CHAPTER 2

The IDE: Eclipsing the Command Line

There are two main ways you can work with Python: through the command line or through an IDE. Both have their distinct advantages and disadvantages. I'm not going to give short shrift to either, but this chapter is mostly about using the Eclipse IDE to work with Python.

To truly take advantage of agile development methodologies, chances are you're going to want to use an IDE. IDEs offer a range of features that are at best poorly implemented in command-line tools. A short list includes a wealth of code navigation features, intuitive auto-completion, refactoring support, revision control integration, automatic builds, polished debugging, integration with unit testing tools, language-aware editors, jobs contexts, and ticket system integration. All of these features are useful, but when integrated they deliver more than the sum of their parts.

I've chosen the Eclipse IDE for the purposes of this book. Eclipse has a number of things going for it. It's free, it's widely used (I've seen it in most companies I've visited), and it's available on just about any platform you could desire (I'm almost certain that it will run on HP calculators these days). It's extensible, and it has Python support. It also has a few features that others lack, notably a job system called Mylyn (formerly known as Mylar—the name has changed, but the functionality remains the same). It also has a plug-in architecture that allows users to write custom extensions.

WHAT IS A JOB MANAGEMENT SYSTEM, AND WHY DO I NEED ONE?

A large project will have hundreds or thousands of files. Typically, each task that you work on will have only a small subset of these open at a time. You'll be working on a few tasks on and off, and you may have to go back and forth between them. These tasks will have few if any files in common, so you'll either end up with many tens of editors open or spend a great deal of time opening and closing them to keep your workspace clean.

Even though you'll have many files open, you'll typically only reference a few classes or methods. Eclipse offers navigation panes to help locate specific program elements such as files, classes, methods, and functions; but when working with a large program, finding a specific element is still troublesome. It's a bit like finding one particular piece in a big box of Lego parts.

Mylyn addresses both of these issues. It defines *tasks*, which represent units of work. These tasks may be defined locally, or they may reference external tickets in a defect-tracking system such as Bugzilla or Jira. One task is active at a time. Mylyn tracks program elements as you work on them, and it associates these with the active task. This collection of elements is referred to as the *context*.

When you activate a new task, the current task is deactivated. All of the editors associated with it are closed, and all of the editors associated with the newly activated task are opened. It also restricts navigation panes to show only those packages, directories, files, classes, methods, and functions that are associated with the active task. This filtering is turned on and off on a pane-by-pane basis by toggling an icon in the pane's menu bar.

In addition to context, tasks also have associated planning information. This information includes priority, severity, and scheduling data such as expected delivery date. Although not perfect, Mylyn is a major innovation in IDE interfaces.

A large ecosystem has grown up around Eclipse and its plug-in architecture. There are plug-ins for just about every imaginable task. Examples include style checking, C and C++ compilers, database tools, revision control, web server integration, spelling, and hundreds of others.

The plug-ins I'm interested in showing are Mylyn, Pydev, Pydev Extensions, Subversive, and SQLExplorer. Eclipse is natively a Java development environment. Pydev is a free plug-in that teaches Eclipse to work with Python. Pydev Extensions is a commercial addition to Pydev that does even more, and I find it well worth the small price. Subversive allows Eclipse to work with the Subversion source code repository, making simple revision control tasks almost transparent. Finally, SQLExplorer lets you browse and query databases and their schemas. I'll come back to Subversive in Chapter 3 and SQLExplorer in Chapter 9, but for now we're going to look at Pydev.

From all the glowing statements I've made, you might get the impression that Eclipse is the only game in town. It's not. There are a number of other good choices out there. Most are not free, but they are comparatively low cost. The ones worth noting are Wingware's Wing IDE (`www.wingware.com/`), ActiveState's Komodo (`www.activestate.com/`), and in the Microsoft Windows world, the .NET development environment. I wish I could include JetBrains IntelliJ IDEA in here, but until someone produces a mature Python plug-in for it, we're out of luck.

Eclipse feels a little clunky compared to these others. It has all the features of its competitors, but the interface is a little less polished. Having made my disparaging comments, I'm still going to use Eclipse. There are mind-boggling numbers of plug-ins available, and with the right ones, it does what you'll need it to do.

Installing Eclipse

Having decided on Eclipse, the first step is to get it onto your system. Eclipse lives at `www.eclipse.org/`. The download URL (as of this writing) is `www.eclipse.org/downloads/`. You'll want to get the package called Eclipse Classic. This gives you the kitchen sink.

■ **Warning** The Pydev plug-in can't cope with spaces in the workspace path, so you should ensure that it does not contain any.

Start Eclipse once you've downloaded and installed it. When Eclipse starts up, it will ask you for a workspace location (as shown in Figure 2-1). This is the directory tree in which Eclipse will store all of its data. The workspace isn't a shared resource, so it should be within your home directory. It should be easily backed up, easily remembered, and quickly accessible from the command line. I personally choose the default. (On my Mac, that's `/Users/jeff/Documents/workspace`.)

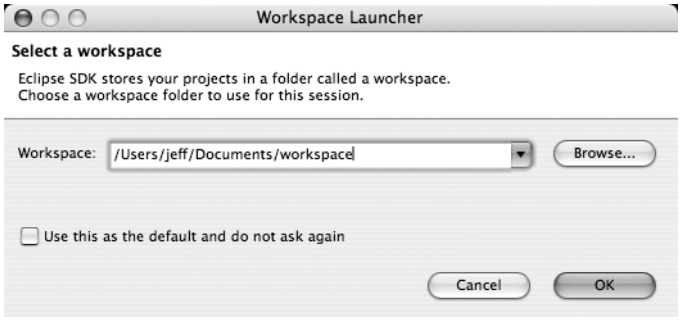

Figure 2-1. *Selecting the workspace root*

Eclipse will grind for a while as it starts up and creates your workspace. It will bring you to the initial landing page, shown in Figure 2-2. This page only shows up until you've created a project, so you won't see it very often.

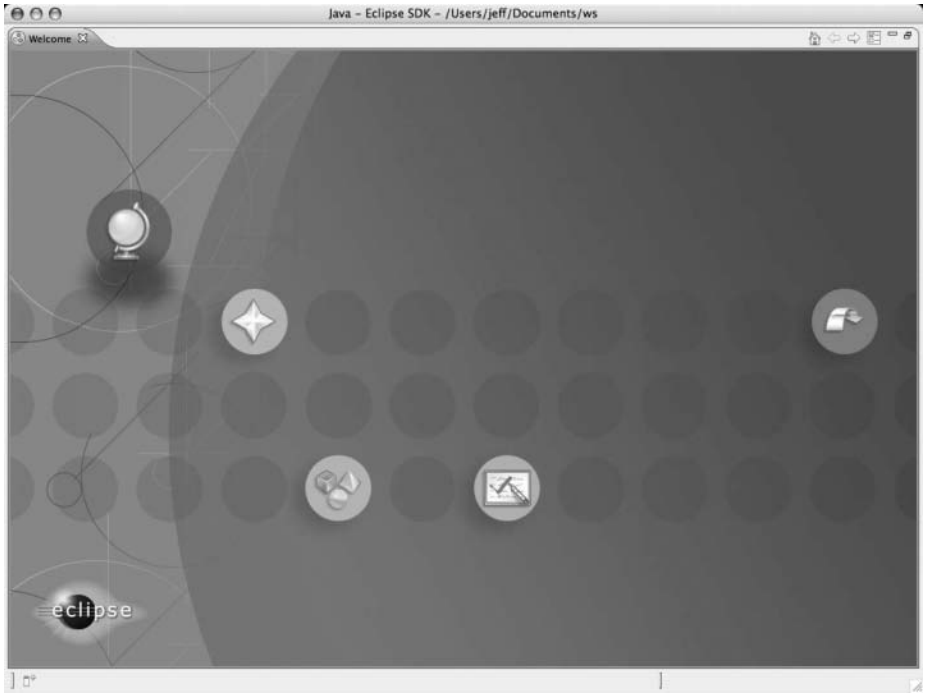

Figure 2-2. *The startup screen, which you probably won't see again*

On the right side is an arrow heading into the distance. This takes you to the workbench. The workbench is where everything happens in Eclipse. It is shown in Figure 2-3.

Now that you can see what Eclipse looks like, it's time for some explanation of the major pieces. Within the workbench are *perspectives*. Perspectives are dedicated to some particular kind of task. Examples include Java development, debugging, source code repository, and plug-in development. The perspective is indicated in the tab at the top right of the window. By default, Eclipse opens into the Java perspective.

The little glyph to the right of the Java indicator allows you to select a different perspective. Once you open a perspective, it stays active, and its icon stays in the tab. By default, the tab is a little small. As you switch between perspectives to perform different tasks, it will quickly fill up, making it harder to switch back to previously used perspectives. You can remedy this by grabbing the tab's left edge and dragging it further to the left, making room for additional icons. This demonstrates a general principle of Eclipse's interface: pretty much everything can be moved or rearranged.

The smaller panes are called *views*. Views do specific functions within a given perspective. Examples are showing console output, viewing outlines, and editing files. You can rearrange the views to your heart's content. You can do this by grabbing the tab and dragging it to another spot within the perspective, or you can drag it onto the desktop, and it will become a free-floating window.

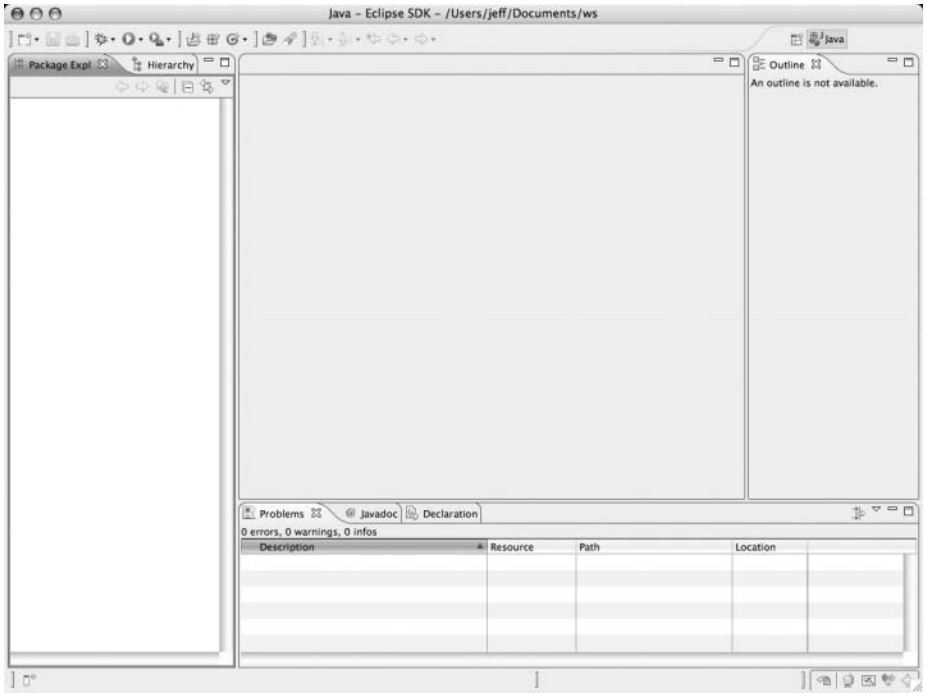

Figure 2-3. *The empty workbench*

We're working in Python, so the Java perspective isn't going to do us much good. We need a plug-in that teaches Eclipse to work with our language. This plug-in is called Pydev. We could hop into installing Pydev right away, but it has some features that depend on another plug-in you're eventually going to want to use. That plug-in is called Mylyn. It is absolutely generic in its installation, so it makes a good example. The process for installing Mylyn and Pydev is much the same.

Installing Plug-Ins

Eclipse plug-ins are published on little web sites known as update sites, and are very easy to install. You give Eclipse the URL for an update site, and it sucks down the plug-in and installs it. Be careful though—don't confuse the web site for a plug-in with its update site. The web site for Mylyn is `www.eclipse.org/mylyn`, while the update site is located at `http://download.eclipse.org/tools/mylyn/update/e3.3`.

Start the installation process by selecting Help ➤ Software Updates ➤ Find and Install. This brings up the Install/Update window, shown in Figure 2-4.

Figure 2-4. *The first screen of the Install/Update wizard*

Select "Search for new features to install," and then click Next. The resulting dialog, shown in Figure 2-5, lets you add a new update site.

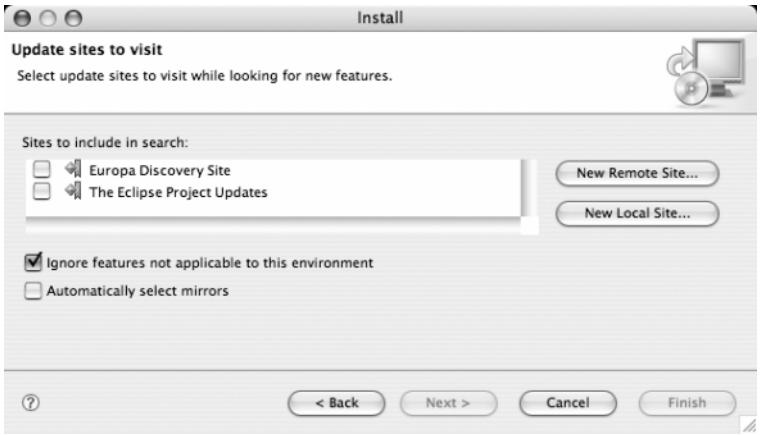

Figure 2-5. *Choosing features to install*

Click the New Remote Site button on the right. This takes you to a dialog with two fields (shown in Figure 2-6).

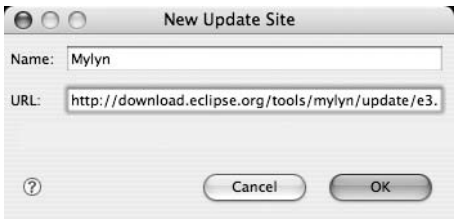

Figure 2-6. *Creating a new feature*

Fill them in as shown in the figure, and click OK. (Note that the URL is bigger than the window and has been cropped a bit.) You'll be taken back to the Install window, as shown in Figure 2-7.

Figure 2-7. *Choosing the newly created Mylyn update site*

The Mylyn site should be highlighted. When you click Finish, Eclipse will download all of the selected updates.

Eclipse will grind for a moment or two while it queries the update site. Assuming that it is successful, it will bring you to the Search Results screen, shown in Figure 2-8.

It's possible to have more than one result, so you have an opportunity to select which plug-ins you want to install (and which parts of which plug-ins if you so desire). Select Mylyn, as shown in the Figure 2-8, and click Next. This takes you to the Feature License screen, shown in Figure 2-9.

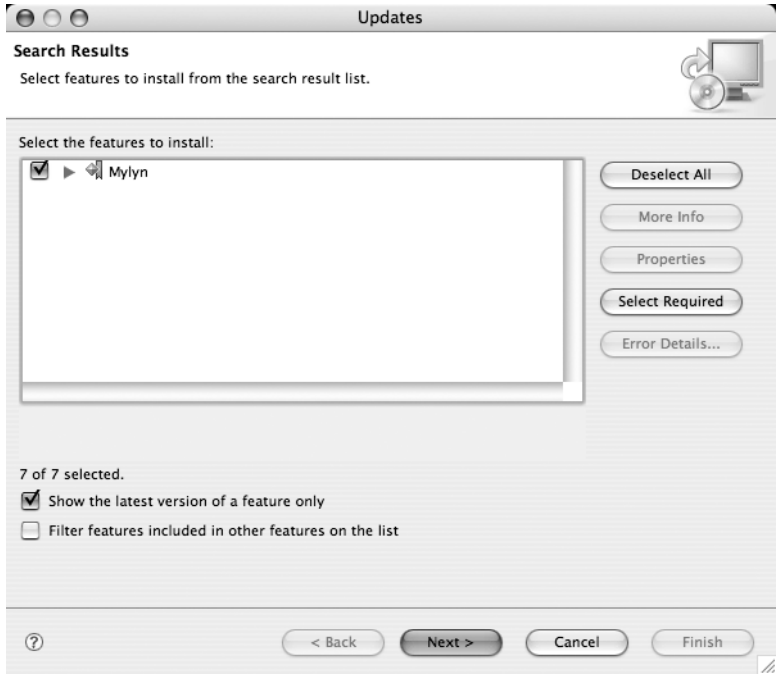

Figure 2-8. *Choosing from the (only) returned update site*

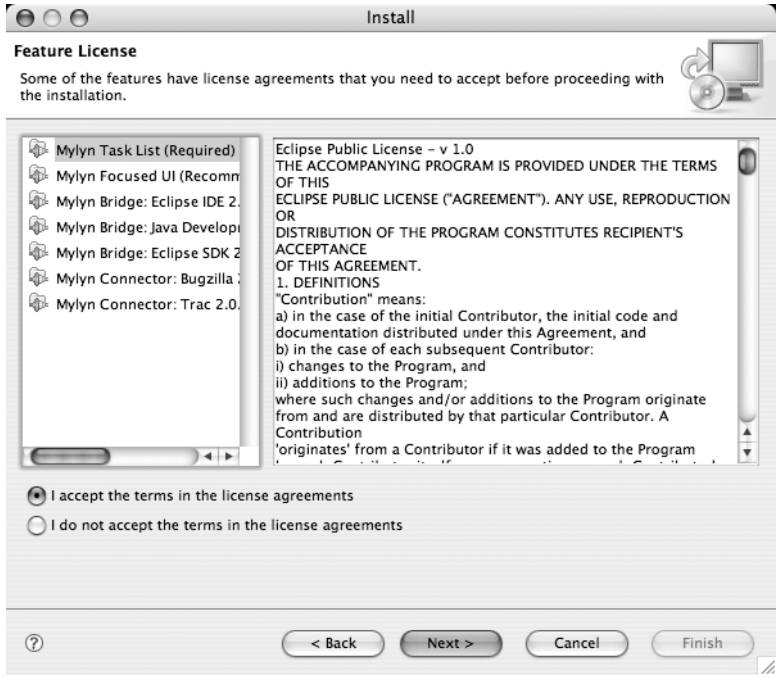

Figure 2-9. *Accepting the license agreements*

Read the license agreements, and if you agree, click "I accept the terms in the license agreements." Then click Next. That will take you (finally) to the Installation screen, shown in Figure 2-10.

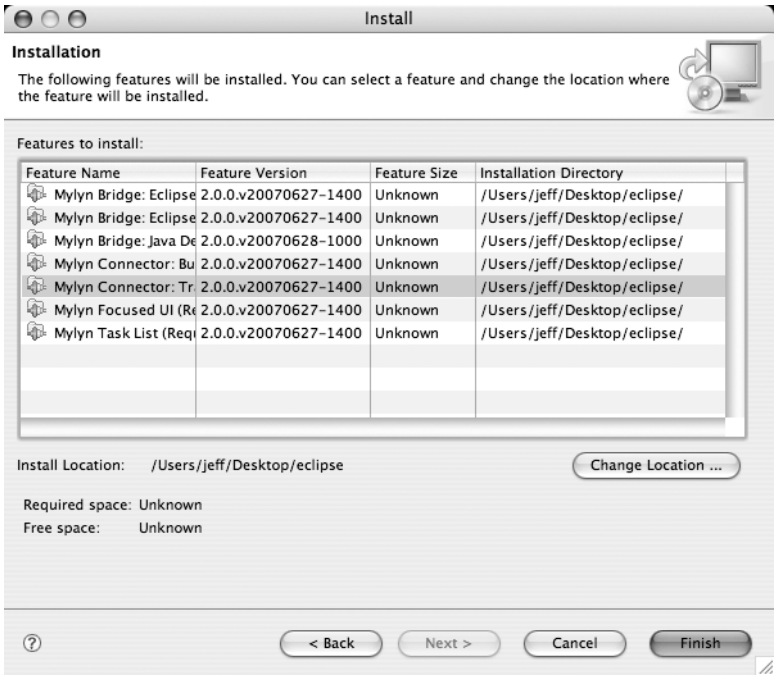

Figure 2-10. *The components will be downloaded.*

The Installation screen gives you a chance to change where the components are located. Don't be tempted. Click Finish, and Mylyn will start downloading and installing. You'll see the dialog shown in Figure 2-11.

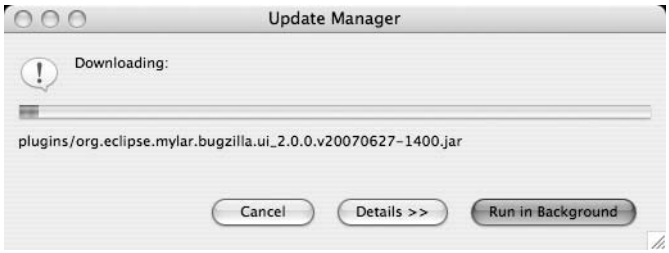

Figure 2-11. *The update downloading*

The installation will progress for a while. At some point, the process will halt, and you will see the window shown in Figure 2-12.

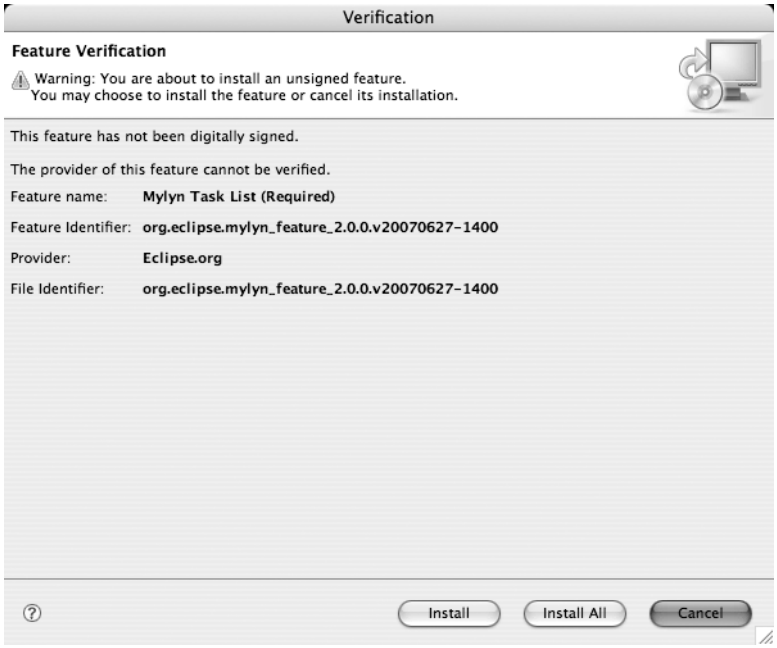

Figure 2-12. *Verifying the unsigned plug-in*

This screen is called Feature Verification, but it's really complaining about the Mylyn package being unsigned. You should get used to this. While cryptographic signing of features is a neat idea, it doesn't happen much. Just click Install All.

A few windows will flit by as Mylyn is installed, and once it's complete, Eclipse will ask if you want to apply the changes or restart (see Figure 2-13).

Figure 2-13. *Installation complete*

Choose Yes to restart Eclipse. When it restarts, it will go to the Eclipse overview, which is shown in Figure 2-14. In the middle of the top bar is the folder over arrow, which will take you back to the workspace. Go there and rejoice in your new accomplishment. Your Eclipse installation has just grown a new capability, even if you haven't used it yet.

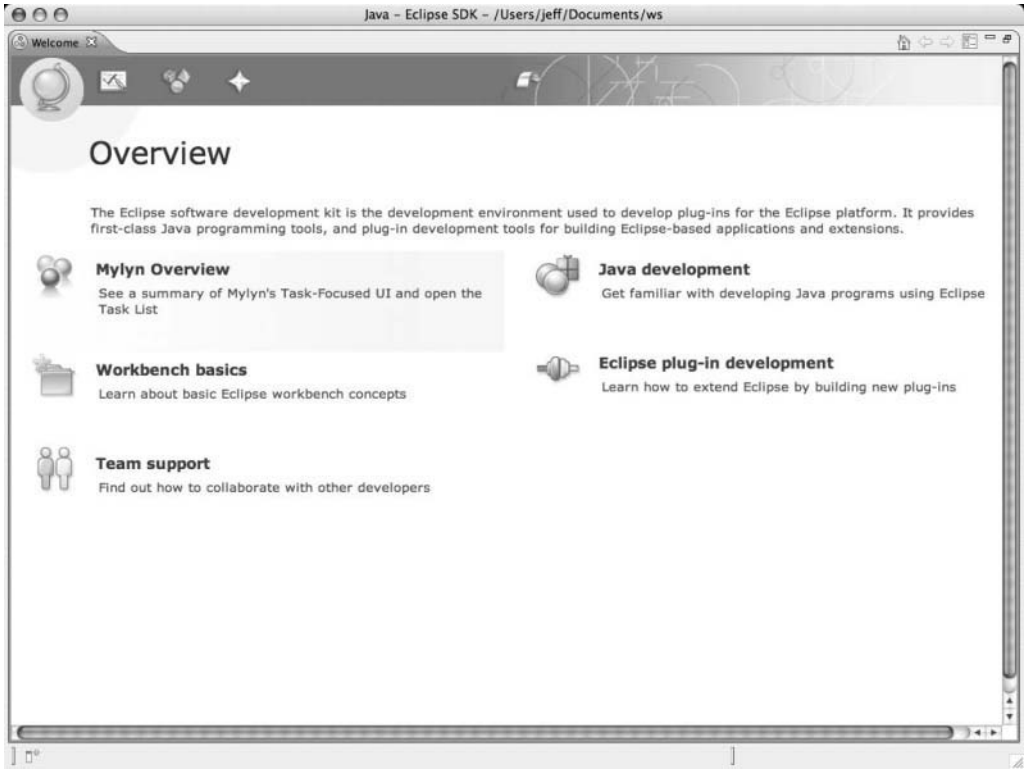

Figure 2-14. *The Overview screen after the installation has completed*

Installing and Configuring Pydev

Mylyn is installed once Eclipse restarts. Installing Pydev is the next step. The Pydev web site is `http://pydev.sourceforge.net/`. The Pydev update site is `http://pydev.sourceforge.net/updates/`. Follow the same procedure as with Mylyn.

Once Pydev is installed, it must be configured. Out of the box, it doesn't know where Python is located, so the first step is configuring the Python interpreter. If you're on Linux, then the Python interpreter will probably be located in `/usr/bin/python`; if you're on OS X, it will probably be located in `/Library/Frameworks/Python.framework/Versions/2.5/bin/python`; and if you're on Windows, it will probably be in `C:\Python25`.

There are three steps. First, open Window ➤ Preferences ➤ Interpreter ➤ Python. Second, enter the path you previously chose. Third, choose the paths to be in your System Python path. You should *not* select folders that will be used in your project, but when starting out, that shouldn't be a problem. By default, it only checks the correct paths, so you'll only need to worry when you start doing more complicated things.

■ **Note** The Python path is a series of directories that are searched when packages are imported. The PYTHONPATH variable contains additional directories that are searched.

When you click OK, it should process through the libraries very quickly, and you should find yourself back at the Python Interpreters screen (shown in Figure 2-15).

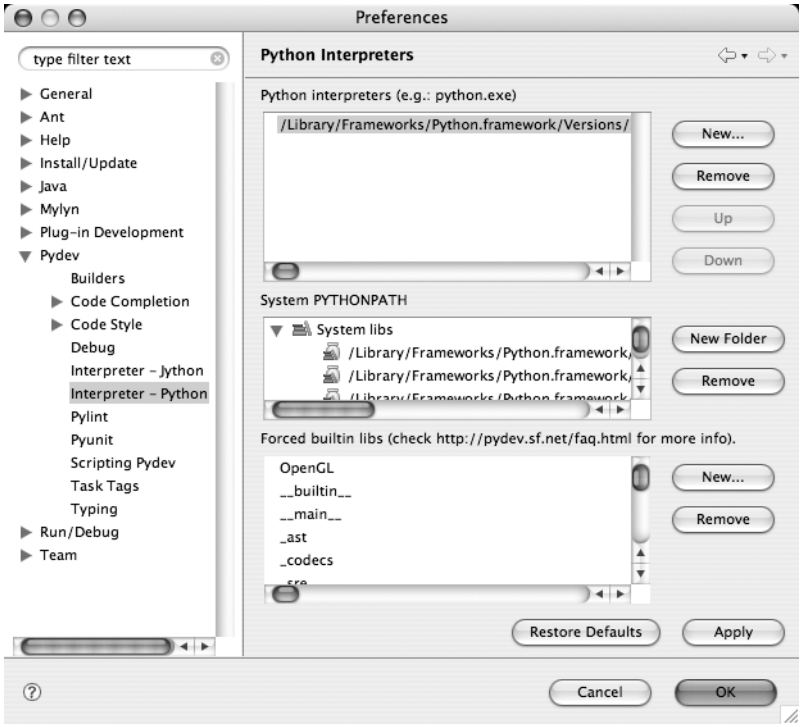

Figure 2-15. *The Python interpreters have been set.*

The system PYTHONPATH box has been filled in with the values from the browser, and so has the "Forced builtin libs" box. You should see "__builtin__" and a slew of other libraries with underscore-prefix names.

When you click OK, you'll see a window entitled Progress Information, and you'll watch a thousand or two module names go flying by as Pydev builds its cache.

Your First Project

At this point, you can start working on a project. Choose File ➤ New ➤ Project. From there, you can select Pydev ➤ Pydev Project. You should see the window shown in Figure 2-16.

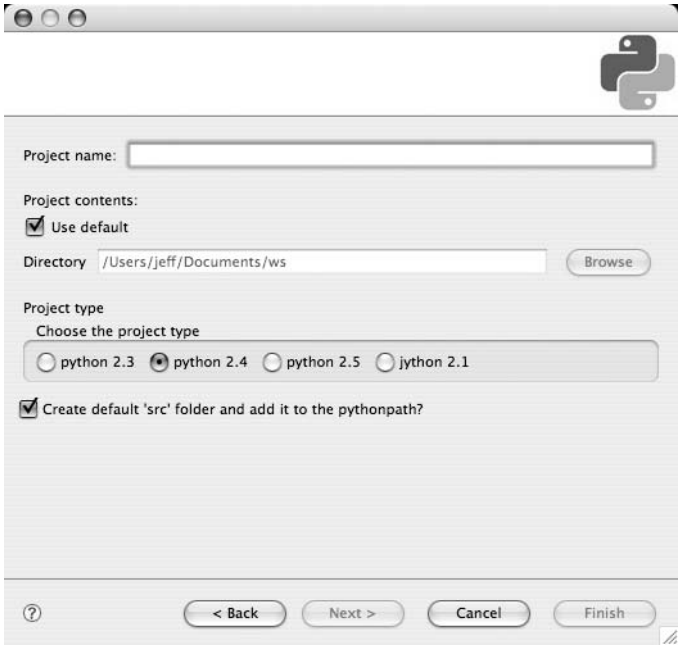

Figure 2-16. *Starting a new project*

Enter **agile** in the "Project name" field. Choose "python 2.5" from the list of project types. (For those of you wondering about the "jython 2.1" option, Jython is a Java-based Python interpreter. I'm not using it in this book.) The final option, labeled "Create default 'src' folder and add it to the pythonpath?" should remain checked.

Source folders are Eclipse directories that contain code. They are automatically added to the Python interpreter's path. To do any development, you need at least one in your project, but it is possible to leave this box unchecked. If you do, then you'll have to add the directories later. You could click Next at this point to reference code in other modules, but we're not doing that in our current project, so click Finish, which will return you to the workbench, as shown in Figure 2-17.

On the left-hand side in a pane entitled Pydev Package Explorer, you'll now see a blue folder entitled `agile`. If you open it, you'll see the `src` folder inside. (You should also note that, up in the right-hand corner, the active icon in the perspective tab shows you are in the Pydev perspective.)

Now you've done the grunt work of setting up a project, and you're at the point where you can start working with Python. You're going to create a Python class called `examples.hello.world.Greet`. Pydev calls library directories *packages*, and it calls source files *modules*. You'll create the package structure, and then create the module.

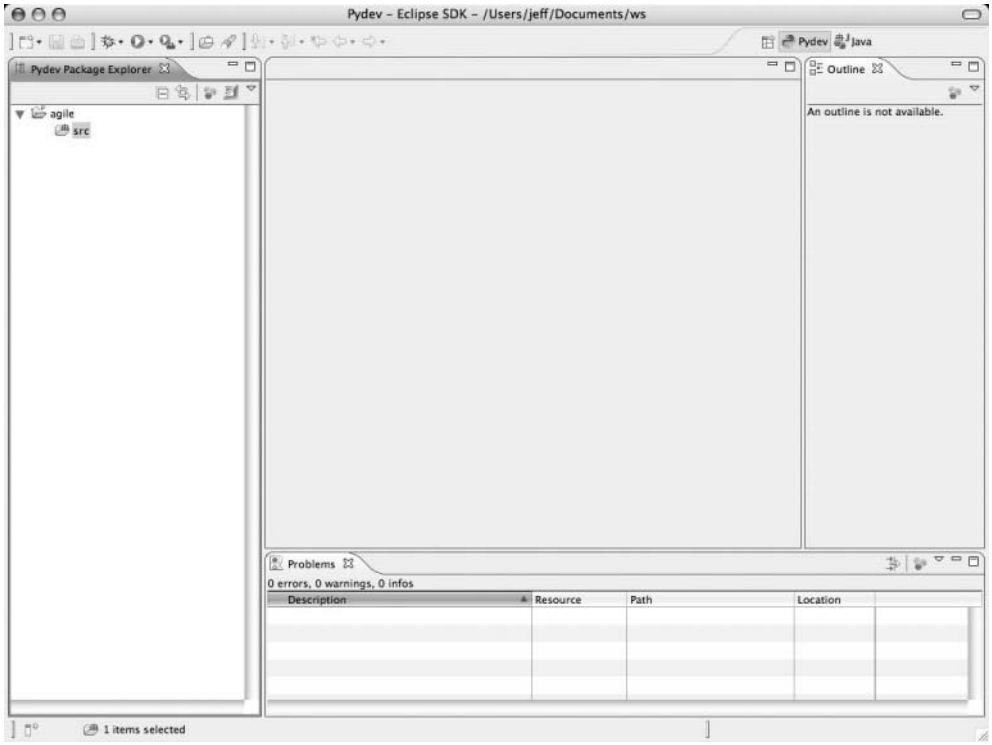

Figure 2-17. *The workbench with your new project*

Right-click the `src` directory and select New ➤ Pydev Package. This brings up a window with two fields, as shown in Figure 2-18.

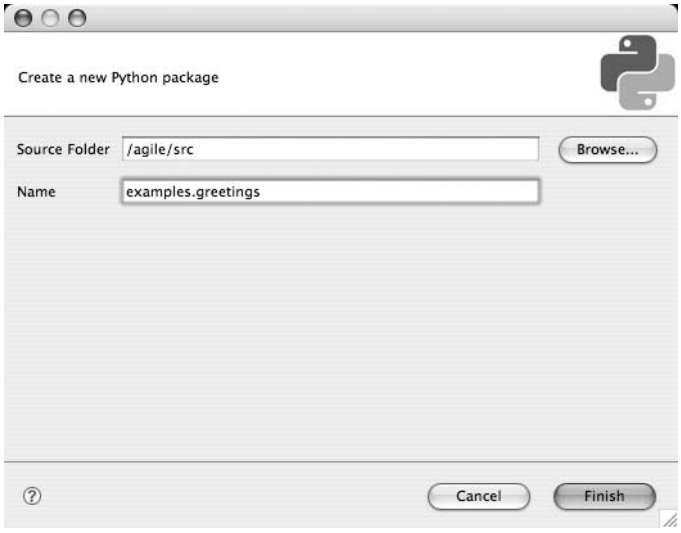

Figure 2-18. *Creating a new project*

The two fields are Source Folder and Name. Source Folder is already filled in with `/agile/src`, and Name is empty; enter the package name `examples.greetings` as shown previously, and then click Finish. This creates the named packages and all of the `__init__.py` files, and takes you back to the workbench, as shown in Figure 2-19.

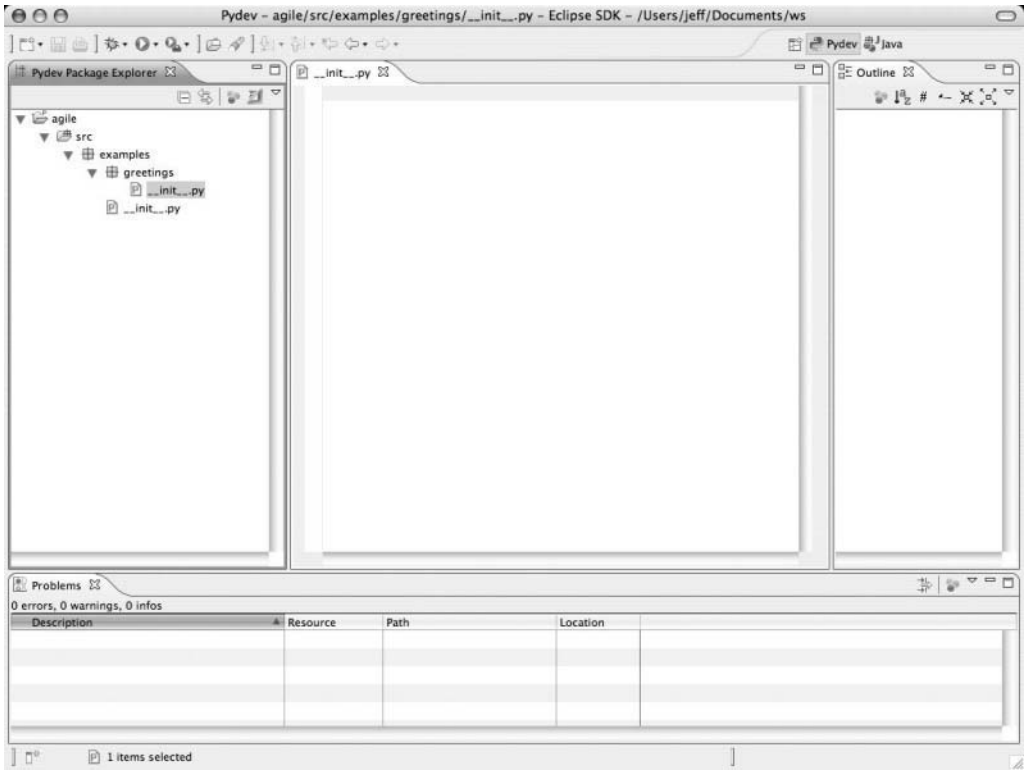

Figure 2-19. *The new packages have been created.*

Now you'll create the module `examples.greetings.standard`. Right-click the `greetings` package in the Package Explorer on the left side of the workspace. Select New ➤ Pydev Module. That will bring up the (unnamed) module creation window, shown in Figure 2-20.

Figure 2-20. *Choosing the module name*

The window has three fields: Source Folder, Package, and Name. The first two are filled in for you. Enter **standard** into the Name field, and then click Finish. You will be taken back to the workbench, which should look something like Figure 2-21.

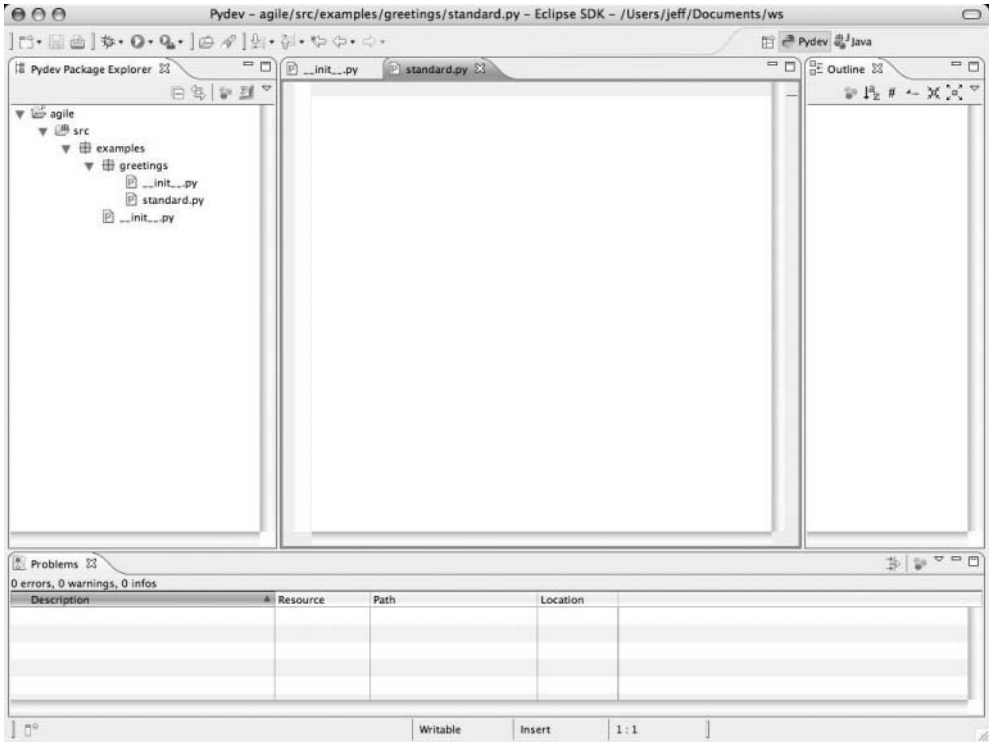

Figure 2-21. *The module has been created.*

You'll notice that `standard.py` shows up in the Package Explorer on the left, and that the editor in the center pane is open to this file. Click into that window and enter the following program:

```
#!/usr/bin/python

class HelloWorld(object):

    def main(self):
        print "Hello World!"

if __name__ == '__main__':
    HelloWorld().main()
```

You'll notice several things while you type, the most interesting of which is that Pydev makes guesses about what you're about to type. In the case of the `def main`, it makes the correct guess, but it doesn't move the cursor. You can either continue typing, or you can accept Pydev's guess by pressing Ctrl+Enter. After you enter and save this text, the workspace will look something like Figure 2-22.

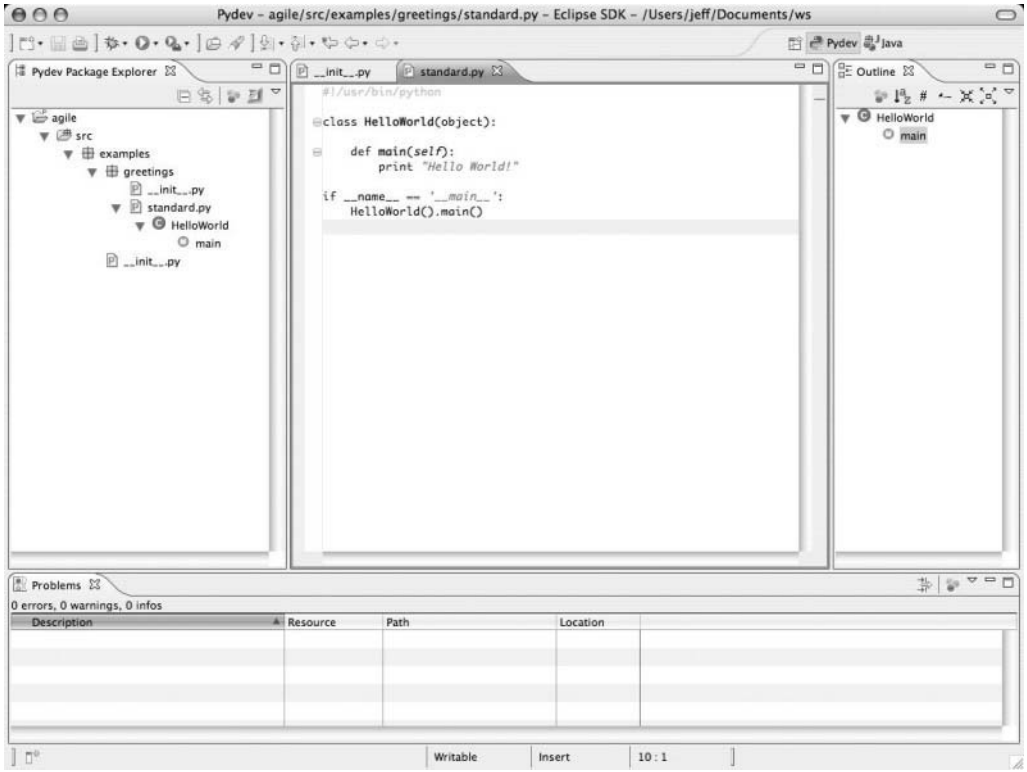

Figure 2-22. *The HelloWorld class has been created.*

As you typed, Pydev also updated the Package Explorer on the left and the Outline view on the right. You can use both to navigate through your program. Double-clicking any node in the Package Explorer opens an editor for the corresponding file. The cursor is positioned at the point in the program that the node in the explorer represents. The Outline view to the right shows the elements corresponding to the currently active editor. Clicking any element in the Outline view will also take you to the corresponding definition.

Running the program is easy. Right-click inside the text editor, or from the context menu select Run As ➤ 4 Python Run. The program's output will appear in a Console view at the bottom of the workspace. You can also bring up the same menu by right-clicking the `standard.py` module in the Package Explorer window.

Looking Under the Hood

Many external tools interact with the code stored in Eclipse. In order to work with them, it is necessary to understand how Eclipse lays out its directories. To begin, you'll need to change directories to the root of your Eclipse workspace; on my machine, it is `~/Documents/workspace`. From there, we'll go on a brief tour. (My machine's name is phytoplankton and my username is jeff.)

```
phytoplankton:~ jeff$ cd ~/Documents/workspace
phytoplankton:~/Documents/workspace jeff$ ls -aF
```

```
. ./            ../.            .metadata       agile/
```

The directory `.metadata` contains all of your Eclipse preferences and system-wide configuration data. We really don't care much about the details of this directory. The directory `agile` is of more interest. It contains the project that we are working with, so we'll take a look inside there.

```
phytoplankton:~/Documents/workspace jeff$ cd agile
phytoplankton:~/Documents/workspace/agile jeff$ ls -aF
```

```
./              ../             .project        .pydevproject   src/
```

Besides the ubiquitous `.` and `..`, there are three filesystem entries here: two files and one directory. The file `.project` contains the Eclipse configuration information for the project. The file `.pydevproject` contains the project configuration information for Pydev. Both are XML files, and they are specific to the project.

■ **Note** Under the UNIX filesystem, every directory has two subdirectories. The directory named `.` (one period) is the current directory. The directory named `..` is the parent directory. These are shown, but they are ignored during this discussion.

The directory src is the source folder you defined when the project was created. It contains the Python packages and modules that you defined. They're just normal Python library directories with __init__.py files and .py files.

```
phytoplankton:~/Documents/workspace/agile jeff$ cd src
phytoplankton:~/Documents/workspace/agile/src jeff$ ls -aF
```

```
./              ../             examples/
```

```
phytoplankton:~/Documents/workspace/agile/src jeff$ ls -aF examples
```

```
./              ../             __init__.py     greetings/
```

```
phytoplankton:~/Documents/workspace/agile/src jeff$ ls -aF examples/greetings
```

```
./              ../             __init__.py     standard.py
```

These are just normal Python files organized just like you'd expect. If you set the PYTHONPATH to the root of the source directory (~/Documents/workspace/agile/src) you can develop from the command line using the same files. In fact you're going to be doing that often in the coming chapters.

Paying for More Functionality

At this point, I'd suggest trying out Pydev Extensions. If you end up using Eclipse for Python development (as I hope you do), then it is a worthwhile investment. It gives you a number of features missing from the free version of Pydev, most of them relating to much-improved code analysis. Notably, it includes a much better "go to definition of function" feature for navigating your code.

Pydev uses an external program (Bicycle Repair Man) that has trouble with class methods, but Pydev Extensions improves upon this. It does real-time code analysis, and its results are superior to those of vanilla Pydev. It offers auto-management of imports and much better code completion. It offers a wider variety of refactorings and the ability to do remote debugging. I could go on, but I'll let you explore on your own.

The extensions have a 30-day free trial. They continue working after the trial is over, but they nag you every couple of hours, suggesting that you should buy a copy. The nagging isn't a hindrance to your work, but it is sufficiently annoying that you'll probably give in and buy the software just to make the messages go away. The web site for Pydev Extensions is http://fabioz.com/pydev/. The download site is www.fabioz.com/pydev/updates/. The extensions install from the update site, just as with Mylyn and Pydev.

Summary

Eclipse is an increasingly popular IDE that offers many advantages over the command line. It's free, and it has an open plug-in architecture that is being exploited by many component producers. There are plug-ins available for a plethora of purposes, many of which are relevant to this book.

Plug-ins are easily loaded from within Eclipse itself. You've done this with several so far. Mylyn is the most detailed example of the procedure, but more significantly we've installed the Pydev plug-in and configured it.

Pydev turns Eclipse into a Python development environment. It acts as a sophisticated wrapper around the system Python installations. It includes everything users have come to expect from an IDE, providing structural browsers, integrated editors, and code completion. It has a number of features that haven't been examined, as well. These include unit testing support, an integrated graphical debugger, remote debugging, and refactoring support.

There is a slew of additional features that are added by Pydev Extensions. It's free for 30 days, so it's worth looking at. If you decide the features aren't worth the money, then you can turn Pydev Extensions off again through the plug-in management screens.

Now that I've introduced a basic working environment, I'll examine revision control and source repositories. Revision control is the technology that underlies the agile practice of continuous integration. The source repository is where file revisions are stored, and it is the location where all the group's development work resides. It is the means by which members' changes are integrated on a daily basis, and it serves as the source of all truth for the code base. Users get the most recent code from here, and so does all of the build automation.

Subversion is the revision control system examined in this book. It's free, and it's in common use. It's a favorite with developers, and it is far superior to its predecessor, CVS. It can be run locally or as a remote service. In the next chapter you'll learn how to use it both from the command line and from within Eclipse using the Subversive plug-in.