. The following sections discuss IPY.EXE in more detail.

Adding IPY.EXE to the Windows Environment

Before you can use IPY.EXE effectively, you need to add it to the Windows path statement. The following steps provide a brief procedure.

1. Open the Advanced tab of the Computer (or My Computer) applet.
2. Click Environment Variables. You'll see an Environment Variables dialog box.
3. Highlight Path in the System Variables list. Click Edit. You'll see the Edit Environment Variable dialog box.

FIGURE 1-6: Drill down into help to find topics of interest.

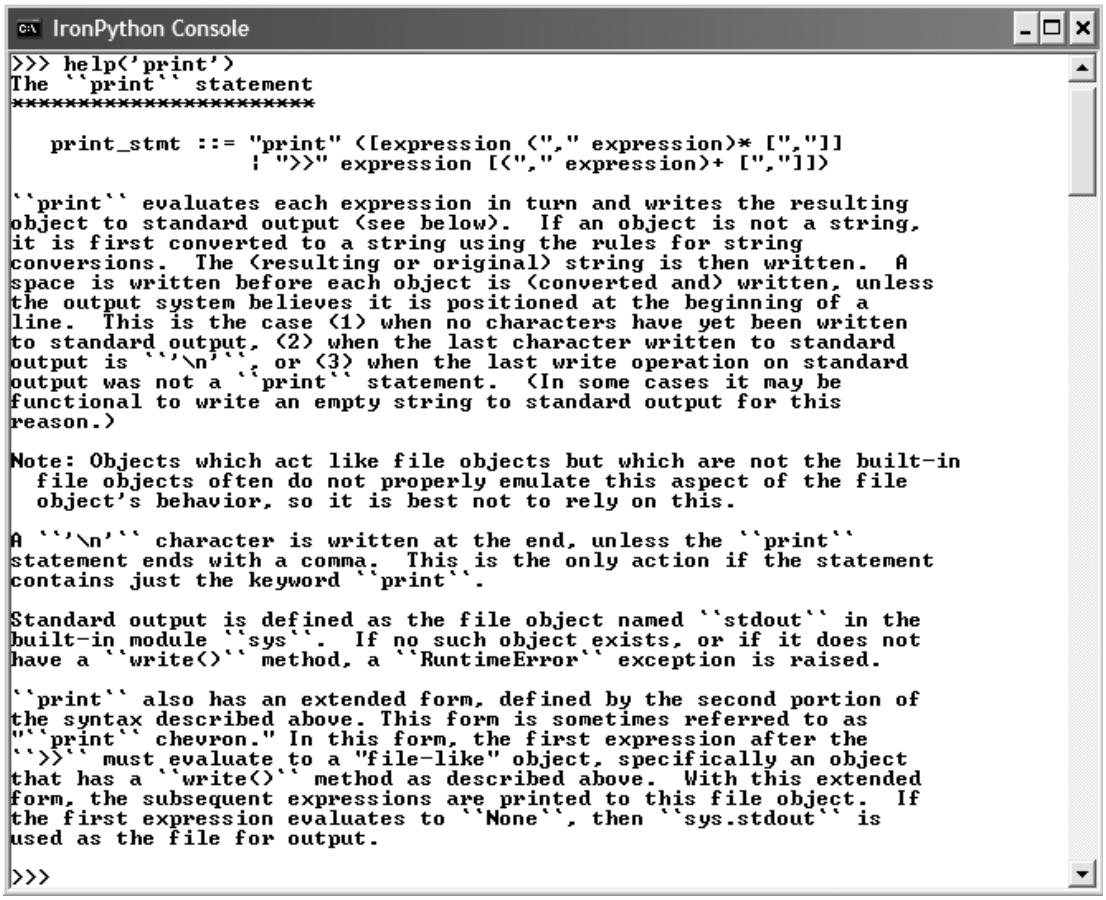

FIGURE 1-7: The console also provides the means to obtain precise help about any module, keyword, or topic.

4. Select the end of the string that appears in the Variable Value field. Type `;C:\Program Files\IronPython 2.6` and click OK. Make sure you modify this path to match your IronPython configuration.
5. Click OK three times to close the Edit System Variable, Environment Variables, and System Properties dialog boxes. When you open a command prompt, you'll be able to access the IronPython executables.

Executing an Application from the Command Prompt

Normally, you execute an application by typing IPY <Python Filename> and pressing Enter. Give it a try now. Open a command prompt, type `CD \Program Files\IronPython 2.6\Tutorial`, and press Enter. You're in the sample files supplied by IronPython. Type `IPY WFDemo.py` and press Enter. You'll see a window displayed. When you click your mouse in the window, you see the word Hello

displayed at each click point, as shown in Figure 1-8. If you look at the command prompt window at this point, you'll see that the mouse cursor is blinking but you can't type anything because the command prompt is waiting for the IronPython interpreter to end. When you click the Close button, the application ends and you can again type something at the command prompt.

Understanding the IPY.EXE Standard Command Line Switches

Sometimes you need to provide `IPY.EXE` with more information about a particular application. In this case, you can use one of the command line switches shown in the following list to provide `IPY.EXE` with the required information. It's important to note that the command line switches are case sensitive; `-v` isn't the same as `-V`.

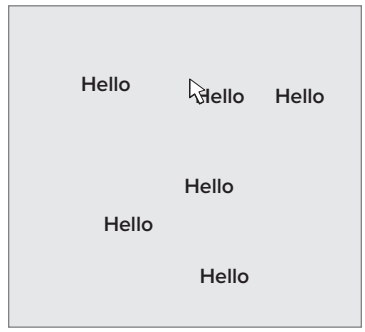

FIGURE 1-8: The WFDemo shows that you can create windowed environments for IronPython applications.

`-3`: Forces the interpreter to warn about Python 3 compatibility issues in your application.

`-c cmd`: Specifies a command you want to execute. This command line switch must appear last on the line because anything after this command line switch is interpreted as a command you want to execute. For example, if you type `ipy -c "print ('Hello')"`, the interpreter will output the word Hello.

`-D`: Enables application debugging.

`-E`: Ignores any environment variables that you specified as part of the Windows environment variable setup or on the command line after you started it. Some applications may not run after you use this command line switch because they won't be able to find modules and other files they need.

`-h`: Displays a complete list of the command line arguments.

`-i`: Displays the console after running the script. You can then inspect the results of the script using console commands.

`-m module`: Runs library module as a script.

`-O`: Tells the interpreter to generate optimized code, which means you can't perform debugging, but the application will run faster.

`-OO`: Removes all of the doc strings and applies `-O` optimizations so that the application runs even faster than using the `-O` command line switch alone.

`-Q arg`: Specifies use of one of several division options. You can use any of these values.

- **`-Qold`** (default)**:** The precision of the output depends on the operators used. For example, if you divide two integers, you get an integer as output.
- **`-Qwarn`:** Outputs warnings about a loss of precision when performing division using integers.
- **`-Qwarnall`:** Outputs warnings about all uses of the classic division operator.
- **`-Qnew`:** The output is always a precise floating point fraction.

-s: Specifies that the interpreter shouldn't add the user site directory to `sys.path`.

-S: Specifies that the interpreter shouldn't imply that it should execute the `import site` command on initialization.

-t: Outputs warnings about inconsistent tab usage, which can lead to code interpretation problems.

-tt: Outputs errors for inconsistent tab usage. Inconsistent tab usage can lead to code interpretation problems, which can result in hard-to-locate bugs.

-u: Provides unbuffered stdout and stderr devices. Typically, the interpreter uses buffering to provide better application performance.

-v: Specifies that the interpreter should provide verbose output, which means that you can see everything going on in the background. You can also obtain this result by using `PYTHONVERBOSE=x` (where x is a True or False environment variable).

-V: Prints the version number and exits. This option is useful when you want to be sure you're using the correct version of IronPython for your application.

-W *arg*: Defines the kind of warning control. Specifying these command line switches tells the interpreter to add the specified warning messages to the output. (Don't worry too much about these warnings — you learn more about them in Chapter 12.) You can use any of these values:

- **-W*action*:** Actions are one of the following strings: `error` (turns matching warnings into exceptions), `ignore` (never prints matching warnings), `always` (always prints matching warnings), `default` (prints the first occurrence of a warning for each location where the interpreter issues the warning), `module` (prints the first occurrence of a warning for each module where the error occurs), and `once` (prints only the first occurrence of a warning no matter where it appears).
- **-W*message*:** Messages are Regular Expressions that define which warning messages to match.
- **-W*category*:** Categories specify the class of the warning message.
- **-W*module*:** Modules are Regular Expressions that define which module to match.
- **-W*lineno*:** Line numbers are integer values that specify a line number to match. Using 0 matches all line numbers.

-x: Skips the first line of the source code, which may have special instructions that you don't need for the current session.

IronPython doesn't support all of the CPython command line switches. Consequently, you may find that a batch file written to execute a CPython application won't work properly with IronPython. For example, IronPython doesn't appear to support the PYTHONHOME environment variable. All IronPython environment variables begin with IRON, so you need to modify batch files to include this first word as part of any environmental variable setup.

Working with the –X: Command Line Switches

In addition to the standard command line switches, you also have access to the –X: command line switches, which configure the IronPython interpreter. The following list describes each of the configuration options:

`-X:AutoIndent`: Enables auto-indenting in the read-evaluation-print loop (REPL).

`-X:ColorfulConsole`: Enables ColorfulConsole support.

`-X:Debug`: Enables application debugging. This option is preferred over the `-D` command line switch because it's newer and will enjoy a longer support period.

`-X:EnableProfiler`: Enables profiling support in the compiler, which helps you optimize your applications.

`-X:ExceptionDetail`: Enables ExceptionDetail mode, which gives you more information about every exception that occurs, making it easier to locate the source of the problem (but filling the screen much faster as well).

`-X:Frames`: Enables basic `sys._getframe()` support.

`-X:FullFrames`: Enables `sys._getframe()` with access to local objects and variables.

`-X:GCStress`: Specifies the garbage collector (GC) stress level. Stressing the GC can point out potential resource problems in your application.

`-X:LightweightScopes`: Generates optimized scopes that are easier for the GC to collect. Optimizing GC functionality tends to improve the overall performance (both speed and reliability) of your application.

`-X:MaxRecursion`: Determines the maximum recursion level within the application. Recursion can use a lot of system resources, so controlling the amount of recursion tends to reduce resource usage by applications that rely on recursion. Of course, reducing the recursion levels can also cause application exceptions.

`-X:MTA`: Runs the application in a multithreaded apartment (MTA).

`-X:NoAdaptiveCompilation`: Disables the adaptive compilation feature.

`-X:PassExceptions`: Tells the interpreter not to catch exceptions that are unhandled by script code.

`-X:PrivateBinding`: Enables binding to private members.

`-X:Python30`: Enables available Python 3.0 features, such as classic division (where dividing two integers produces an integer result).

`-X:ShowClrExceptions`: Displays the Common Language Specification (CLS) exception information.

`-X:TabCompletion`: Enables TabCompletion mode.

`-X:Tracing`: Enables support for tracing all methods even before the code calls `sys.settrace()`.

Modifying the IPY.EXE Environment Variables

IPY also supports a number of environment variables. The following list describes each of these environment variables.

IRONPYTHONPATH: Specifies the path to search for modules used within an application

IRONPYTHONSTARTUP: Specifies the name and location of the startup module

Exiting the IronPython Interpreter

Eventually, you'll want to leave the console. In order to end your session, simply type `exit()` and press Enter. As an alternative, you can always press Ctrl+Z and then Enter. The console will close.

USING THE IRONPYTHON WINDOWED ENVIRONMENT

IronPython also provides access to a windowed environment, but you can't access it from the start menu. Instead, you must provide a shortcut to the file you want to run or open a command prompt and start the application manually. The windowed environment simply provides a GUI interface for working with IronPython, but doesn't do anything else for you. You start the windowed environment by using IPYW .EXE. If you type `IPYW` and press Enter, you see the command line switch help shown in Figure 1-9.

As you can see from Figure 1-9, the windowed environment supports the same command line switches as the character mode command line version. However, you can't use the windowed environment to run the interpreted console environment, which is a shame because many developers would prefer working in the nicer environment. To see that the windowed environment works the same way as the standard console, type `IPYW WFDemo.py` and press Enter. You'll see the test application shown earlier in Figure 1-8.

FIGURE 1-9: The windowed version supports the same features as the command line version.

CREATING YOUR FIRST APPLICATION

After all this time, you might have started wondering whether you would get to write any code at all in this chapter. The first application won't be very fancy, but it'll be more than a simple Hello World kind of application. You can use any editor that outputs pure text in this section. Notepad will work just fine. Listing 1-1 shows the code you should type in your editor.

LISTING 1-1: A simple first application that multiplies two numbers

```
def mult(a, b):
    return a * b

print('5 * 10 ='),
print(mult(5, 10))
```

Just five lines, including the blank line between the function and the main code, are all you need for this example. Functions begin with the `def` keyword. You then give the function a name, `mult` in this case, followed by a list of arguments (if any) — `a` and `b` for this example.

Don't worry too much about the details of the IronPython language just yet. Part II of the book provides you with a good overview of all the language details. As the book progresses, you'll be exposed to additional language details. By the time you reach the end of the book, you'll be working with some relatively complex examples.

The content of the function is indented with a tab. In this case, the function simply returns the value of multiplying `a` by `b`. Except for the indentation requirement, this could easily be a function written in any other language.

The main code section comes next. In this case, the code begins by printing 5 * 10 =. Notice that you enclose the string values in single quotes. The function call ends with an odd-looking comma. This comma tells the interpreter not to add a `/n` (newline) character after the `print()` call.

At this point, the code calls `print()` a second time, but it calls `mult()` instead of writing text directly. The output of `mult()` is an integer, which IronPython automatically converts to a string for you and then prints out. You'll find that IronPython does a lot of work for you in the background — dynamically (as explained earlier in the chapter).

Save the code you've typed into Notepad as `MyFirst.py`. Make sure you choose All Files in the Save As Type field so that Notepad doesn't add a .txt extension to the output. To execute this example, type `IPY MyFirst.py` at the command line and press Enter. Figure 1-10 shows the output from this quick example.

FIGURE 1-10: The output of the example is a simple equation.

USING IRONPYTHON CONSTRUCTIVELY

This chapter has introduced you to IronPython. You should have a good understanding of why you want to use IronPython and how it differs from other, static .NET languages. Dynamic languages have a special place in your toolbox. They aren't the answer to every need, but they can address specific needs — just as other languages address the needs for which they were built. At the end of the day, the computer doesn't care what language you use — the computer simply cares how that language is translated into the bits and bytes it requires to do something useful. Languages address human needs and it's important to keep that in mind.

Before you do anything else, make sure you get IronPython installed on your system and test the installation out using the examples in this chapter. If you're getting some weird result or nothing at all, you might have a bad installation. Once you know that you do have a good installation, try playing around with the example application in the "Creating Your First Application" section of the chapter. Work with this application as a means of working with the tools discussed in the "Using the IronPython Console" and "Using the IronPython Windowed Environment" sections of the chapter.

At this point, you really don't know too much about the Python language or the IronPython implementation of that language. However, you probably do know something about other .NET languages, and that's a good starting point. Chapter 2 builds on the information you've learned in this chapter and also builds on your personal knowledge of the .NET Framework. In Chapter 2, you begin building knowledge about IronPython so you can see what an interesting language it is and so you can also begin to understand the example in the "Creating Your First Application" section of the chapter. When you get done with Chapter 2, you may want to take another look at the sample application — you'll be surprised to discover that you really do know how the example works.