
jython Documentation

Release latest

Sep 07, 2017

Contents

1	Python for the Java Platform	1
2	Part I: Jython Basics: Learning the Language	3
2.1	Chapter 2: Data Types and Referencing	3
2.2	Chapter 3: Operators, Expressions, and Program Flow	21
2.3	Chapter 4: Defining Functions and Using Built-Ins	29
2.4	Chapter 9: Input and Output	46
2.5	Chapter 5: Object Oriented Jython	52
2.6	Chapter 6: Exception Handling and Debugging	64
2.7	Chapter 7: Modules and Packages	73
3	Part II: Using the Language	85
3.1	Chapter 8: Scripting With Jython	85
3.2	Chapter 10: Jython and Java Integration	89
3.3	Chapter 11: Using Jython in an IDE	106
3.4	Chapter 12- Databases and Jython: Object Relational Mapping and Using JDBC	131
4	Part III: Developing Applications with Jython	157
4.1	Chapter 13: Simple Web Applications	157
4.2	Chapter 14: Web Applications with Django	171
4.3	Chapter 15: Introduction to Pylons	208
4.4	Chapter 16: GUI Applications	223
4.5	Chapter 17: Deployment Targets	232
5	Part IV: Strategy and Technique	249
5.1	Chapter 18: Testing and Continuous Integration	249
5.2	Chapter 19: Concurrency	274
6	Part V: Appendicies and Attribution	289
6.1	Appendix A: Using Other Tools with Jython	289
6.2	Appendix B: Jython Cookbook - A compilation of community submitted code examples	294
6.3	Attribution	310
7	Indices and tables	313

Python for the Java Platform

Authors Josh Juneau, Jim Baker, Victor Ng, Leo Soto, Frank Wierzbicki

Version 0.9 of 10/18/2009

This book is presented in open source and licensed through Creative Commons 3.0. You are free to copy, distribute, transmit, and/or adapt the work. This license is based upon the following conditions:

Attribution:

You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike:

If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Any of the above conditions can be waived if you get permission from the copyright holder.

In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights
- The author's moral rights
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights

Notice: For any reuse or distribution, you must make clear to the others the license terms of this work. The best way to do this is with a direct link to this page: <http://creativecommons.org/licenses/by-sa/3.0/>

To be printed by Apress Fall 2009 – <http://www.apress.com/book/view/9781430225270>

ISBN10: 1-4302-2527-0

ISBN13: 978-1-4302-2527-0

Source code will be available at: <http://www.apress.com>

Part I: Jython Basics: Learning the Language

Chapter 2: Data Types and Referencing

We all know that programming languages and applications need data. We define applications to work with data, and we need to have containers that can be used to hold it. This chapter is all about defining containers and using them to work with application data. This is the foundation of any programming language...it is how we get tasks accomplished. Whether the data we are using is coming from a keyboard entry or if we are working with a database, there needs to be a way to temporarily store it in our programs so that it can be manipulated and used. Once we're done working with the data then these temporary containers can be destroyed in order to make room for new constructs.

We'll start by taking a look at the different data types that are offered by the Python language, and then we'll follow by discussing how to use that data once it has been collected and stored. We will compare and contrast the different types of structures that we have in our arsenal, and I'll give some examples of which structures to use for working with different types of data. There are a multitude of tasks that can be accomplished through the use of lists, dictionaries, and tuples and I will cover the majority of them. Once you learn how to define and use these structures, then we'll talk a bit about what happens to them once they are no longer needed by our application.

Lets begin our journey into exploring data types and structures within the Python programming language...these are skills that you will use in each and every practical Jython program.

Python Data Types

As we've discussed, there is a need to store and manipulate data within programs. In order to do so then we must also have the ability to create containers used to hold that data so that the program can use it. The language needs to know how to handle data once it is stored, and we can do that by assigning data type to our containers. However, in Python it is not a requirement to do so because the interpreter is able to determine which type of data we are storing in a dynamic fashion.

The following table lists each data type and gives a brief description of the characteristics that define each of them.

Data Type	Characteristics
None	NULL value object
Numeric	A data type used to hold numeric values of integer, decimal, float, complex, and long
Boolean	True or False value (also characterized as numeric values of 1 and 0 respectively)
Sequence	Includes the following types: string, unicode string, basestring, xrange, list, tuple
Mapping	Includes the dictionary type
Set	Unordered collection of distinct objects; includes the following types: set, frozenset
File	Used to make use of file system objects
Iterator	Allows for iteration over a container

Table 2-1. Python Data Types

Given all of that information and the example above, we need to know a way to declare a variable in the Python language. You’ve seen some examples in the previous chapter, but here I will formally show how it is done. Let’s take a look at some examples of defining variables in the following lines of code.

```
# Defining a String
x = 'Hello World'
x = "Hello World Two"

# Defining a number
y = 10

# Float
z = 8.75

# Complex
i = 8.07j
```

An important point to note is that there really are no types in Jython. Every object is an instance of a class. Therefore, in order to find the type of an object in Jython it is perfectly valid to write `obj.__class__`.

```
# Return the type of an object in Jython using __class__
>>> a = 'Hello'
>>> a.__class__
<type 'str'>
```

Strings and String Methods

Strings are a special type within most programming languages because they are often used to manipulate data. A string in Python is a sequence of characters, which is immutable. This is very important to know as it has a large impact on the overall understanding of strings. Once a string has been defined it cannot be changed. However, there are a large amount of string methods that can be used to manipulate the contents of a particular string. Although we can manipulate the contents, Python really gives us a manipulated copy of the string... the original string is left unchanged.

Prior to the 2.5.0 release of Jython, CPython and Jython treated strings a bit differently. There are two types of string objects in CPython, these are known as *Standard* strings and *Unicode* strings. Standard strings contain 8-bit data, whereas Unicode strings are sequences of data composed of 16-bit characters. There is a lot of documentation available that specifically focuses on the differences between the two types of strings, this reference will only cover the basics. It is worth noting that Python contains an abstract string type known as *basestring* so that it is possible to check any type of string to ensure that it is a string instance.

Prior to the 2.5.0 release of Jython, there was only one string type. The string type in Jython supported full two-byte Unicode characters and all functions contained in the string module are Unicode-aware. If the `u` string modifier was specified, it was ignored by Jython. Since the release of 2.5.0, strings in Jython are treated just like those in CPython, so the same rules will apply to both implementations. It is also worth noting that Jython uses character properties from

the Java platform. Therefore properties such as `isupper` and `islower`, which we will discuss later in the section, are based upon the Java properties.

In remainder of this section we will go through each of the many string functions that are at our disposal. These functions will work on both Standard and Unicode strings. As with many of the other features in Python and other programming languages, there are often times more than one way to accomplish a task. In the case of strings and string manipulation, this definitely holds true. However, you will find that in most cases, although there are more than one way to do things, Python experts have added functions which allow us to achieve better performing and easier to read code. Sometimes one way to perform a task is better achieved by utilizing a certain function in one case, and doing something different in another case.

The following table lists all of the string methods that have been built into the Python language as of the 2.5 release. Since Python is an evolving language, this list is sure to change in future releases. Most often, additions to the language will be made, or existing features are enhanced. Following the table, I will give numerous examples of the methods and how they are used. Although I cannot provide an example of how each of these methods work (that would be a book in itself), they all function in the same manner so it should be rather easy to pick up.

Method	Description of Functionality
<code>capitalize()</code>	Capitalize string
<code>center(width[,fill])</code>	Reposition string and provide optional padding filler character
<code>count(sub[,start[,end]])</code>	Count the number of times the substring occurs within the string
<code>decode([encoding[,errors]])</code>	Decodes and returns Unicode string
<code>encode([encoding[,errors]])</code>	Produces an encoded version of a string
<code>endswith(suffix[,start[,end]])</code>	Returns a boolean to state whether the string ends in a given pattern
<code>expandtabs([tabsize])</code>	Converts tabs within a string into spaces
<code>find(sub[,start[,end]])</code>	Returns the index of the position where the first occurrence of the given substring begins
<code>index(sub[,start[,end]])</code>	Returns the index of the position where the first occurrence of the given substring begins
<code>isalnum()</code>	Returns a boolean to state whether the string contain both alphabetic and numeric characters
<code>isalpha()</code>	Returns a boolean to state whether the string contains all alphabetic characters
<code>isdigit()</code>	Returns a boolean to state whether the string contains all numeric characters
<code>islower()</code>	Returns a boolean to state whether a string contains all lowercase characters
<code>isspace()</code>	Returns a boolean to state whether the string consists of all whitespace
<code>istitle()</code>	Returns a boolean to state whether the first character of each word in the string is capitalized
<code>isupper()</code>	Returns a boolean to state whether all characters within the string are uppercase
<code>join(sequence)</code>	Joins two strings by combining
<code>ljust(width[,fillchar])</code>	Align the string to the left by width
<code>lower()</code>	Converts all characters in the string to lowercase
<code>lstrip([chars])</code>	Removes the first found characters in the string from the left that match the given characters. Also removes v
<code>partition(separator)</code>	Partitions a string starting from the left using the provided separator
<code>replace(old,new[,count])</code>	Replaces the portion of string given in <i>old</i> with the portion given in <i>new</i>
<code>rfind(sub[,start[,end]])</code>	Searches and finds the first occurrence from the end of the given string
<code>rindex(sub[,start[,end]])</code>	Searches and finds the first occurrence of the given string or returns an error
<code>rjust(width[,fillchar])</code>	Align the string to the right by width
<code>rpartition(separator)</code>	Partitions a string starting from the right using the provided separator object
<code>rsplit([separator[,maxsplit]])</code>	Splits the string from the right side and uses the given separator as a delimiter
<code>rstrip([chars])</code>	Removes the first found characters in the string from the right that match those given. Also removes v
<code>split([separator[,maxsplit]])</code>	Splits the string and uses the given separator as a delimiter.
<code>splitlines([keepends])</code>	Splits the string into a list of lines. Keepends denotes if newline delimiters are removed.
<code>startswith(prefix[,start[,end]])</code>	Returns a boolean to state whether the string starts with the given prefix
<code>strip([chars])</code>	Removes the given characters from the string.
<code>swapcase()</code>	Converts the case of each character in the string.
<code>title()</code>	Returns the string with the first character in each word uppercase.
<code>translate(table[,deletechars])</code>	Use the given character translation table to translate the string.

Table 2.1 – continued from previous page

Method	Description of Functionality
<code>upper()</code>	Converts all of the characters in the string to lowercase.
<code>zfill(width)</code>	Pads the string from the left with zeros for the specified width.

Table 2-2. String Methods

Now let's take a look at some examples so that you get an idea of how to use the string methods. As stated previously, most of them work in a similar manner.

```
ourString='python is the best language ever'

# Capitalize a String
>>> ourString.capitalize()
'Python is the best language ever'

# Center string
>>> ourString.center(50)
'          python is the best language ever          '
>>> ourString.center(50,'-')
'-----python is the best language ever-----'

# Count substring within a string
>>> ourString.count('a')
2

# Partition a string
>>> x = "Hello, my name is Josh"
>>> x.partition('\n')
('Hello, my ', '\n', 'ame is Josh')
```

String Formatting

You have many options when printing strings using the *print* statement. Much like the C programming language, Python string formatting allows you to make use of a number of different conversion types when printing.

```
Using String Formatting
# The two syntaxes below work the same
>>> x = "Josh"
>>> print "My name is %s" % (x)
My name is Josh
>>> print "My name is %s" % x
My name is Josh
```

Type	Description
d	signed integer decimal
i	signed integer decimal
o	unsigned octal
u	unsigned decimal
x	unsigned hexadecimal
X	unsigned hexadecimal (upper)
E	floating point exponential format (upper)
e	floating point exponential format
f	floating point decimal format
F	floating point decimal format (upper)
g	floating point exponential format if exponent > -4, otherwise float
G	floating point exponential format (uppr) if exponent > -4, otherwise float
c	single character
r	string (converts any python object using repr())
s	string (converts any python object using str())
%	no conversion, results in a percent (%) character

Table 2-3. Conversion Types

```
>>> x = 10
>>> y = 5.75
>>> print 'The expression %d * %f results in %f' % (x, y, x*y)
The expression 10 * 5.750000 results in 57.500000
```

Ranges

Ranges are not really a data type or a container; they are really a Jython built-in function (Chapter 4). For this reason, we will only briefly touch upon the range function here, and they'll be covered in more detail in Chapter 4. However, because they play such an important role in the iteration of data, usually via the *for* loop, I think it is important to discuss them in this section of the book. The range is a special function that allows one to iterate between a range of numbers; and/or list a specific range of numbers. It is especially helpful for performing mathematical iterations, but it can also be used for simple iterations.

The format for using the range function includes an optional starting number, an ending number, and an optional stepping number. If specified, the starting number tells the range where to begin, whereas the ending number specifies where the range should end. The optional step number tells the range how many numbers should be placed between each number contained within the range output.

Range Format

`range([start], stop, [step])`

```
>>> range(0, 10)

>>> range(10)

>>> range(0, 10, 2)
>>> range(100, 0, -10)
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
```

As stated previously, this function can be quite useful when used within a *for* loop as the Jython *for* loop syntax works very well with it. The following example displays a couple examples of using the range function within a *for* loop context.

```
>>> for i in range(10):
...     print i
...
0
1
2
3
4
5
6
7
8
9

# Multiplication Example
>>> x = 1
>>> for i in range(2, 10, 2):
...     x = x + (i * x)
...     print x
...
3
15
105
945
```

As you can see, a range can be used to iterate through just about any number set...be it positive or negative in range.

Lists, Dictionaries, Sets, and Tuples

Data collection containers are a useful tool for holding and passing data throughout the lifetime of an application. The data can come from any number of places, be it the keyboard, a file on the system, or a database...it can be stored in a collection container and used at a later time. Lists, dictionaries, sets, and tuples all offer similar functionality and usability, but they each have their own niche in the language. We'll go through several examples of each since they all play an important role under certain circumstances.

Since these containers are so important, we'll go through an exercise at the end of this chapter, which will give you a chance to try them out for yourself.

Lists

Perhaps one of the most used constructs within the Python programming language is the list. Most other programming languages provide similar containers for storing and manipulating data within an application. The Python list provides an advantage to those similar constructs which are available in statically typed languages. The dynamic tendencies of the Python language help the list construct to harness the great feature of having the ability to contain values of different types. This means that a list can be used to store any Python data type, and these types can be mixed within a single list. In other languages, this type of construct is defined as a typed object, which locks the construct to using only one data type.

The creation and usage of Jython lists is just the same as the rest of the language...very simple and easy to use. Simply assigning a set of empty square brackets to a variable creates an empty list. We can also use the built-in list() type to create a list. The list can be constructed and modified as the application runs, they are not declared with a static length. They are easy to traverse through the usage of loops, and indexes can also be used for positional placement or removal of particular items in the list. We'll start out by showing some examples of defining lists, and then go through each of the different avenues which the Jython language provides us for working with lists.

```
# Define an empty list
myList = []
myList = list()

# Define a list of string values
myStringList = ['Hello', 'Jython', 'Lists']

# Define a list containing multiple data types
multiList = [1, 2, 'three', 4, 'five', 'six']

# Define a list containing a list
comboList = [1, myStringList, multiList]
```

As stated previously, in order to obtain the values from a list we can make use of indexes. Much like the Array in the Java language, using the `list[index]` notation will allow us to access an item. If we wish to obtain a range or set of values from a list, we can provide a *starting* index, and/or an *ending* index. This technique is also known as *slicing*. What's more, we can also return a set of values from the list along with a stepping pattern by providing a *step* index as well. One key to remember is that while accessing a list via indexing, the first element in the list is contained within the 0 index.

```
# Obtain elements in the list
>>> myStringList[0]
'Hello'

>>> myStringList[2]
'Lists'

>>> myStringList[-1]
'Lists'

# Using the slice method
>>> myStringList[0:2]
['Hello', 'Jython']

# Return every other element in a list
>>> newList=[2,4,6,8,10,12,14,16,18,20]
>>> newList[0:10:2]
[2, 6, 10, 14, 18]

# Leaving a positional index blank will also work
>>> newList[::2]
[2, 6, 10, 14, 18]
```

Modifying a list is much the same, you can either use the index in order to insert or remove items from a particular position. There are also many other ways that you can insert or remove elements from the list. Jython provides each of these different options as they provide different functionality for your operations.

In order to add an item to a list, you can make use of the `append()` method in order to add an item to the end of a list. The `extend()` method allows you to add an entire list or sequence to the end of a list. Lastly, the `insert()` method allows you to place an item or list into a particular area of an existing list by utilizing positional indexes. You will examples of each method below.

Similarly, we have plenty of options for removing items from a list. The `del` statement, as explained in Chapter 1, can be used to remove or delete an entire list or values from a list using the index notation. You can also use the `pop()` *or* `remove()` method to remove single values from a list. The `pop()` method will remove a single value from the end of the list, and it will also return that value at the same time. If an index is provided to the `pop()` function, then it will remove and return the value at that index. The `remove()` method can be used to find and remove a particular value in

the list. If more than one value in the list matches the value passed into the *remove()* function, the first one will be removed. Another note about the *remove()* function is that the value removed is not returned. Let's take a look at these examples of modifying a list.

```
# Adding values to a list
>>> newList=['a','b','c','d','e','f','g']
>>> newList.append('h')
>>> print newList
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

# Add another list to the existing list
>>> newList2=['h','i','j','k','l','m','n','o','p']
>>> newList.extend(newList2)
>>> print newList
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p']

# Insert a value into a particular location via the index
>>> newList.insert(2,'c')
>>> print newList
['a', 'b', 'c', 'c', 'd', 'e', 'f', 'g', 'h', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
↪ 'p']

# Use the slice notation to insert another list or sequence
>>> newListA=[100,200,300,400]
>>> newListB=[500,600,700,800]
>>> newListA[0:2]=newListB
>>> print newListA
[500, 600, 700, 800, 300, 400]

# Use the del statement to delete a list
>>> newList3=[1,2,3,4,5]
>>> print newList3
[1, 2, 3, 4, 5]
>>> del newList3
>>> print newList3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'newList3' is not defined

# Use the del statement to remove a value or range of values from a list
>>> newList3=['a','b','c','d','e','f']
>>> del newList3[2]
>>> newList3
['a', 'b', 'd', 'e', 'f']
>>> del newList3[1:3]
>>> newList3
['a', 'e', 'f']

# Remove values from a list using pop and remove functions
>>> print newList
['a', 'b', 'c', 'c', 'd', 'e', 'f', 'g', 'h', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
↪ 'p']
>>> newList.pop(2)
'c'
>>> print newList
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p']
>>> newList.remove('h')
>>> print newList
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p']

# Useful example of using pop() function
>>> x = 5
>>> timesList = [1,2,3,4,5]
>>> while timesList:
...     print x * timesList.pop(0)
...
5
10
15
20
25
```

Now that we know how to add and remove items from a list, it is time to learn how to manipulate the data within them. Python provides a number of different methods that can be used to help us manage our lists. See the table below for a list of these functions and what they can do.

Method	Tasks Performed
index	Returns the index of the first value in the list which matches a given value.
count	Returns the number of items in the list which match a given value.
sort	Sorts the items contained within the list.
reverse	Reverses the order of the items contained within the list

Table 2-4. Python List Methods

Let's take a look at some examples of how these functions can be used on lists.

```
# Returning the index for any given value
>>> newList=[1,2,3,4,5,6,7,8,9,10]
>>> newList.index(4)
3

# Add a duplicate into the list and then return the index
>>> newList.append(6)
>>> newList
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 6]
>>> newList.index(6)
5

# Using count() function to return the number of items which match a given value
>>> newList.count(2)
1
>>> newList.count(6)
2

# Sort the values in the list
>>> newList.sort()
>>> newList
[1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10]

# Reverse the order of the value in the list
>>> newList.reverse()
>>> newList
[10, 9, 8, 7, 6, 6, 5, 4, 3, 2, 1]
```

Lists

Moving around within a list is quite simple. Once a list is populated, often times we wish to traverse through it and perform some action against each element contained within it. You can use any of the Python looping constructs to traverse through each element within a list. While there are plenty of options available, the *for* loop works especially well. The reason is because of the simple syntax that the Python *for* loop uses. This section will show you how to traverse a list using each of the different Python looping constructs. You will see that each of them has advantages and disadvantages.

Let's first take a look at the syntax that is used to traverse a list using a *for* loop. This is by far one of the easiest modes of going through each of the values contained within a list. The *for* loop traverses the list one element at a time, allowing the developer to perform some action on each element if so desired.

```
>>> ourList=[1,2,3,4,5,6,7,8,9,10]
>>> for elem in ourList:
...     print elem
...
1
2
3
4
5
6
7
8
9
10
```

As you can see from this simple example, it is quite easy to go through a list and work with each item individually. The *for* loop syntax requires a variable to which each element in the list will be assigned for each pass of the loop. Additionally, we can still make use of the current index while traversing a loop this way if needed. The only requirement is to make use of the *index()* method on the list and pass the current element.

```
>>>ourList=[1,2,3,4,5,6,7,8,9,10]
>>> for elem in ourList:
...     print 'The current index is: %d' % (ourList.index(elem))
...
The current index is: 0
The current index is: 1
The current index is: 2
The current index is: 3
The current index is: 4
The current index is: 5
The current index is: 6
The current index is: 7
The current index is: 8
The current index is: 9
```

If we do not wish to go through each element within the list then that is also possible via the use of the *for* loop. In this case, we'll simply use a list slice to retrieve the exact elements we want to see. For instance, take a look at the following code which traverses through the first 5 elements in our list.

```
>>> for elem in ourList[0:5]:
...     print elem
...
1
2
```



```
3
4
5
```

To illustrate a more detailed example, let's say that you wished to retrieve every other element within the list.

```
>>> for elem in ourList[0::2]:
...     print elem
...
1
3
5
7
9
```

As you can see, doing so is quite easy by simply making use of the built-in features that Python offers.

List Comprehensions

There are some advanced features for lists that can help to make a developer's life easier. Once such feature is known as a *list comprehension*. While this concept may be daunting at first, it offers a good alternative to creating many separate lists manually or using `map()`. List comprehensions take a given list, and then iterate through it and apply a given expression against each of the objects in the list. This allows one to quickly take a list and alter it via the use of the provided expression. Of course, as with many other Python methods the list comprehension returns an altered copy of the list. The original list is left untouched.

Let's take a look at the syntax for a list comprehension. They are basically comprised of an expression of some kind followed by a *for* statement and then optionally more *for* or *if* statements. As they are a difficult technique to describe, let's take a look at some examples. Once you've seen list comprehensions in action you are sure to understand them and see how useful they can be.

```
# Create a list of ages and add one to each of those ages using a list comprehension
>>> ages=[20,25,28,30]
>>> [age+1 for age in ages]
[21, 26, 29, 31]

# Create a list of names and convert the first letter of each name to uppercase as it
↳ should be
>>> names=['jim','frank','vic','leo','josh']
>>> [name.title() for name in names]
['Jim', 'Frank', 'Vic', 'Leo', 'Josh']

# Create a list of numbers and return the square of each EVEN number
>>> numList=[1,2,3,4,5,6,7,8,9,10,11,12]
>>> [num*num for num in numList if num % 2 == 0]
[4, 16, 36, 64, 100, 144]

# Use a list comprehension with a range
>>> [x*5 for x in range(1,20)]
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
```

List comprehensions can make code much more concise and allows one to apply expressions or functions to list elements quite easily. Let's take a quick look at an example written in Java for performing the same type of work as an easy list comprehension. It is plain to see that list comprehensions are much more concise.

Java Code

```
// Define original integer array
int[] ages = {20, 25, 28, 30};

// Print original int array
System.out.println("Starting List:");

for (int age : ages) {
    System.out.println(age);
}

// Create new int array by adding one to each element of first array
int x = 0;
int[] newages = new int[4];
for (int age : ages) {
    newages[x] = age+1;
    x++;
}

// Print ending list
System.out.println("Ending List:");
for (int age : newages) {
    System.out.println(age);
}
```

Dictionaries

A dictionary is quite different than a typical list in Python as there is no automatically populated index for any given element within the dictionary. When you use a list, you need not worry about assigning an index to any value that is placed within it. However, a dictionary forces the developer to assign an index or “key” for every element that is placed into the construct. Therefore, each entry into a dictionary requires two values, the *key* and the *element*.

The beauty of the dictionary is that it allows the developer to choose the data type of the key value. Therefore, if one wishes to use a string value as a key then it is entirely possible. Dictionary types also have a multitude of methods and operations that can be applied to them to make them easier to work with.

Method or Operation	Description
<code>len(dictionary)</code>	Returns number of items within the given dictionary.
<code>dictionary[key]</code>	Returns the item from the list that is associated with the given key.
<code>dictionary[key] = value</code>	Sets the associated item in the list to the given value.
<code>del dictionary[key]</code>	Deletes the given key/value pair from the list.
<code>dictionary.clear()</code>	Removes all items from the dictionary.
<code>dictionary.copy()</code>	Creates a shallow copy of the dictionary.
<code>has_key(key)</code>	Returns a boolean stating whether the dictionary contains the given key.
<code>items()</code>	Returns a copy of the key/value pairs within the dictionary.
<code>keys()</code>	Returns the keys within the dictionary.
<code>update([dictionary2])</code>	Updates dictionary with the key/value pairs from the given dictionary. Existing keys will be overwritten.
<code>fromkeys(sequence[,value])</code>	Creates a new dictionary with keys from the given sequence. The values will be set to the values given.
<code>values()</code>	Returns the values within the dictionary.
<code>get(key[, b])</code>	Returns the value associated with the given key. If the key does not exist, then returns b.
<code>setdefault(key[, b])</code>	Returns the value associated with the given key. If the key does not exist, then returns and sets b.
<code>pop(key[, b])</code>	Returns and removes the value associated with the given key. If the key does not exist then returns b.
<code>popItem()</code>	Removes and returns the first key/value pair in the dictionary.
<code>iteritems()</code>	Returns an iterator over the key/value pairs in the dictionary.
<code>iterkeys()</code>	Returns an iterator over the keys in the dictionary.
<code>itervalues()</code>	Returns an iterator over the values in the dictionary.

Table 2-5. Mapping type methods and operations.

Now we will take a look at some dictionary examples. This reference will not show you an example of using each of the mapping operations, but it should provide you with a good enough base understanding of how they work.

```
# Create an empty dictionary and a populated dictionary
>>> myDict={}
>>> myDict.values()
[]
>>> myDict.has_key(1)
False
>>> myDict[1] = 'test'
>>> myDict.values()
['test']
>>> len(myDict)
1

# Replace the original dictionary with a dictionary containing string-based keys
# The following dictionary represents a hockey team line
>>> myDict = {'r_wing':'Josh','l_wing':'Frank','center':'Jim','l_defense':'Leo','r_
↳defense':'Vic'}
>>> myDict.values()
['Josh', 'Vic', 'Jim', 'Frank', 'Leo']
>>> myDict.get('r_wing')
'Josh'

# Iterate over the items in the dictionary
>>> hockeyTeam = myDict.iteritems()
>>> for player in hockeyTeam:
...     print player
...
('r_wing', 'Josh')
```

```
('r_defense', 'Vic')
('center', 'Jim')
('l_wing', 'Frank')
('l_defense', 'Leo')

>>> for key,value in myDict.iteritems():
...     print key, value
...
r_wing Josh
r_defense Vic
center Jim
l_wing Frank
l_defense Leo
```

Sets

Sets are unordered collections of unique elements. What makes sets different than other sequence types is that they contain no indexing. They are also unlike dictionaries because there are no key values associated with the elements. They are an arbitrary collection of unique elements. Sets cannot contain mutable objects, but they can be mutable.

There are two different types of sets, namely *set* and *frozenset*. The difference between the two is quite easily conveyed from the name itself. A regular *set* is a mutable collection object, whereas a *frozen* set is immutable. Much like sequences and mapping types, sets have an assortment of methods and operations that can be used on them. Many of the operations and methods work on both mutable and immutable sets. However, there are a number of them that only work on the mutable set types. In the two tables that follow, we'll take a look at the different methods and operations.

Method or Operation	Description
<code>len(set)</code>	Returns the number of elements in a given set.
<code>copy()</code>	
<code>difference(set2)</code>	
<code>intersection(set2)</code>	
<code>issubbset(set2)</code>	
<code>issuperset(set2)</code>	
<code>symmetric_difference(set2)</code>	
<code>union(set2)</code>	

Table 2-6. Set Type Methods and Operations

Method or Operation	Description
<code>add(item)</code>	Adds an item to a set if it is not already in the set.
<code>clear()</code>	Removes all items in a set.
<code>difference_update(set2)</code>	
<code>discard(item)</code>	
<code>intersection_update(set2)</code>	
<code>pop()</code>	
<code>remove()</code>	
<code>symmetric_difference_update(set2)</code>	
<code>update(set2)</code>	

Table 2-7. Mutable Set Type Methods and Operations

Tuples

Tuples are much like lists, however they are immutable. Once a tuple has been defined, it cannot be changed. They contain indexes just like lists, but again, they cannot be altered once defined. Therefore, the index in a tuple may be used to retrieve a particular value and not to assign or modify.

Since tuples are a member of the sequence type, they can use the same set of methods and operations available to all sequence types.

```
# Creating an empty tuple
>>> myTuple = ()

# Creating tuples and using them
>>> myTuple2 = (1, 'two', 3, 'four')
>>> myTuple2
(1, 'two', 3, 'four')
```

Jython Specific Collections

There are a number of Jython specific collection objects that are available for use. Most of these collection objects are used to pass data into Java classes and so forth, but they add additional functionality into the Jython implementation that will assist Python newcomers that are coming from the Java world. Nonetheless, many of these additional collection objects can be quite useful under certain situations.

In the Jython 2.2 release, Java collection integration was introduced. This enables a bidirectional interaction between Jython and Java collection types. For instance, a Java ArrayList can be imported in Jython and then used as if it were part of the language. Prior to 2.2, Java collection objects could act as a Jython object, but Jython objects could not act as Java objects.

```
# Import and use a Java ArrayList
>>> import java.util.ArrayList as ArrayList
>>> arr = ArrayList()
>>> arr.add(1)
True
>>> arr.add(2)
True
>>> print arr
[1, 2]
```

Ahead of the integration of Java collections, Jython also had implemented the *jarray* object which basically allows for the construction of a Java array in Jython. In order to work with a *jarray*, simply define a sequence type in Jython and pass it to the *jarray* object along with the type of object contained within the sequence. The *jarray* is definitely useful for creating Java arrays and then passing them into java objects, but it is not very useful for working in Jython objects. Moreover, all values within a *jarray* must be the same type. If you try to pass a sequence containing multiple types to a *jarray* then you'll be given a *TypeError* of one kind or another.

Character	Java Equivalent	
z		boolean
b		byte
c		char
d		double
f		float
h		short
i		int
l		long

Table 2-8. Character Typecodes for use with Jarray

```
>>> mySeq = (1,2,3,4,5)
>>> from jarray import array
>>> array(mySeq,int)
array(org.python.core.PyInteger, [1, 2, 3, 4, 5])

>>> myStr = "Hello Jython"
>>> array(myStr,'c')
array('c', 'Hello Jython')
```

Files

File objects are used to read and write data to a file on disk. The file object is used to obtain a reference to the file on disk and open it for reading, writing, appending, or a number of different tasks. If we simply use the *open(filename[, mode])* function, we can return a file type and assign it to a variable for processing. If the file does not yet exist on disk, then it will automatically be created. The *mode* argument is used to tell what type of processing we wish to perform on the file. This argument is optional and if omitted then the file is opened in read-only mode.

Mode	Description
'r'	read only
'w'	write
'a'	append
'r+'	read and write
'rb'	Windows binary file read
'wb'	Windows binary file write
'r+b'	Windows binary file read and write

Table 2-9. Modes of Operations for File Types

Open a file and assign it to variable f

There are plenty of methods that can be used on file objects for manipulation of the file content. We can call *read([size])* on a file in order to read it's content. Size is an optional argument here and it is used to tell how much content to read from the file. If it is omitted then the entire file content is read. The *readline()* method can be used to read a single line from a file. *readlines([size])* is used to return a list containing all of the lines of data that are contained within a file. Again, there is an optional *size* parameter that can be used to tell how many bytes from the file to read. If we wish to place content into the file, the *write(string)* method does just that. The *write()* method writes a string to the file.

When writing to a file it is oftentimes important to know exactly what position in the file you are going to write to. There are a group of methods to help us out with positioning within a file using integers to represent bytes in the file. The *tell()* method can be called on a file to give the file object's current position. The integer returned is in bytes and is an offset from the beginning of the file. The *seek(offset, from)* method can be used to change position in a file. The *offset* is the number in bytes of the position you'd like to go, and *from* represents the place in the file where you'd like to calculate the *offset* from. If *from* equals 0, then the offset will be calculated from the beginning of the file. Likewise, if it equals 1 then it is calculated from the current file position, and 2 will be from the end of the file. The default is 0 if *from* is omitted.

Lastly, it is important to allocate and de-allocate resources efficiently in our programs or we will incur a memory overhead and leaks. The *close()* method should be called on a file when we are through working with it. The proper methodology to use when working with a file is to open, process, and then close each time. However, there are more efficient ways of performing such tasks. In Chapter 5 we will discuss the use of context managers to perform the same functionality in a more efficient manner.

```
File Manipulation in Python
# Create a file, write to it, and then read it's content
```

```

>>> f = open('newfile.txt', 'r+')
>>> f.write('This is some new text for our file\n')
>>> f.write('This should be another line in our file\n')
# No lines will be read because we are at the end of the written content
>>> f.read()
''
>>> f.readlines()
[]
>>> f.tell()
75L
# Move our position back to the beginning of the file
>>> f.seek(0)
>>> f.read()
'This is some new text for our file\nThis should be another line in our file\n'
>>> f.seek(0)
>>> f.readlines()
['This is some new text for our file\n', 'This should be another line in our file\n']
>>> f.close()

```

Iterators

The iterator type was introduced into Python back in version 2.2. It allows for iteration over Python containers. All iterable containers have built-in support for the iterator type. For instance, sequence objects are iterable as they allow for iteration over each element within the sequence. If you try to return an iterator on an object that does not support iteration, you will most likely receive an *AttributeError* which tells you that `__iter__` has not been defined as an attribute for that object.

Iterators allow for easy access to sequences and other iterable containers. Some containers such as dictionaries have specialized iteration methods built into them as you have seen in previous sections. Iterator objects are required to support two main methods that form the iterator protocol. Those methods are defined below.

Method	Description	
iterator. <code>__iter__()</code>	Returns the iterator object on a container. Required to allow use with <i>for</i> and <i>in</i> statements	
iterator. <code>next()</code>	Returns the next item from a container.	

Table 2-10: Iterator Protocol

To return an iterator on a container, just assign `container.__iter__()` to some variable. That variable will become the iterator for the object. If using the `next()` call, it will continue to return the next item within the list until all items have been retrieved. Once this occurs, a *StopIteration* error is issued. The important thing to note here is that we are actually creating a copy of the list when we return the iterator and assign it to a variable. That variable returns and removes an item from that copy each time the `next()` method is called on it. If we continue to call `next()` on the iterator variable until the *StopIteration* error is issued, the variable will no longer contain any items and is empty.

Referencing and Copies

Creating copies and referencing items in the Python language is fairly straightforward. The only thing you'll need to keep in mind is that the techniques used to copy mutable and immutable objects differ a bit.

In order to create a copy of an immutable object, you simply assign it to a different variable. The new variable is an exact copy of the object. If you attempt to do the same with a mutable object, you will actually just create a reference to the original object. Therefore, if you perform operations on the “copy” of the original then the same operation will actually be performed on the original. This occurs because the new assignment references the same mutable object in

memory as the original. It is kind of like someone calling you by a different name. One person may call you by your birth name and another may call you by your nickname, but both names will reference you of course.

To effectively create a copy of a mutable object, you have two choices. You can either create what is known as a *shallow* copy or a *deep* copy of the original object. The difference is that a shallow copy of an object will create a new object and then populate it with references to the items that are contained in the original object. Hence, if you modify any of those items then each object will be affected since they both reference the same items. A deep copy creates a new object and then recursively copies the contents of the original object into the new copy. Once you perform a deep copy of an object then you can perform operations on the copied object without affecting the original. You can use the *deepcopy* function in the Python standard library to create such a copy. Let's look at some examples of creating copies in order to give you a better idea of how this works.

```
# Create an integer variable, copy it, and modify the copy
>>> a = 5
>>> b = a
>>> print b
5
>>> b = a * 5
>>> b
25
>>> a
5

# Create a list, assign it to a different variable and then modify
>>> listA = [1,2,3,4,5,6]
>>> print listA
[1, 2, 3, 4, 5, 6]
>>> listB = listA
>>> print listB
[1, 2, 3, 4, 5, 6]
>>> del listB[2]
# Oops, we've altered the original list!
>>> print listA
[1, 2, 4, 5, 6]

# Create a deep copy of the list and modify it
>>> import copy
>>> listA = [1,2,3,4,5,6]
>>> listB = copy.deepcopy(listA)
>>> print listA
[1, 2, 3, 4, 5, 6]
>>> del listB[2]
>>> print listB
[1, 2, 4, 5, 6]
>>> print listA
[1, 2, 3, 4, 5, 6]
```

Garbage Collection

This is one of those major differences between CPython and Jython. Unlike CPython, Jython does not implement a reference counting technique for aging out or garbage collection unused objects. Instead, Jython makes use of the garbage collection mechanisms that the Java platform provides. When a Jython object becomes stale or unreachable, the JVM may or may not reclaim it. One of the main aspects of the JVM that made developers so happy in the early days is that there was no longer a need to worry about cleaning up after your code. In the C programming language, one must maintain an awareness of which objects are currently being used so that when they are no longer needed the program would perform some clean up. Not in the Java world, the gc thread on the JVM takes care of all garbage

collection and cleanup for you. This is a benefit of using the Jython implementation; unlike Python there is no need to worry about reference counting.

Even though we haven't spoken about classes yet, it is a good time to mention that Jython provides a mechanism for object cleanup. A finalizer method can be defined in any class in order to ensure that the garbage collector performs specific tasks. Any cleanup code that needs to be performed when an object goes out of scope can be placed within this finalizer method. It is important to note that the finalizer method cannot be counted on as a method which will always be invoked when an object is stale. This is the case because the finalizer method is invoked by the Java garbage collection thread, and there is no way to be sure when and if the garbage collector will be called on an object. Another issue of note with the finalizer is that they incur a performance penalty. If you're coding an application that already performs poorly then it may not be a good idea to throw lots of finalizers into it.

Below is an example of a Jython finalizer. It is an instance method that must be named `__del__`.

```
class MyClass:
    def __del__(self):
        pass      # Perform some cleanup here
```

The downside to using the JVM garbage collection mechanisms is that there is really no guarantee as to when and if an object will be reclaimed. Therefore, when working with performance intensive objects it is best to not rely on a finalizer to be called. It is always important to ensure that proper coding techniques are used in such cases when working with objects like files and databases. Never code the `close()` method for a file into a finalizer because it may cause an issue if the finalizer is not invoked. Best practice is to ensure that all mandatory cleanup activities are performed before a finalizer would be invoked.

Summary

A lot of material was covered in this chapter. You should be feeling better acquainted with Python after reading through this material. We began the chapter by covering the basics of assignment and assigning data to particular objects or data types. We learned that working with each type of data object opens different doors as the way we work with each type of data object differs. Our journey into data objects began with numbers and strings, and we discussed the many functions available to the string object. We learned that strings are part of the sequence family of Python collection objects along with lists and tuples. We covered how to create and work with lists, and the variety of options available to us when using lists. We discovered that list comprehensions can help us create copies of a given list and manipulate their elements according to an expression or function. After discussing lists, we went on to discuss dictionaries, sets and tuples. These objects give us different alternatives to the list object.

After discussing the collection types, we learned that Jython has its own set of collection objects that differ from those in Python. We can leverage the advantage of having the Java platform at our fingertips and use Java collection types from within Jython. Likewise, we can pass a Jython collection to Java as a *jarray* object. We followed that topic with a discussion of file objects and how they are used in Python. The topic of iteration and creating iterables followed. We finished up by discussing referencing, copies, and garbage collection. We saw how creating different copies of objects does not always give you what you'd expect, and that Jython garbage collection differs quite a bit from that of Python.

The next chapter will help you to combine some of the topics you've learned about in this chapter as you will learn how to define expressions and work with control flow.

Chapter 3: Operators, Expressions, and Program Flow

Up until this point, we've have not yet covered the different means of writing expressions in Python. The focus of this chapter is to go in depth on each of the ways we can evaluate code, and write meaningful blocks of conditional logic. We'll cover the details of each operator that can be used in Python expressions. This chapter will also cover some topics that have already been discussed in more meaningful detail.

We will begin by discussing details of expressions. We have already seen some expressions in use while reading through the previous chapters. Here we will focus more on the internals of operators used to create expressions, and also different types of expressions that we can use. This chapter will go into further detail on how we can define blocks of code for looping and conditionals.

Types of Expressions

An expression in Python is a block of code that produces a result or value. Most often, we think of expressions that are used to perform mathematical operations within our code. However, there are a multitude of expressions used for other purposes as well. In Chapter 2, we covered the details of String manipulation, sequence and dictionary operations, and touched upon working with sets. All of the operations performed on these objects are forms of expressions in Python.

This chapter will go into detail on how you write and evaluate mathematical expressions, boolean expressions, and augmented assignments.

Mathematical Operations

The Python language of course contains all of your basic mathematical operations. This section will briefly touch upon each operator that is available for use in Python and how they function. You will also learn about a few built-in functions which can be used to assist in your mathematical expressions. Finally, you'll see how to use conditionals in the Python language and learn order of evaluation.

Assuming that this is not the first programming language you are learning, there is no doubt that you are at least somewhat familiar with performing mathematical operations within your programs. Python is no different than the rest when it comes to mathematics, as with most programming languages, performing mathematical computations and working with numeric expressions is straightforward.

Operator	Description
'+'	Addition
'-'	Subtraction
'*'	Multiplication
'/'	Division
'//'	Truncating Division
'%'	Modulo (Remainder of Division)
'**'	Power Operator
'+var'	Unary Plus
'-var'	Unary Minus

Table 3-1: Numeric Operators

Most of the operators in the table above are easily understood. However, the truncating division, modulo, power, and unary operators could use some explanation. Truncating division will automatically truncate a division result into an integer. Modulo will return the remainder of a division operation. The power operator does just what you'd expect as it returns the result of the number to the left of the operator multiplied by itself *n* times, where *n* represents the number to the right of the operator. Unary plus and unary minus are used to evaluate positive or negative numbers. The following set of examples will help to clarify these topics.

```
# Performing basic mathematical computations

>>> 10 - 6
4
>>> 9 * 7
63
>>> 9 / 3
```

```

3
>>> 10 / 3
3
>>> 10 // 3
3
>>> 3.14 / 2
1.57
>>> 3.14 // 2
1.0
>>> 36 / 5
7
>>> 36 % 5
1
>>> 5**2
25
>>> 100**2
10000
>>> -10 + 5
-5
>>> +5 - 5
0

```

There is a new means of division available in Jython 2.5 by importing from `__future__`. In a standard division for 2.5 and previous releases, the quotient returned is an integer or the floor of the quotient when arguments are ints or longs. However, a reasonable approximation of the division is returned if the arguments are floats or complex. Often times this solution is incorrect as the quotient should be the reasonable approximation or “true division” in any case. When we import *division* from the `__future__` module then we alter the return value of division by causing true division when using the `/` operator, and floor division when using the `//` operator. Since this is going to be the standard in future releases, it is best practice to import from `__future__` when performing division in Jython 2.5.

```

>>> from __future__ import division
>>> 9/5
1.8
>>> 9/4
2.25
>>> 9/3
3.0
>>> 9//3
3
>>> 9//5
1

```

As stated at the beginning of the section, there are a number of built-in mathematical functions that are at your disposal.

Function	Description
<code>abs(var)</code>	Absolute value
<code>pow(x, y)</code>	Used in place of power operator
<code>pow(x,y,modulo)</code>	Ternary power-modulo
<code>round(var[, n])</code>	Returns a value rounded to the nearest 10-n
<code>divmod(x, y)</code>	Returns both the quotient and remainder of division operation

Table 3-2: Mathematical Built-in functions

```

# The following code provides some examples for using mathematical built-ins
>>> abs(9)
9
>>> abs(-9)
9

```

```
>>> divmod(8, 4)
(2, 0)
>>> pow(8, 2)
64
>>> pow(8, 2, 3)
1
>>> round(5.67, 1)
5.7
>>> round(5.67)
6.0
```

The bitwise and logical operators as well as the conditional operators can be used to combine and compare logic. As with the mathematical operators described above, these operators have no significant difference to that of Java.

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
!=	Not equal
==	Equal
&	Bitwise and
	Bitwise or
^	Bitwise xor
~	Bitwise negation
<<	Shift left
>>	Shift right

Table 3-3: Bitwise and Conditional Operators

Augmented assignment operators are those that combine two or more operations into one. While augmented assignment can assist in coding concisely, some say that too many such operators can make code more difficult to read.

Operator	Description and Logic
+=	$a = a + b$
-=	$a = a - b$
*=	$a = a * b$
/=	$a = a / b$
%=	$a = a \% b$
//=	$a = a // b$
=	$a = a^{} b$
&=	$a = a \& b$
=	$a = a b$
^=	$a = a \wedge b$
>>=	$a = a \gg b$
<<=	$a = a \ll b$

Table 3-4: Augmented Assignment Operators

Boolean Expressions

Comparing two or more values or expressions also uses a similar syntax to that of other languages, and the logic is quite the same. Note that in Python, *True* and *False* are very similar to constants in the Java language. *True* actually represents the number 1, and *False* represents the number 0. One could just as easily code using 0 and 1 to represent the boolean values, but for readability and maintenance the *True* and *False* “constants” are preferred. Java developers, make sure that you capitalize the first letter of these two words as you will receive an ugly *NameError* if you do not.

Conditional	Logic	
and		In an <i>x and y</i> evaluation, both x and y must evaluate to True
or		In an <i>x or y</i> evaluation, if x is false then y is evaluated.
not		In a <i>not x</i> evaluation, if <i>not x</i> , we mean the opposite of x

Table 3-5: Boolean Conditionals

Conversions

There are a number of conversion functions built into the language in order to help conversion of one data type to another. While every data type in Jython is actually a class object, these conversion functions will really convert one class type into another. For the most part, the built-in conversion functions are easy to remember because they are primarily named after the type to which you are trying to convert.

Function	Description
chr(value)	Converts integer to a character
complex(real [,imag])	Produces a complex number
dict(sequence)	Produces a dictionary from a given sequence of (key,value) tuples
eval(string)	Evaluates a string to return an object. . . useful for mathematical computations
float(value)	Converts to float
frozenset(set)	Converts a set into a frozen set
hex(value)	Converts an integer into a hex string
int(value [, base])	Converts to an integer using a base if a string is given
list(sequence)	Converts a given sequence into a list
long(value [, base])	Converts to a long using a base if a string is given
oct(value)	Converts integer to octal
ord(value)	Converts a character into it's integer value
repr(value)	Converts object into an expression string. Same as enclosing expression in reverse quotes (<i>x + y</i>)
set(sequence)	Converts a sequence into a set
str(value)	Converts an object into a string
tuple(sequence)	Converts a given sequence to a tuple
unichr(value)	Converts integer to a Unicode character

Table 3-6: Conversion Functions

The following is an example of using the *eval()* functionality as it is perhaps the one conversion function for which an example helps to understand.

```
# Suppose keyboard input contains an expression in string format (x * y)
>>> x = 5
>>> y = 12
>>> keyboardInput = 'x * y'
>>> eval(keyboardInput)
60
```

Program Flow

The Python programming language has structure that sets it apart from the others. As you've learned in previous references in this book, the statements that make up programs in Python are structured with attention to spacing, order, and technique. In order to develop a statement in Python, you must adhere to proper spacing techniques throughout the code block. Convention and good practice adhere to four spaces of indentation per statement throughout the entire program. Follow this convention along with some control flow and you're sure to develop some easily maintainable software.

The standard Python *if-else* conditional statement is used in order to evaluate expressions and branch program logic based upon the outcome. Expressions that are usable in an *if-else* statement can consist of any operators we've discussed previously. The objective is to write and compare expressions in order to evaluate to a *True* or *False* outcome. As shown in Chapter 1, the logic for an *if-else* statement follows one path if an expression evaluates to *True*, or a different path if it evaluates to *False*.

You can chain as many *if-else* expressions together as needed. The combining *if-else* keyword is *elif*, which is used for every expression in between the first and the last expressions within a conditional statement.

```
# terminal symbols are left out of this example so that you can see the concise
↪indentation
pi =3.14
x = 2.7 * 1.45
if x == pi:
    print 'The number is pi'
elif x > pi:
    print 'The number is greater than pi'
else:
    print 'The number is less than pi'
```

Another construct that we touched upon in Chapter 1 was the loop. Every programming language provides looping implementations, and Python is no different. The Python language provides two main types of loops known as the *while* and the *for* loop. The *while* loop logic follows the same semantics as the *while* loop in Java. The loop will continue processing until the expression evaluates to *False*. At this time the looping ends and that would be it for the Java implementation. However, in Python the *while * loop construct also contains an *else* clause which is executed when the looping completes.

```
while True:
    # perform some processing
else:
    print 'Processing Complete...'
```

This *else* clause can come in handy while performing intensive processing so that we can inform the user of the completion of such tasks. It can also be handy when debugging code. Also mentioned in Chapter 1 were the *break*, and *continue* statements. These all come in handy when using any looping construct. The *break* statement can be used to break out of a loop. It should be noted that if there are nested loops then the *break* statement will break out of the inner-most loop only, the outer loops will continue to process. The *continue* statement can be used to break out of the current processing statement and continue the loop from the beginning. The *continue* can be thought of as a skipping statement as it will cause execution to skip all remaining statements in the block and restart from the beginning (if the loop expression still evaluates to *True* of course).

```
while x != y:
    # perform some processing
    if x < 0:
        break
else:
    print 'The program executed to completion'
```

In the example above, the program will continue to process until *x* does not equal *y*. However, if at any point during the processing the *x* variable evaluates less than zero, then the execution stops. The *else* clause will not be executed if the *break* statement is invoked. It will only be executed under normal termination of the loop.

The *for* loop can be used on any iterable object. It will simply iterate through the object and perform some processing during each pass. Both the *break* and *continue* statements can also be used within the *for* loop. The *for* statement in Python also differs from the same statement in Java because in Python we also have the *else* clause with this construct. Once again, the *else* clause is executed when the *for* loop processes to completion without any *break* intervention. Also, if you are familiar with pre-Java 5 *for* loops then you will love the Python syntax. In Java 5, the syntax of the *for* statement was adjusted a bit to make it more in line with syntactically easy languages such as Python.

```
for(x = 0; x <= myList.size(); x++){
    // processing statements iterating through myList
    System.out.println("The current index is: " + x);
}

x = 0
for value in myList:
    # processing statements using value as the current item in myList
    print 'The current index is %i' % (x)
    x = x + 1
```

As you can see, the Python syntax is a little easier to understand, but it doesn't really save too many keystrokes at this point. We still have to manage the index (*x* in this case) by ourselves by incrementing it with each iteration of the loop. However, Python does provide a built-in function that can save us some keystrokes and provides a similar functionality to that of Java with the automatically incrementing index on the *for* loop. The *enumerate(sequence)* function does just that. It will provide an index for our use and automatically manage it for us.

```
>>> myList = ['jython', 'java', 'python', 'jruby', 'groovy']
>>> for index, value in enumerate(myList):
...     print index, value
...
0 jython
1 java
2 python
3 jruby
4 groovy
```

Now we have covered the program flow for conditionals and looping constructs in the Python language. Note that you can nest to any level, and provide as many *if-else* conditionals as you'd like. However, good programming practice will tell you to keep it as simple as possible or the logic will become too hard to follow.

Example Code

Let's take a look at an example program that uses some of the program flow which was discussed in this chapter. The example program simply makes use of an external text file to manage a list of players on a sports team. You will see how to follow proper program structure and use spacing effectively in this example. You will also see file utilization in action, along with utilization of the *raw_input()* function.

```
playerDict = {}
saveFile = False
exitSystem = False
# Enter a loop to enter information from keyboard
while not exitSystem:
    print 'Sports Team Administration App'
    enterPlayer = raw_input("Would you like to create a team or manage an existing_
↵team?\n (Enter 'C' for create, 'M' for manage, 'X' to exit) ")
```

```
if enterPlayer.upper() == 'C':
    exitSystem = False
    # Enter a player for the team
    print 'Enter a list of players on our team along with their position'
    enterCont = 'Y'
    # While continuing to enter new players, perform the following
    while enterCont.upper() == 'Y':
        name = raw_input('Enter players first name: ')
        position = raw_input('Enter players position: ')
        playerDict[name] = position
        saveFile = True
        enterCont = raw_input("Enter another player? (Press 'N' to exit or 'Y' to
↪continue)")
    else:
        exitSystem = True
elif enterPlayer.upper() == 'M':
    exitSystem = False
    # Read values from the external file into a dictionary object
    print '\n'
    print 'Manage the Team'
    playerfile = open('players.txt','r')
    for player in playerfile:
        playerList = player.split(':')
        playerDict[playerList[0]] = playerList[1]
    print 'Team Listing'
    print '+++++++'
    for i, player in enumerate(playerDict):
        print 'Player %s Name: %s -- Position: %s' %(i, player,
↪playerDict[player])
    else:
        exitSystem = True
else:
    # Save the external file and close resources
    if saveFile:
        print 'Saving Team Data...'
        playerfile = open('players.txt','w')
        for player in playerDict:
            playerfile.write(player + ':' + playerDict[player] + '\n')
        playerfile.close()
```

Summary

All programs are constructed out of definitions, statements and expressions. In this chapter we covered details of creating expressions and using them. Expressions can be composed of any number of mathematical operators and comparisons. In this chapter we discussed the basics of using mathematical operators in our programs. The `__future__` division topic introduced us to using features from the `__future__`. We then delved into comparisons and comparison operators.

We ended this short chapter by discussing proper program flow and properly learned about the *if* statement as well as how to construct different types of loops in Python. In the next chapter you will learn how to write functions, and the use of many built-in functions will be discussed.

Chapter 4: Defining Functions and Using Built-Ins

Functions are the fundamental unit of work in Python. A function in Python performs a task and returns a result. In this chapter, we will start with the basics of functions. Then we look at using the builtin functions. These are the core functions that are always available, meaning they don't require an explicit import into your namespace. Next we will look at some alternative ways of defining functions, such as lambdas and classes. We will also look at more advanced types of functions, namely closures and generator functions.

As you will see, functions are very easy to define and use. Python encourages an incremental style of development that you can leverage when writing functions. So how does this work out in practice? Often when writing a function it may make sense to start with a sequence of statements and just try it out in a console. Or maybe just write a short script in an editor. The idea is to just to prove a path and answer such questions as, "Does this API work in the way I expect?" Because top-level code in a console or script works just like it does in a function, it's easy to later isolate this code in a function body and then package it as a function, maybe in a library, or as a method as part of a class. The ease of doing this style of development is one aspect that makes Python such a joy to program in. And of course in the Jython implementation, it's easy to use this technique within the context of any Java library.

An important thing to keep in mind is that functions are first-class objects in Python. They can be passed around just like any other variable, resulting in some very powerful solutions. We'll see some examples of using functions in such a way later in this chapter.

Function Syntax and Basics

Functions are usually defined by using the `def` keyword, the name of the function, its parameters (if any), and the body of code. We will start by looking at this example function:

```
def times2(n):  
    return n * 2
```

In this example, the function name is `times2` and it accepts a parameter `n`. The body of the function is only one line, but the work being done is the multiplication of the parameter by the number 2. Instead of storing the result in a variable, this function simply returns it to the calling code. An example of using this function would be as follows.

```
>>> times2(8)  
16  
>>> x = times2(5)  
>>> x  
10
```

Normal usage can treat function definitions as being very simple. But there's subtle power in every piece of the function definition, due to the fact that Python is a dynamic language. We look at these pieces from both a simple (the more typical case) and a more advanced perspective. We will also look at some alternative ways of creating functions in a later section.

The `def` Keyword

Using `def` for *define* seems simple enough, and this keyword certainly can be used to declare a function just like you would in a static language. You should write most code that way in fact.

However, the more advanced usage is that a function definition can occur at any level in your code and be introduced at any time. Unlike the case in a language like C or Java, function definitions are not declarations. Instead they are *executable statements*. You can nest functions, and we'll describe that more when we talk about nested scopes. And you can do things like conditionally define them.

This means it's perfectly valid to write code like the following:

```
if variant:
    def f():
        print "One way"
else:
    def f():
        print "or another"
```

Please note, regardless of when and where the definition occurs, including its variants as above, the function definition will be compiled into a function object at the same time as the rest of the module or script that the function is defined in.

Naming the Function

We will describe this more in a later section, but the `dir` builtin function will tell us about the names defined in a given namespace, defaulting to the module, script, or console environment we are working in. With this new `times2` function defined above, we now see the following (at least) in the console namespace:

```
>>> dir()
['__doc__', '__name__', 'times2']
```

We can also just look at what is bound to that name:

```
>>> times2
<function times2 at 0x1>
```

(This object is further introspectable. Try `dir(times2)` and go from there.)

We can also redefine a function at any time:

```
>>> def f(): print "Hello, world"
...
>>> def f(): print "Hi, world"
...
>>> f()
Hi, world
```

This is true not just of running it from the console, but any module or script. The original version of the function object will persist until it's no longer referenced, at which point it will be ultimately be garbage collected. In this case, the only reference was the name `f`, so it became available for GC immediately upon rebinding.

What's important here is that we simply rebound the name. First it pointed to one function object, then another. We can see that in action by simply setting another name (equivalently, a variable) to `times2`.

```
>>> t2 = times2
>>> t2(5)
10
```

This makes passing a function as a parameter very easy, for a callback for example. But first, we need to look at function parameters in more detail.

Function Metaprogramming

A given name can only be associated with one function at a time, so can't overload a function with multiple definitions. If you were to define two or more functions with the same name, the last one defined is used, as we saw.

However, it is possible to overload a function, or otherwise genericize it. You simply need to create a dispatcher function that then dispatches to your set of corresponding functions.

Function Parameters and Calling Functions

When defining a function, you specify the parameters it takes. Typically you will see something like the following. The syntax is familiar:

```
def tip_calc(amt, pct)
```

Calling a function is symmetric. You can call a function. The parentheses are mandatory. Calling functions is also done by with a familiar syntax. For example, for the function `x` with parameters `a`, `b`, `c` that would be `x(a,b,c)`. Unlike some other dynamic languages like Ruby and Perl, the use of parentheses is required syntax (due the function name being just like any other name).

Objects are strongly typed, as we have seen. But function parameters, like names in general in Python, are not typed. This means that any parameter can refer to any type of object.

We see this play out in the `times2` function. The `*` operator not only means multiply for numbers, it also means repeat for sequences (like strings and lists). So you can use the `times2` function as follows:

```
>>> times2(4)
8
>>> times2('abc')
'abccabc'
>>> times2([1,2,3])
[1, 2, 3, 1, 2, 3]
```

All parameters in Python are passed by reference. This is identical to how Java does it with object parameters. However, while Java does support passing unboxed primitive types by value, there are no such entities in Python. Everything is an object in Python.

Functions are objects too, and they can be passed as parameters:

```
# Define a function that takes two values and a mathematical function
>>> def perform_calc(value1, value2, func):
...     return func(value1, value2)
...
# Define a mathematical function to pass
>>> def mult_values(value1, value2):
...     return value1 * value2
...
>>> perform_calc(2, 4, mult_values)
8

# Define another mathematical function to pass
>>> def add_values(value1, value2):
...     return value1 + value2
...
>>> perform_calc(2, 4, add_values)
6
>>>
```

If you have more than two or so arguments, it often makes more sense to call a function by parameter, rather than by the defined order. This tends to create more robust code. So if you have a function `draw_point(x, y)`, you might want to call it as `draw_point(x=10, y=20)`.

Defaults further simplify calling a function. You use the form of `param=default_value` when defining the function. For instance, you might take our `times2` function and generalize it.

```
def times_by(n, by=2):  
    return n * by
```

This function is equivalent to `times2` when called using that default value.

There's one point to remember that oftens trips up developers. The default value is initialized exactly once, when the function is defined. That's certainly fine for immutable values like numbers, strings, tuples, frozensets, and similar objects. But you need to ensure that if the default value is mutable, that it's being used in this fashion correctly. So a dictionary for a shared cache makes sense. But this mechanism won't work for but a list where we expect it is initialized to an empty list upon invocation. If you're doing that, you need to write that explicitly in your code.

Lastly, a function can take an unspecified number of ordered arguments, through `*args`, and keyword args, through `**kwargs`. These parameter names (`args` and `kwargs`) are conventional, so you can use whatever name makes sense for your function. The markers `*` and `**` are used to to determine that this functionality should be used. The single `*` argument allows for passing a sequence of values, and a double `**` argument allows for passing a dictionary of names and values. If either of these types of arguments are specified, they must follow any single arguments in the function declaration. Furthermore, the double `**` must follow the single `*`.

Definition of a function that takes a sequence of numbers:

```
def sum_args(*nums):  
    return sum(nums)
```

Calling the function using a sequence of numbers:

```
>>> seq = [6,5,4,3]  
>>> sum_args(*seq)  
18
```

Note: This is not exhaustive. You can also use tuple parameters, but in practice, they are not typically used, and were removed in Python 3. We recommend you don't use them. For one thing, they cannot be properly introspected from Jython.

Recursive Function Calls

It is also quite common to see cases in which a function calls itself from inside the function body. This type of function call is known as a recursive function call. Let's take a look at a function that computes the factorial of a given argument. This function calls itself passing in the provided argument decremented by 1 until the argument reaches the value of 0 or 1.

```
def fact(n):  
    if n in (0, 1):  
        return 1  
    else:  
        return n * fact(n - 1)
```

It is important to note that Jython is not unlike CPython in that it is ultimately stack based. Stacks are regions of memory where data is added and removed in a last-in first-out manner. If a recursive function calls itself too many times then it is possible to exhaust the stack, which results in an *OutOfMemoryError*. Therefore, be cautious when developing software using recursion.

Function Body

This section will break down the different components that comprise the body of a function. The body of a function is the part that performs the work. Throughout the next couple of sub-sections, you will see that a function body can be comprised of many different parts.

Documenting Functions

First, you should specify a document string for the function. The docstring, if it exists, is a string that occurs as the first value of the function body.

```
def times2(n):
    """Given n, returns n * 2"""
    return n * 2
```

As mentioned in chapter 1, by convention we use triple-quoted strings, even if your docstring is not multiline. If it is multiline, this is how we recommend you format it.

```
def fact(n):
    """Returns the factorial of n

    Computes the factorial of n recursively. Does not check its
    arguments if nonnegative integer or if would stack
    overflow. Use with care!
    """

    if n in (0, 1):
        return 1
    else:
        return n * fact(n - 1)
```

Any such docstring, but with leading indendetation stripped, becomes the `__doc__` attribute of that function object. Incidentally, docstrings are also used for modules and classes, and they work exactly the same way.

You can now use the `help` built-in function to get the docstring, or see them from various IDEs like PyDev for Eclipse and nbPython for NetBeans as part of the auto-complete.

```
>>> help(fact)
Help on function fact in module __main__:

fact(n)
    Returns the factorial of n

>>>
```

Returning Values

All functions return some value. In `times2`, we use the `return` statement to exit the function with that value. Functions can easily return multiple values at once by returning a tuple or other structure. The following is a simple example of a function that returns more than one value. In this case, the tip calculator returns the result of a tip based upon two percentage values.

```
>>> def calc_tips(amount):
...     return (amount * .18), (amount * .20)
... 
```

```
>>> calc_tips(25.25)
(4.545, 5.0500000000000001)
```

A function can return at any time, and it can also return any object as its value. So you can have a functions that look like the following:

```
>>> def check_pos_perform_calc(num1, num2, func):
...     if num1 > 0 and num2 > 0:
...         return func(num1, num2)
...     else:
...         return 'Only positive numbers can be used with this function!'
...
>>> def mult_values(value1, value2):
...     return value1 * value2
...
>>> check_pos_perform_calc(3, 4, mult_values)
12
>>> check_pos_perform_calc(3, -44, mult_values)
'Only positive numbers can be used with this function!'
```

If a return statement is not used, the value `None` is returned. There is no equivalent to a `void` method in Java, because every function in Python returns a value. However, the Python console will not show the return value when it's `None`, so you need to explicitly print it to see what is returned.

```
>>> do_nothing()
>>> print do_nothing()
None
```

Introducing Variables

A function introduces a scope for new names, such as variables. Any names that are created in the function are only visible within that scope. In the following example, the `sq` variable is defined within the scope of the function definition itself. If we try to use it outside of the function then we'll receive an error.

```
>>> def square_num(num):
...     """ Return the square of a number """
...     sq = num * num
...     return sq
...
>>> square_num(35)
1225
>>> sq
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sq' is not defined
```

Global Variables

global keyword - [Useful for certain circumstances, certainly not core/essential, much like `nonlocal` in Py3K, so let's not put too much focus on it.]

The *global* keyword is used to declare that a variable name is from the module scope (or script) containing this function. Using *global* is rarely necessary in practice, since it is not necessary if the name is called as a function or an attribute is accessed (through dotted notation).

This is a good example of where Python is providing a complex balancing between a complex idea - the lexical scoping of names, and the operations on them - and the fact that in practice it is doing the right thing.

Here is an example of using a global variable in the same `square_num()` function.

```
>>> sq = 0
>>> def square_num(n):
...     global sq
...     sq = n * n
...     return sq
...
>>> square_num(10)
100
>>> sq
100
```

Other Statements

What can go in a function body? Pretty much any statement, including material that we will cover later in this book. So you can define functions or classes or use even import, within the scope of that function.

In particular, performing a potentially expensive operation like import as last as possible, can reduce the startup time of your app. It's even possible it will be never needed too.

There are a couple of exceptions to this rule. In both cases, these statements must go at the beginning of a module, similar to what we see in a static language like Java:

- Compiler directives. Python supports a limited set of compiler directives that have the provocative syntax of `from __future__ import X`; see [PEP 236](#). These are features that will be eventually be made available, generally in the next minor revision (such as 2.5 to 2.6). In addition, it's a popular place to put Easter eggs, such as `from __future__ import braces`. (Try it in the console, which also relaxes what it means to be performed at the beginning.)
- Source encoding declaration. Although technically not a statement – it's in a specially parsed comment – this must go in the first or second line.

Empty Functions

It is also possible to define an empty function. Why have a function that does nothing? As in math, it's useful to have an operation that stands for doing nothing, like “add zero” or “multiply by one”. These identity functions eliminate special cases. Likewise, as see with `empty_callback`, we may need to specify a callback function when calling an API, but nothing actually needs to be done. By passing in an empty function – or having this be the default – we can simplify the API. An empty function still needs something in its body. You can use the `pass` statement.

```
def do_nothing():
    pass # here's how to specify an empty body of code
```

Or you can just have a docstring for the function body as in the following example.

```
def empty_callback(*args, **kwargs):
    """Use this function where we need to supply a callback,
       but have nothing further to do.
    """
```

Miscellaneous Information for the Curious Reader

As you already know, Jython is an interpreted language. That is, the Python code that we write for a Jython application is ultimately compiled down into Java bytecode when our program is run. So oftentimes it is useful for Jython developers to understand what is going on when this code is interpreted into Java bytecode.

What do functions look like from Java? They are instances of an object named `PyObject`, supporting the `__call__` method.

Additional introspection is available. If a function object is just a standard function written in Python, it will be of class `PyFunction`. A builtin function will be of class `PyBuiltinFunction`. But don't assume that in your code, because many other objects support the function interface (`__call__`), and these potentially could be proxying, perhaps several layers deep, a given function. You can only assume it's a `PyObject`.

Much more information is available by going to the Jython wiki. You can also send questions to the jython-dev mailing list for more specifics.

Builtin Functions

Builtin functions are those functions that are always in the Python namespace. In other words, these functions – and builtin exceptions, boolean values, and some other objects – are the only truly globally defined names. If you are familiar with Java, they are somewhat like the classes from `java.lang`.

Builtins are rarely sufficient, however; even a simple command line script generally needs to parse its arguments or read in from its standard input. So for this case you would need to `import sys`. And in the context of Jython, you will need to import the relevant Java classes you are using, perhaps with `import java`. But the builtin functions are really the core function that almost all Python code uses.

The documentation for covering all of the built-in functions that are available is extensive. However, it has been included in this book as Appendix C. It should be easy to use Appendix C as a reference when using a built-in function, or for choosing which built-in function to use.

Alternative Ways to Define Functions

The `'def'` keyword is not the only way to define a function. Here are some alternatives:

- Lambda Functions:

`'lambda'` functions. The `'lambda'` keyword creates an unnamed function. Some people like this because it requires minimal space, especially when used in a callback.

- Classes:

In addition, we can also create objects with classes whose instance objects look like ordinary functions. Objects supporting the `__call__` protocol. This will be covered in a later chapter. For Java developers, this is familiar. Classes implement such single-method interfaces as `Callable` or `Runnable`.

- Bound Methods:

Instead of calling `x.a()`, I can pass `x.a` as a parameter or bind to another name. Then I can invoke this name. The first parameter of the method will be passed the bound object, which in OO terms is the receiver of the method. This is a simple way of creating callbacks. (In Java you would have just passed the object of course, then having the callback invoke the appropriate method such as `call` or `run`.)

- **staticmethod, classmethod, descriptors functools, such as `for`** partial construction.
- Other function constructors, including yours?

Lambda Functions

As stated in the introduction, a lambda function is an anonymous function. In other words, a lambda function is not bound to any name. This can be useful when you are trying to create compact code or when it does not make sense to declare a named function because it will only be used once.

A lambda function is usually written inline with other code, and most often the body of a lambda function is very short in nature. A lambda function is comprised of the following segments:

```
lambda <<argument(s)>> : <<function body>>
```

A lambda function accepts arguments just like any other function, and it uses those arguments within its function body. Also, just like other functions in Python a value is always returned. Let's take a look at a simple lambda function to get a better understanding of how they work.

Example of using a lambda function to combine two strings. In this case, a first and last name

```
>>> name_combo = lambda first,last: first + ' ' + last
>>> name_combo('Jim','Baker')
'Jim Baker'
```

In the example above, we assigned the function to a name. However, a lambda function can also be defined in-line with other code. Oftentimes a lambda function is used within the context of other functions, namely built-ins.

Generator Functions

Generators are functions that construct objects implementing Python's iterator protocol.

iter() - obj.__iter__ Call obj.next

Generators advance to the next point by calling the special method `next`. Usually that's done implicitly, typically through a loop or a consuming function that accepts iterators, including generators. They return values by using the `yield` statement. Each time a `yield` statement is encountered then the current iteration halts and a value is returned. Generators have the ability to remember where they left off. Each time `next()` is called, the generator resumes where it had left off. If a generator function is not used in the context of a loop, then a `StopIteration` error will be raised once the generator has been terminated.

Over the next couple of sections, we will take a closer look at generators and how they work. Along the way, you will see many examples for creating and using generators.

Defining Generators

A generator function is written so that it consists of one or more `yield` points, which are marked through the use of the `yield` statement. As mentioned previously, each time the `yield` statement is encountered, a value is returned.

```
def g():
    print "before yield point 1"
    # The generator will return a value once it encounters the yield statement
    yield 1
    print "after 1, before 2"
    yield 2
    yield 3
```

In the example above, the generator function `g()` will halt and return a value once the first `yield` statement is encountered. In this case, a 1 will be returned. The next time `g.next()` is called, the generator will continue until it encounters the next `yield` statement. At that point it will return another value, the 2 in this case. Let's see this generator in action.

```
>>> x = g()
>>> x.next()
before the yield point 1
1
>>> x.next()
after 1, before 2
2
>>> x.next()
3
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Let's take a look at another more useful example of a generator. In the following example, the *step_to()* function is a generator that increments based upon a given factor. The generator starts at zero and increments each time *next()* is called. It will stop working once it reaches the value that is provided by the *stop* argument.

```
>>> def step_to(factor, stop):
...     step = factor
...     start = 0
...     while start <= stop:
...         yield start
...         start += step
...
>>> for x in step_to(1, 10):
...     print x
...
0
1
2
3
4
5
6
7
8
9
10
>>> for x in step_to(2, 10):
...     print x
...
0
2
4
6
8
10
>>>
```

If the *yield* statement is seen in the scope of a function, then that function is compiled as if it's a generator function. Unlike other functions, you use the *return* statement only to say, "I'm done", that is, to exit the generator. It is not necessary to explicitly *return*. You can think of *return* as acting like a *break* in a for-loop or while-loop. Let's change the *step_to* function just a bit to check and ensure that the factor is less than the stopping point. We'll add a *return* statement to exit the generator if the factor is greater or equal to the stop.

```

>>> def step_return(factor, stop):
...     step = factor
...     start = 0
...     if factor >= stop:
...         return
...     while start <= stop:
...         yield start
...         start += step
>>> for x in step_return(1,10):
...     print x
...
0
1
2
3
4
5
6
7
8
9
10
>>> for x in step_return(3,10):
...     print x
...
0
3
6
9
>>> for x in step_return(3,3):
...     print x
...

```

If you attempt to return an argument then a syntax error will be raised.

```

def g():
    yield 1
    yield 2
    return None

for i in g():
    print i

SyntaxError: 'return' with argument inside generator

```

Many useful generators actually will have an infinite loop around their yield expression, instead of ever exiting, explicitly or not. The generator will essentially work each time *next()* is called throughout the life of the program.

Pseudocode for generator using infinite loop

```

while True:
    yield stuff

```

This works because a generator object can be garbage collected, just like any other object implementing the iteration protocol. The fact that it uses the machinery of function objects to implement itself doesn't matter.

How it actually works

Generators are actually compiled differently from other functions. Each yield point saves the state of unnamed local variables (Java temporaries) into the frame object, then returns the value to the function that had called `next` (or `send` in the case of a coroutine). The generator is then indefinitely suspended, just like any other iterator. Upon calling `next` again, the generator is resumed by restoring these local variables, then executing the next bytecode instruction following the yield point. This process continues until the generator is either garbage collected or it exits.

You can determine if the underlying function is a generator if it's code object has the `CO_GENERATOR` flag set in `co_flags`.

Generators can also be resumed from any thread, although some care is necessary to ensure that underlying system state is shared (or compatible). We will explore how to use effectively use this capability in the chapter on concurrency.

Generator Expressions

Generator expressions are an alternative way to create the generator object. Please note that this is not the same as a generator function! It's the equivalent to what a generator function yields when called. Generator expressions basically create an unnamed generator.

```
>>> x = (2 * x for x in [1,2,3,4])
>>> x
<generator object at 0x1>
>>> x()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not callable
```

Let's see this generator expression in action:

```
>>> for v in x:
...     print v
...
2
4
6
8
>>>
```

Typically generator expressions tend to be more compact but less versatile than generator functions. They are useful for getting things done in a concise manner.

Namespaces, Nested Scopes and Closures

Note that you can introduce other namespaces into your function definition. It is possible to include import statements directly within the body of a function. This allows such imports to be valid only within the context of the function. For instance, in the following function definition the imports of *A* and *B* are only valid within the context of *f()*.

```
def f():
    from NS import A, B
```

At first glance, including import statements within your function definitions may seem unnecessary. However, if you think of a function as an object then it makes much more sense. We can pass functions around just like other objects in Python such as variables. As mentioned previously, functions can even be passed to other functions as arguments. Function namespaces provide the ability to treat functions as their own separate piece of code. Oftentimes, functions that are used in several different places throughout an application are stored in a separate module. The module is then imported into the program where needed.

Functions can also be nested within each other to create useful solutions. Since functions have their own namespace, any function that is defined within another function is only valid within the parent function. Let's take a look at a simple example of this before we go any further.

```
>>> def parent_function():
...     x = [0]
...     def child_function():
...         x[0] += 1
...         return x[0]
...     return child_function
...
>>> p()
1
>>> p()
2
>>> p()
3
>>> p()
4
```

While this example is not extremely useful, it allows you to understand a few of the concepts for nesting functions. As you can see, the *parent_function* contains a function named *child_function*. The *parent_function* calls the *child_function* passing an argument. What we have created in this example is a simple *Closure* function. Each time the function is called, it executes the inner function and increments the variable *x* which is only available within the scope of this closure.

In the context of Jython, using closures such as the one defined above can be useful for integrating Java concepts as well. It is possible to import Java classes into the scope of your function just as it is possible to work with other Python modules. You will learn more about using Java within Jython in Chapter 7 and Chapter 10.

Function Decorators

Decorators are a convenient syntax that describes how to transform a function. They are essentially a metaprogramming technique that enhances the action of the function that they decorate. To program a function decorator, a function that has already been defined can be used to decorate another function, which basically allows the decorated function to be passed into the function that is named in the decorator. Let's look at a simple example.

Example 4_1.py

```
def plus_five(func):
    x = func()
    return x + 5

@plus_five
def add_nums():
    return 1 + 2
```

In the given example, the *add_nums()* function is decorated with the *plus_five()* function. This has the same effect as passing the *add_nums* function into the *plus_five* function. In other words, this decorator is syntactic sugar that makes this technique easier to use. The decorator above has the same functionality as the following code.

```
add_nums = plus_five(add_nums)
```

Now that we've covered the basics of function decorators it is time to take a look at a more in-depth example of the concept. In the following decorator function example, we are taking a twist on the old `tip_calculator` function and adding a sales tax calculation. As you see, the original `calc_bill` function takes a sequence of amounts, namely the amounts for each item on the bill. The `calc_bill` function then simply sums the amounts and returns the value. In the given example, we apply the `sales_tax` decorator to the function which then transforms the function so that it not only calculates and returns the sum of all amounts on the bill, but it also applies a standard sales tax to the bill and returns the tax amount and total amounts as well.

Example 4_2.py

```
def sales_tax(func):
    ''' Applies a sales tax to a given bill calculator '''
    def calc_tax(*args, **kwargs):
        f = func(*args, **kwargs)
        tax = f * .18
        print "Total before tax: $ %.2f" % (f)
        print "Tax Amount: $ %.2f" % (tax)
        print "Total bill: $ %.2f" % (f + tax)
    return calc_tax

@sales_tax
def calc_bill(amounts):
    ''' Takes a sequence of amounts and returns sum '''
    return sum(amounts)
```

The decorator function contains an inner function that accepts two arguments, a sequence of arguments and a dictionary of keyword args. We must pass these arguments to our original function when calling from the decorator to ensure that the arguments that we passed to the original function are applied within the decorator function as well. In this case, we want to pass a sequence of amounts to `calc_bill`, so passing the `*args`, and `**kwargs` arguments to the function ensures that our amounts sequence is passed within the decorator. The decorator function then performs simple calculations for the tax and total dollar amounts and prints the results. Let's see this in action:

```
>>> amounts = [12.95,14.57,9.96]
>>> calc_bill(amounts)
Total before tax: $ 37.48
Tax Amount: $ 6.75
Total bill: $ 44.23
```

It is also possible to pass arguments to decorator functions. In order to do so, we must nest another function within our decorator function. The outer function will accept the arguments to be passed into the decorator function, the inner function will accept the decorated function, and the inner most function will perform the work. We'll take another spin on the tip calculator example and create a decorator that will apply the tip calculation to the `calc_bill` function.

```
def tip_amount(tip_pct):
    def calc_tip_wrapper(func):
        def calc_tip_impl(*args, **kwargs):
            f = func(*args, **kwargs)
            print "Total bill before tip: $ %.2f" % (f)
            print "Tip amount: $ %.2f" % (f * tip_pct)
            print "Total with tip: $ %.2f" % (f + (f * tip_pct))
        return calc_tip_impl
    return calc_tip_wrapper
```

Now let's see this decorator function in action. As you'll notice, we pass a percentage amount to the decorator itself and it is applied to the decorator function.

```
>>> @tip_amount(.18)
... def calc_bill(amounts):
...     ''' Takes a sequence of amouunts and returns sum '''
...     return sum(amounts)
...
>>> amounts = [20.95, 3.25, 10.75]
>>> calc_bill(amounts)
Total bill before tip: $ 34.95
Tip amount: $ 6.29
Total with tip: $ 41.24
```

As you can see, we have a similar result as was produced with the sales tax calculator. All of the amounts in the sequence of amounts are summed up and then the tip is applied.

Coroutines

Coroutines are often compared to generator functions in that they also make use of the *yield* statement. However, a coroutine is exactly the opposite of a generator in terms of functionality. A coroutine actually treats a *yield* statement as an expression, and it accepts data instead of returning it. Coroutines are oftentimes overlooked as they may at first seem like a daunting topic. However, once it is understood that coroutines and generators are not the same thing then the concept of how they work is a bit easier to grasp.

A coroutine is a function that receives data and does something with it. We will take a look at a simple coroutine example and then break it down to study the functionality.

```
def co_example(name):
    print 'Entering coroutine %s' % (name)
    my_text = []
    while True:
        txt = (yield)
        my_text.append(txt)
        print my_text
```

Here we have a very simplistic coroutine example. It accepts a value as the “name” of the coroutine. It then accepts strings of text, and each time a string of text is sent to the coroutine, it is appended to a list. The *yield* statement is where text is being entered by the user. It is assigned to the *txt* variable and then processing continues. Let’s take a look at how to actually use the coroutine.

```
>>> ex = co_example("example1")
>>> ex.next()
Entering coroutine example1
```

In the code above, we assign the name “example1” to this coroutine. We could actually accept any type of argument for the coroutine and do whatever we want with it. We’ll see a better example after we understand how this works. The next line of code calls *next()* on the function. The *next()* must be called once to initialize the coroutine. Once this has been done, the function is ready to accept values.

```
>>> ex.send("test1")
['test1']
>>> ex.send("test2")
['test1', 'test2']
>>> ex.send("test3")
['test1', 'test2', 'test3']
```

As you can see, we use the *send()* method to actually send data values into the coroutine. In the function itself, the text we *send* is inserted where the (*yield*) expression is placed. We can really continue to use the coroutine forever, or

until our JVM is out of memory. However, it is a best practice to *close()* the coroutine once it is no longer needed. The *close()* call will cause the coroutine to be garbage collected.

```
>>> ex.close()
>>> ex.send("test1")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

If we try to send more data to the function once it has been closed then a *StopIteration* error is raised. Coroutines can be very helpful in a number of situations. While the example above doesn't do much, there are a number of great applications to which we can apply the use of coroutines.

Decorators in Coroutines

While the initialization of a coroutine by calling the *next()* method is not difficult to do, we can eliminate this step to help make things even easier. By applying a decorator function to our coroutine, we can automatically initialize it so it is ready to receive data.

Let's define a decorator that we can apply to the coroutine in order to make the call to *next()*.

```
def coroutine_next(f):
    def initialize(*args,**kwargs):
        coroutine = f(*args,**kwargs)
        coroutine.next()
        return coroutine
    return initialize
```

Now we will apply our decorator to the coroutine function and then make use of it.

```
>>> @coroutine_next
... def co_example(name):
...     print 'Entering coroutine %s' % (name)
...     my_text = []
...     while True:
...         txt = (yield)
...         my_text.append(txt)
...         print my_text
...
>>> ex2 = co_example("example2")
Entering coroutine example2
>>> ex2.send("one")
['one']
>>> ex2.send("two")
['one', 'two']
>>> ex2.close()
```

As you can see, while it is not necessary to use a decorator for performing such tasks, it definitely makes things easier to use.

Coroutine Example

Now that we understand how coroutines are used, let's take a look at a more in-depth example. Hopefully after reviewing this example you will understand how useful such functionality can be.

In this example, we will pass the name of a file to the coroutine on initialization. After that, we will send strings of text to the function and it will open the text file that we sent to it (given that the file resides in the correct location), and

search for the number of matches per a given word. The numeric result for the number of matches will be returned to the user.

Example-4_3.py

```
def search_file(filename):
    print 'Searching file %s' % (filename)
    my_file = open(filename, 'r')
    file_content = my_file.read()
    my_file.close()
    while True:
        search_text = (yield)
        search_result = 0
        search_result = file_content.count(search_text)
        print 'Number of matches: %d' % (search_result)
```

The coroutine above opens the given file, reads it's content, and then searches and returns the number of matches for any given *send* call.

```
>>> search = search_file("example4_3.txt")
>>> search.next()
Searching file example4_3.txt
>>> search.send('python')
Number of matches: 0
>>> search.send('Jython')
Number of matches: 1
>>> search.send('the')
Number of matches: 4
>>> search.send('This')
Number of matches: 2
>>> search.close();
```

Conclusion

In this chapter, we have covered the use of functions in the Python language. There are many different use-cases for functions and we have learned techniques that will allow us to apply the functions to many situations. Functions are first-class objects in Python, and they can be treated as any other object. We started this chapter by learning the basics of how to define a function. After learning about the basics, we began to evolve our knowledge of functions by learning how to use parameters and make recursive function calls.

There are a wide variety of built-in functions available for use. If you take a look at Appendix C of this book you can see a listing of these built-ins. It is a good idea to become familiar with what built-ins are available for you. After all, it doesn't make much sense to re-write something that has already been done.

This chapter also discussed some alternative ways to define functions including the lambda notation, as well as some alternative types of functions including decorators, generators and coroutines. Wrapping up this chapter, you should now be familiar with Python functions and how to create and use them. You should also be familiar with some of the advanced techniques that can be applied to functions.

In the next chapter, you will learn a bit about input and output with Jython and the basics of Python I/O. Later in Part I of this book, we will build upon object-orientation and learn how to use classes in Python.

Chapter 9: Input and Output

A program means very little if it does not take input of some kind from the program user. Likewise, if there is no form of output from a program then one may ask why we have a program at all. Input and output operations can define the user experience and usability of any program. This chapter is all about how to put information or data into a program, and then how to display it or save it to a file. This chapter does not discuss working with databases, but rather, working at a more rudimentary level with files. Throughout this chapter you will learn such techniques as how to input data for a program via a terminal or command line, likewise, you will learn how to read input from a file and write to a file. After reading this chapter, you should know how to persist Python objects to disk using the *pickle* module and also how to retrieve objects from disk and use them. There will be a broad variety of topics discussed in this chapter, all of them relating to input and output.

Input from the Keyboard

As stated in the preface to this chapter, almost every program takes input from a user in one form or another. Most basic applications allow for keyboard entry via a terminal or command line environment. Python makes keyboard input easy, and as with many other techniques in Python there are more than one way to enable keyboard input. In this section, we'll cover each of those different ways to perform this task, along with a couple of use-cases. In the end you should be able to identify the most suitable application of this feature for your needs.

`sys.stdin` and `raw_input`

Making use of `sys.stdin` is by far the most widely used method to read input from the command line or terminal. This procedure consists of importing the `sys` package, then writing a message prompting the user for some input, and lastly reading the input by making a call to `sys.stdin.readline()` and assigning the returned value to a variable. The process looks like the code that is displayed below.

```
# Obtain a value from the command line and store it into a variable
>>> import sys
>>> sys.stdout.write("Enter your favorite team: ")
>>> fav_team = sys.stdin.readline()
>>> print fav_team
Enter your favorite team: Cubs
```

You can see that the usage of `sys` modules is quite simplistic. However, another approach to performing this same task is to make use of the `raw_input` function. This function uses a more simplistic syntax in order to perform the same procedure. It basically generates some text on the command line or terminal, accepts user input, and assigns it to a variable. Let's take a look at the same example from above using the `raw_input` syntax.

```
# Obtain a value using raw_input and store it into a variable
>>> fav_team = raw_input("Enter your favorite team: ")
Enter your favorite team: Cubs
```

Obtaining Variables from Jython Environment

It is possible to retrieve values directly from the Jython environment for use within your applications. For instance, we can obtain system environment variables or use the variables that are assigned to `sys.argv` at runtime.

To use environment variable values within your Jython application, simply import the `os` module and use its `environ` dictionary to access them. Since this is a dictionary object, you can obtain a listing of all environment variables by simply typing `os.environ` at will.

```
>>> import os
>>> home = os.environ["HOME"]
>>> home
'/Users/juneau'

# Change home directory for the Python session
>>> os.environ["HOME"] = "/newhome"
>>> home = os.environ["HOME"]
>>> home
'/newhome'
```

When you are executing a Jython module from the command prompt or terminal, you can make use of the `sys.argv` list that takes values from the command prompt or terminal after invoking the Jython module. For instance, if we are interested in having our program user enter some text to be used by the module, they can simply invoke the module and then use type all of the text entries followed by spaces, using quotes if you wish to pass an argument that contains a space. The number of arguments can be any size (I've never hit an upper bound anyways), so the possibilities are endless.

```
# sysargv_print.py - Prints all of the arguments provided at the command line
import sys
    for sysargs in sys.argv:
        print sysargs

# Usage
>>> jython sysargv_print.py "test" "test2" "test3"
sysargv_print.py
test
test2
test3
```

As you can see, the first entry in `sys.argv` is the script name, and then each additional argument provided after the module name is then added to the `sys.argv` list. This is quite useful for creating scripts to use for automating tasks, etc.

File I/O

We learned a bit about the *File* data type in chapter 2 of this book. In that chapter, we briefly discussed a few of the operations that can be performed using this type. In this section, we will go into detail on what we can do with a *File* object. We'll start with the basics, and move into more detail. To begin, you should take a look at the table below that lists all of the methods available to a *File* object and what they do.

Method	Description
<code>close()</code>	Close file
<code>fileno()</code>	Returns integer file descriptor
<code>flush()</code>	Used to flush the output buffers
<code>isatty()</code>	If the file is an interactive terminal, returns 1
<code>next()</code>	Returns the next line in the file. If no line is found, returns <code>StopIteration</code>
<code>read(x)</code>	Reads x bytes
<code>readline(x)</code>	Reads single line up to x characters, or entire line if x is omitted
<code>readlines(size)</code>	Reads all lines in file into a list. If <i>size</i> > 0, reads that number of characters
<code>seek()</code>	Moves cursor to a new position in the file
<code>tell()</code>	Returns the current position of the cursor
<code>truncate()</code>	Truncates a file
<code>write(string)</code>	Writes a string
<code>writelines(seq)</code>	Writes all strings contained in a sequence

Table 9-1: File Object Methods

We'll start by creating a file for use. As discussed in chapter 2, the `open(filename[, mode])` built-in function creates and opens a specified file in a particular manner. The `mode` specifies what mode we will open the file into, be it read, read-write, etc.

```
>>> myFile = open('mynewfile.txt', 'w')
>>> firstString = "This is the first line of text."
>>> myFile.write(firstString)
>>> myFile.close()
```

In the example above, the file “mynewfile.txt” did not exist until the `open` function was called. The file was created in *write* mode and then we do just that, write a string to the file. Now, it is important to make mention that the *firstString* is not actually written to the file until it is closed or *flush()* *is performed*. *It is also worth mentioning that if we were to perform a subsequent *write() operation on the file then the first contents of the file would be overwritten by the subsequent contents.*

Now we'll step through each of the file functions in an example. The main focus of this example is to provide you with a place to look for actual working file I/O code.

```
# Write lines to file, flush, and close
>>> myFile.write('This is the first line of text.')
>>> myFile.write('This is the second line of text.')
>>> myFile.write('This is the last line of text.')
>>> myFile.flush()
>>> myFile.close()

# Open file in read mode
>>> myFile = open('mynewfile.txt', 'r')
>>> myFile.read()
'My second line of text.This is the first line of text.This is the second line of
↳text.This is the last line of text.'

# If we read again, we get a '' because cursor is at the end of text
>>> myFile.read()
''

# Seek back to the beginning of file and perform read again
>>> myFile.seek(0)
>>> myFile.read()
'My second line of text.This is the first line of text.This is the second line of
↳text.This is the last line of text.'

# Seek back to beginning of file and perform readline()
>>> myFile.seek(0)
>>> myFile.readline()
'This is the first line of text.This is the second line of text.This is the last line
↳of text.'

# Use tell() to display current cursor position
>>> myFile.tell()
93L
>>> myFile.seek(0)
>>> myFile.tell()
0L

# Loop through lines of file
>>> for line in myFile:
...     print line
```

```
...
This is the first line of text.This is the second line of text.This is the last line_
↳of text.
```

There are a handful of read-only attributes that we can use to find out more information about file objects. For instance, if we are working with a file and want to see if it is still open or if it has been closed, we could view the *closed* attribute on the file to return a boolean stating whether the file is closed. The following table lists each of these attributes and what they tell us about a file object.

Attribute	Description
closed	Returns a boolean to indicate if the file is closed
encoding	Returns a string indicating encoding on file
mode	Returns the I/O mode for a file
name	Returns the name of the file
newlines	Returns the newline representation in the file

File Attributes

```
>>> myFile.closed
False
>>> myFile.mode
'r'
>>> myFile.name
'mynewFile.txt'
```

Pickle

One of the most popular modules in the Python language is the *pickle* module. The goal of this module is basically to allow for the serialization and persistence of Python objects to disk in file format. A *pickled* object can be written to disk using this module, and it can also be read back in and utilized in object format. Just about any Python object can be persisted using *pickle*.

To write an object to disk, we call the *pickle()* function. The object will be written to file in a format that may be unusable by anything else, but we can then read that file back into our program and use the object as it was prior to writing it out. In the following example, we'll create a *Player* object and then persist it to file using *pickle*. Later, we will read it back into a program and make use of it. We will make use of the *File* object when working with the *pickle* module.

```
>>> import pickle
>>> class Player(object):
...     def __init__(self, first, last, position):
...         self.first = first
...         self.last = last
...         self.position = position
...
>>> player = Player('Josh', 'Juneau', 'Forward')
>>> pickleFile = open('myPlayer', 'wb')
>>> pickle.dump(player, pickleFile)
>>> pickleFile.close()
```

In the example above, we've persisted a *Player* object to disk using the *dump(object, file)* method in the *pickle* module. Now let's read the object back into our program and print it out.

```
>>> pickleFile = open('myPlayer', 'rb')
>>> player1 = pickle.load(pickleFile)
>>> pickleFile.close()
```

```
>>> player1.first
'Josh'
>>> player1.last, player1.position
('Juneau', 'Forward')
```

Similarly, we read the pickled file back into our program using the `load(file)` method. Once read and stored into a variable, we can close the file and work with the object. If we had to perform a sequence of dump or load tasks, we could do so one after the other without issue. You should also be aware that there are different pickle protocols that can be used in order to make pickle work in different Python environments. The default protocol is 0, but protocols 1 and 2 are also available for use. It is best to stick with the default as it works well in most situations, but if you run into any trouble using pickle with binary formats then please give the others a try.

If we had to store objects to disk and reference them at a later time, it may make sense to use the `shelve` module which acts like a dictionary for pickled objects. With the `shelve` technique, you basically pickle an object and store it using a string-based key value. You can later retrieve the object by passing the key to the opened file object. This technique is very similar to a filing cabinet for our objects in that we can always reference our objects by key value. Let's take a look at this technique and see how it works.

```
# Store different player objects
>>> import shelve
>>> player1 = Player('Josh', 'Juneau', 'forward')
>>> player2 = Player('Jim', 'Baker', 'defense')
>>> player3 = Player('Frank', 'Wierzbicki', 'forward')
>>> player4 = Player('Leo', 'Soto', 'defense')
>>> player5 = Player('Vic', 'Ng', 'center')
>>> data = shelve.open("players")
>>> data['player1'] = player1
>>> data['player2'] = player2
>>> data['player3'] = player3
>>> data['player4'] = player4
>>> data['player5'] = player5
>>> playerTemp = data['player3']
>>> playerTemp.first, playerTemp.last, playerTemp.position
('Frank', 'Wierzbicki', 'forward')
>>> data.close()
```

In the scenario above, we used the same *Player* object that was defined in the previous examples. We then opened a new *shelve* and named it “players”, this *shelve* actually consists of a set of three files that are written to disk. These three files can be found on disk named “players.bak”, “players.dat”, and “players.dir” once the objects were persisted into the *shelve* and it was closed. As you can see, all of the *Player* objects we’ve instantiated have all been stored into this *shelve* unit, but they exist under different keys. We could have named the keys however we wished, as long as they were each unique. In the example, we persist five objects and then at the end one of the objects is retrieved and displayed. This is quite a nice technique to make a small data store.

Output Techniques

We basically covered the *print* statement in chapter 2 very briefly when discussing string formatting. The *print* statement is by far the most utilized form of output in most Python programs. Although we covered some basics such as conversion types and how to format a line of output in chapter 2, here we will go into a bit more depth on some different variations of the *print* statement as well as other techniques for generating output. There are basically two formats that can be used with the *print* statement. We covered the first in chapter two, and it makes use of a string and some conversion types embedded within the string and preceded by a percent (%) symbol. After the string, we use another percent(%) symbol followed by a parenthesized list of arguments that will be substituted in place of the embedded conversion types in our string in order. We can also use a comma instead of a percent symbol in order to

achieve the same effect. It is merely a matter of preference. Check out the examples of each depicted in the example below.

```
# Using the % symbol
>>> x = 5
>>> y = 10
>>> print 'The sum of %d and %d is %d' % (x, y, (x + y))
The sum of 5 and 10 is 15

>>> adjective = "awesome"
>>> print 'Jython programming is %s' % (adjective)
Jython programming is awesome

# Using a comma
>>> print y, " divided by ", x, " is ", y/5
10 divided by 5 is 2
```

You can also format floating-point output using the conversion types that are embedded in your string. You may specify a number of decimal places you'd like to print by using a “.# of places” syntax in the embedded conversion type.

```
>>> pi = 3.14
>>> print 'Here is some formatted floating point arithmetic: %.2f' % (pi + y)
Here is some formatted floating point arithmetic: 13.14
>>> print 'Here is some formatted floating point arithmetic: %.3f' % (pi + y)
Here is some formatted floating point arithmetic: 13.140
```

If we were working with a list or a range of numbers, we could use a generator to help us with output. It works like as follows: create a generator function that “prints” some output using the *yield* statement. Assign the returned value(s) from the generator to a variable, then print the variable to see the outcome.

```
>>> def writeX(upper, lower):
...     x = lower
...     y = upper
...     while x < y:
...         yield 'The value of x is: %d' % (x+1)
...         x = x + 1
...
>>> out = "".join(writeX(10,5))
>>> print out
The value of x is: 6The value of x is: 7The value of x is: 8The value of x is: 9The_
↪value of x is: 10
```

Conclusion

It goes without saying that Python has its share of input and output strategies. This chapter covered most of those techniques starting with basic terminal or command line I/O and then onto file manipulation. We learned how to make use of the *open* function for creating, reading, or writing a file. The command line *sys.argv* arguments are another way that we can grab input, and environment variables can also be used from within our programs. Following those topics, we took a brief look at the *pickle* module and how it can be used to persist Python objects to disk. *shelve* is another twist on using *pickle* that allows for multiple objects to be indexed and stored within the same file. Finally, we discussed a couple of techniques for performing output in our programs.

Although there are some details that were left out as I/O could consume an entire book, this chapter was a solid starting point into the broad topic of I/O in Python. As with much of the Python language specifics discussed in this book,

there are many resources available on the web and in book format that will help you delve deeper into the topics if you wish.

In the next chapter, we will discuss using Jython and Java together. This topic is at the heart of Jython, it is one of the main reasons why Python was implemented in Java.

Chapter 5: Object Oriented Jython

This chapter is going to cover the basics of object oriented programming. If you're familiar with the concepts. I'll start with covering the basic reasons for why you would want to write object oriented code in the first place, then cover all the basic syntax, and finally I'll show you a non-trivial example.

Object oriented programming is a method of programming where you package your code up into bundles of data and behaviour. In Jython, you can define a template for this bundle with a class definition. With this first class written, you can then instantiate copies of your object and have them act upon each other. This helps you organize your code into smaller more manageable bundles.

Throughout this chapter, I interchangeably use Python and Jython - for regular object oriented programming - the two dialects of Python are so similar that there are no meaningful differences between the two languages. Enough introduction text though - let's take a look at some basic syntax to see what this is all about.

Basic Syntax

Writing a class is really simple. It is fundamentally about managing some kind of 'state' and exposing some functions to manipulate that state. In object jargon - we just call those functions 'methods'.

Let's start by creating a car class. The goal is to create an object that will manage it's own location on a two dimensional plane. We want to be able to tell it to turn, move forward, and we want to be able to interrogate the object to find out where it's current location is.

```
class Car(object):

    NORTH = 0
    EAST = 1
    SOUTH = 2
    WEST = 3

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        self.direction = 0

    def turn_right(self):
        self.direction += 1
        self.direction = self.direction % 4

    def turn_left(self):
        self.direction -= 1
        self.direction = self.direction % 4

    def move(self, distance):
        if self.direction == self.NORTH:
            self.y += distance
        elif self.direction == self.SOUTH:
            self.y -= distance
        elif self.direction == self.EAST:
```



```

        self.x += distance
    else:
        self.x -= distance

    def position(self):
        return (self.x, self.y)

```

We'll go over that class definition in detail but right now, let's just see how to create a car, move it around and ask the car where it is.

```

from car import Car

def test_car():
    c = Car()
    c.turn_right()
    c.move(5)
    assert (5, 0) == c.position()

    c.turn_left()
    c.move(3)

    assert (5, 3) == c.position()

if __name__ == '__main__':
    test_car()

```

The best way to think of a class is to think of it like a special kind of function that acts like a factory that generates object instances. For each call to the class - you are creating a new discrete copy of your object.

Once we've created the car instance, we can simply call functions that are attached to the car class and the object will manage it's own location. From the point of view of our test code - we do not need to manage the location of the car - nor do we need to manage the direction that the car is pointing in. We just tell it to move - and it does the right thing.

Let's go over the syntax in detail to see exactly what's going on here.

In Line 1, we declare that our Car object is a subclass of the root "object" class. Python, like many object oriented languages has a 'root' object that all other objects are based off of. This 'object' class defines basic behavior that all classes can reuse.

Python actually has two kinds of classes - 'newstyle' and old style. The old way of declaring classes didn't require you to type 'object' - you'll occasionally see the old-style class usage in some Python code, but it's not consider a good practice Just subclass 'object' for any of your base classes and your life will be simpler[1].

Lines 3 to 6 declare class attributes for the direction that any car can point to. These are *class* attributes so they can be shared across all object instances of the car object.

Now for the good stuff.

Line 8-11 declares the object initializer. In some languages, you might be familiar with a constructor - in Jython, we have an initializer which lets us pass values into an object at the time of creation.

In our initializer, we are setting the initial position of the car to (0, 0) on a 2 dimensional plane and then the direction of the car is initialized to pointing north. Fairly straight forward so far.

The function signature uses Python's default argument list feature so we don't have to explicitly set the initial location to (0,0), but there's a new argument introduced called 'self'. This is a reference to the current object.

Remember - your class definition is creating instances of objects. Once your object is created, it has it's own set of internal variables to manage. Your object will inevitably need to access these as well as any of the classes internal methods. Python will pass a reference to the current object as the first argument to all your instance methods.

If you're coming from some other object oriented language, you're probably familiar with the 'this' variable. Unlike C++ or Java, Python doesn't magically introduce the reference into the namespace of accessible variables, but this is consistent with Python's philosophy of making things explicit for clarity.

When we want to assign the initial x,y position, we just need to assign values on to the name 'x', and 'y' on the object. Binding the values of x and y to self makes the position values accessible to any code that has access to self - namely the other methods of the object. One minor detail here - in Python, you can technically name the arguments however you want. There's nothing stopping you from calling the first argument 'this' instead of 'self', but the community standard is to use 'self'².

Line 13 to 19 declare two methods to turn the vehicle in different directions. Notice how the direction is never directly manipulated by the caller of the Car object. We just asked the car to turn, and the car changed it's own internal 'direction' state.

Line 21 to 29 define where the car should move to when we move the car forward. The internal direction variable informs the car how it should manipulate the x and y position. Notice how the caller of the car object never needs to know precisely what direction the car is pointing in. The caller only needs to tell the object to turn and move forward. The particular details of how that message is used is abstracted away.

That's not too bad for a couple dozen lines of code.

This concept of hiding internal details is called encapsulation. This is a core concept in object oriented programming. As you can see from even this simple example - it allows you to structure your code so that you can provide a simplified interface to the users of your code.

Having a simplified interface means that we could have all kinds of behaviour happening behind the function calls to turn and move - but the caller can ignore all those details and concentrate on *using* the car instead of managing the car.

As long as the method signatures don't change, the caller really doesn't need to care about any of that. We can easily add persistence to this class - so we can save and load the car's state to disk.

First, pull in the pickle module - pickle will let us convert python objects into byte strings that can be restored to full objects later.

```
import pickle
```

Now, just add two new methods to load and save the state of the object.

```
def save(self):
    state = (self.direction, self.x, self.y)
    pickle.dump(state, open('mycar.pickle', 'wb'))

def load(self):
    state = pickle.load(open('mycar.pickle', 'rb'))
    (self.direction, self.x, self.y) = state
```

Simply add calls to save() at the end of the turn and move methods, and the object will automatically save all the relevant internal values to disk.

People who use the car object don't even need to know that it's saving to disk, because the Car object handles it behind the scenes.

```
def turn_right(self):
    self.direction += 1
    self.direction = self.direction % 4
    self.save()

def turn_left(self):
    self.direction -= 1
```

² One of Python's strengths is legibility - of your code and other code. Community standards help the legibility of code tremendously.

```

        self.direction = self.direction % 4
        self.save()

def move(self, distance):
    if self.direction == self.NORTH:
        self.y += distance
    elif self.direction == self.SOUTH:
        self.y -= distance
    elif self.direction == self.EAST:
        self.x += distance
    else:
        self.x -= distance
    self.save()

```

Now, when you call the turn, or move methods, the car will automatically save itself to disk. If you want to reconstruct the car object's state from a previously saved pickle file, you can simply call the load() method.

Object Attribute Lookups

If you've been paying attention, you're probably wondering how the NORTH, SOUTH, EAST and WEST variables got bound to self. We never actually assigned them to the self variable during object initialization - so what's going on when we call move()? How is Jython actually resolving the value of those four variables?

Now seems like a good time to show how Jython resolves name lookups.

The direction names actually got bound to the Car class. The Jython object system does a little bit of magic when you try accessing any *name* against an object, it first searches for anything that was bound to 'self'. If python can't resolve any attribute on self with that name, it goes up the object graph to the class definition. The direction attributes NORTH, SOUTH, EAST, WEST were bound to the class definition - so the name resolution succeeds and we get the value of the class attribute.

An very short example will help clarify this

```

>>> class Foobar(object):
...     def __init__(self):
...         self.somevar = 42
...         class_attr = 99
...
>>>
>>> obj = Foobar()
>>> obj.somevar
42
>>> obj.class_attr
99
>>> obj.not_there
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Foobar' object has no attribute 'not_there'
>>>

```

So the key difference here is *what* you bind a value to. The values you bind to self are available only to a single object. Values you bind to the class definition are available to all instances of the class. The sharing of class attributes among all instances is a critical distinction because mutating a class attribute will affect all instances. This may cause unintended side effects if you're not paying attention as a variable may change value on you when you aren't expecting it to.

```
>>> other = Foobar()
>>> other.somevar
42
>>> other.class_attr
99
>>> # obj and other will have different values for somevar
>>> obj.somevar = 77
>>> obj.somevar
77
>>> other.somevar
42
>>> # Now show that we have the same copy of class_attr
>>> other.class_attr = 66
>>> other.class_attr
66
>>> obj.class_attr
66
```

I think it's important to stress just how transparent Python's object system really is. Object attributes are just stored in a plain python dictionary. You can directly access this dictionary by looking at the `__dict__` attribute.

```
>>> obj = Foobar()
>>> obj.__dict__
{'somevar': 42}
```

Notice that there are no references to the methods of the class, or the class attribute. I'll reiterate it again - Python is going to just go up your inheritance graph - and go to the class definition to look for the methods of Foobar and the class attributes of foobar.

The same trick can be used to inspect all the attributes of the class, just look into the `__dict__` attribute of the class definition and you'll find your class attributes and all the methods that are attached to your class definition

```
>>> Foobar.__dict__
{'__module__': '__main__',
 'class_attr': 99,
 '__dict__': <attribute '__dict__' of 'Foobar' objects>,
 '__init__': <function __init__ at 1>}
```

This transparency can be leveraged with dynamic programming techniques using closures and binding new functions into your class definition at runtime. We'll revisit this later in the chapter when we look at generating function dynamically and finally with a short introduction to metaprogramming.

Inheritance and Overloading

In the car example, we subclass from the root object type. You can also subclass your own classes to specialize the behaviour of your objects. You may want to do this if you notice that your code naturally has a structure where you have many different classes that all share some common behaviour.

With objects, you can write one class, and then reuse it using inheritance to automatically gain access to the pre-existing behavior and attributes of the parent class. Your 'base' objects will inherit behaviour from the root 'object' class, but any subsequent subclasses will inherit from your own classes.

Let's take a simple example of using some animal classes to see how this works. Define a module "animals.py" with the following code:

```
class Animal(object):
    def sound(self): return "I don't make any sounds"
```

```

class Goat(Animal):
    def sound(self): return "Bleeattt!"

class Rabbit(Animal):
    def jump(self): return "hippity hop hippity hop"

class Jackalope(Goat, Rabbit): pass

```

Now you should be able to explore that module with the jython interpreter:

```

>>> from animals import *
>>> animal = Animal()
>>> goat = Goat()
>>> rabbit = Rabbit()
>>> jack = Jackalope()

```

```

>>> animal.sound()
"I don't make any sounds"
>>> animal.jump()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Animal' object has no attribute 'jump'

```

```

>>> rabbit.sound()
"I don't make any sounds"
>>> rabbit.jump()
'hippity hop hippity hop'

```

```

>>> goat.sound()
'Bleeattt!'
>>> goat.jump()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Goat' object has no attribute 'jump'

```

```

>>> jack.jump()
'hippity hop hippity hop'
>>> jack.sound()
'Bleeattt!'

```

Inheritance is a very simple concept, when you declare your class, you simply specify which parent classes you would like to reuse. Your new class can then automatically access all the methods and attributes of the super class. Notice how the goat couldn't jump and the rabbit couldn't make any sound, but the Jackalope had access to methods from both the rabbit and the goat.

With single inheritance - when your class simply inherits from one parent class - the rules for resolving where to find an attribute or a method are very straight forward. Jython just looks up to the parent if the current object doesn't have a matching attribute.

It's important to point out now that the Rabbit class is a type of Animal - the Python runtime can tell you that programmatically by using the `isinstance` function

```

>>> isinstance(bunny, Rabbit)
True
>>> isinstance(bunny, Animal)
True

```

```
>>> isinstance(bunny, Goat)
False
```

For many classes, you may want to extend the behavior of the parent class instead of just completely overriding it. For this, you'll want to use the `super()`. Let's specialize the `Rabbit` class like this.

```
class EasterBunny(Rabbit):
    def sound(self):
        orig = super(EasterBunny, self).sound()
        return "%s - but I have eggs!" % orig
```

If you now try making this rabbit speak, it will extend the original `sound()` method from the base `Rabbit` class

```
>>> bunny = EasterBunny()
>>> bunny.sound()
"I don't make any sounds - but I have eggs!"
```

That wasn't so bad. For these examples, I only demonstrated that inherited methods can be invoked, but you can do exactly the same thing with attributes that are bound to the self.

For multiple inheritance, things get very tricky. In fact, the rules for resolving how attributes are looked up would easily fill an entire chapter (look up "The Python 2.3 Method Resolution Order" on Google if you don't believe me). There's not enough space in this chapter to properly cover the topic which should be a good indication to you that you really don't want to use multiple inheritance.

More advanced abstraction

Abstraction using plain classes is wonderful and all, but it's even better if your code seems to naturally fit into the syntax of the language. Python supports a variety of underscore methods - methods that start and end with double "_" signs that let you overload the behaviour of your objects. This means that your objects will seem to integrate more tightly with the language itself.

With the underscore methods, you can give your objects behaviour for logical and mathematical operations. You can even make your objects behave more like standard builtin types like lists, sets or dictionaries.

```
from __future__ import with_statement from contextlib import closing

with closing(open('simplefile','w')) as fout: fout.writelines(["blah"])

with closing(open('simplefile','r')) as fin: print fin.readlines()
```

The above snippet of code just opens a file, writes a little bit of text and then we just read the contents out. Not terribly exciting. Most objects in Python are serializable to strings using the `pickle` module. We can leverage `pickle` to write out full blown objects to disk. Let's see the functional version of this:

```
from __future__ import with_statement
from contextlib import closing
from pickle import dumps, loads

def write_object(fout, obj):
    data = dumps(obj)
    fout.write("%020d" % len(data))
    fout.write(data)

def read_object(fin):
    length = int(fin.read(20))
    obj = loads(fin.read(length))
    return obj
```

```

class Simple(object):
    def __init__(self, value):
        self.value = value
    def __unicode__(self):
        return "Simple[%s]" % self.value

with closing(open('simplefile', 'wb')) as fout:
    for i in range(10):
        obj = Simple(i)
        write_object(fout, obj)

print "Loading objects from disk!"
print '=' * 20

with closing(open('simplefile', 'rb')) as fin:
    for i in range(10):
        print read_object(fin)

```

This should output something like this:

```

Loading objects from disk!
=====
Simple[0]
Simple[1]
Simple[2]
Simple[3]
Simple[4]
Simple[5]
Simple[6]
Simple[7]
Simple[8]
Simple[9]

```

So now we're doing something interesting. Let's look at exactly what happening here.

First, you'll notice that the Simple object is rendering a nice - the Simple object can render itself using the `__unicode__` method. This is clearly an improvement over the earlier rendering of the object with angle brackets and a hex code.

The `write_object` function is fairly straight forward, we're just converting our objects into strings using the pickle module, computing the length of the string and then writing the length and the actual serialized object to disk.

This is fine, but the read side is a bit clunky. We don't really know when to stop reading. We can fix this using the iteration protocol. Which bring us to one of my favourite reasons to use objects at all in Python.

Protocols

In Python, we have 'duck typing'. If it sounds like a duck, quacks like a duck and looks like a duck - well - it's a duck. This is in stark contrast to more rigid languages like C# or Java which have formal interface definitions. One of the nice benefits of having duck typing is that Python has the notion of object 'protocols'.

If you happen to implement the right methods - python will recognize your object as a certain type of 'thing'.

Iterators are objects that look like lists that let you read the next object. Implementing an iterator protocol is straight forward - just implement a `next()` method and a `__iter__` method and you're ready to rock and roll. Let's see this in action:

```
class PickleStream(object):
    """
    This stream can be used to stream objects off of a raw file stream
    """
    def __init__(self, file):
        self.file = file

    def write(self, obj):
        data = dumps(obj)
        length = len(data)
        self.file.write("%020d" % length)
        self.file.write(data)

    def __iter__(self):
        return self

    def next(self):
        data = self.file.read(20)
        if len(data) == 0:
            raise StopIteration
        length = int(data)
        return loads(self.file.read(length))

    def close(self):
        self.file.close()
```

The above class will let you wrap a simple file object and you can now send it raw python objects to write to a file, or you can read objects out as if the stream was just a list of objects. Writing and reading becomes much simpler

```
with closing(PickleStream(open('simplefile', 'wb'))) as stream:
    for i in range(10):
        obj = Simple(i)
        stream.write(obj)

with closing(PickleStream(open('simplefile', 'rb'))) as stream:
    for obj in stream:
        print obj
```

Abstracting out the details of serialization into the PickleStream lets us ‘forget’ about the details of how we are writing to disk. All we care about is that the object will do the right thing when we call the write() method.

The iteration protocol can be used for much more advanced uses, but even with this example, it should be obvious how useful it is. While you could implement the reading behaviour with a read() method, just using the stream as something you can loop over makes the code much easier to understand.

An aside a common problem that everyone seems to have

One particular snag that seems to catch every python programmer is when you use default values in a method signature.

```
>>> class Tricky(object):
...     def mutate(self, x=[]):
...         x.append(1)
...         return x
...
>>> obj = Tricky()
>>> obj.mutate()
[1]
```



```
>>> obj.mutate()
[1, 1]
>>> obj.mutate()
[1, 1, 1]
```

What's happening here is that the instance method 'mutate' is an object. The method object stores the default value for 'x' in an attribute *inside* the method object. So when you go and mutate the list, you're actually changing the value of an attribute of the method itself. Remember - this happens because when you invoke the mutate method, you're just accessing a callable attribute on the Tricky object.

Runtime binding of methods

One interesting feature in Python is that instance methods are actually just attributes hanging off of the class definition - the functions are just attributes like any other variable, except that they happen to be 'callable'.

It's even possible to create and bind in functions to a class definition at runtime using the new module to create instance methods. In the following example, you can see that it's possible to define a class with nothing in it, and then bind methods to the class definition at runtime.

```
>>> def some_func(self, x, y):
...     print "I'm in object: %s" % self
...     return x * y
...
>>> import new
>>> class Foo(object): pass
...
>>> f = Foo()
>>> f
<__main__.Foo object at 0x1>
>>> Foo.mymethod = new.instancemethod(some_func, f, Foo)
>>> f.mymethod(6,3)
I'm in object: <__main__.Foo object at 0x1>
18
```

When you invoke the 'mymethod' method, the same attribute lookup machinery is being invoked. Python looks up the name against the 'self' object. When it can't find anything there, it goes to the class definition. When it finds it there, the instancemethod object is returned. The function is then called with two arguments and you get to see the final result.

This kind of dynamism in Jython is extremely powerful. You can write code that generates functions at program runtime and then bind those functions to objects. You can do all of this because in Jython, classes are what are known as 'first class objects'. The class definition itself is an actual object - just like any other object. Manipulating classes is as easy as manipulating any other object.

Closures and Passing Objects

Python supports the notion of nested scopes - this can be used by to preserve some state information inside of another function. This technique isn't all that common outside of dynamic languages, so you may have never seen this before. Let's look at a simple example

```
def adder(x):
    def inner(y):
        return x + y
    return inner
```

```
>>> func = adder(5)
>>> func
<function inner at 0x7adf0>
>>> func(8)
13
```

This is pretty cool - we can actually create functions from templates of other functions. If you can think of a way to parameterize the behavior of a function, it becomes possible to create new functions dynamically. You can think of currying as yet another way of creating templates - this time you are creating a template for new functions.

This is a tremendously powerful tool once you gain some experience with it. Remember - everything in python is an object - even functions are first class objects in Python so you can pass those in as arguments as well. A practical use of this is to partially construct new functions from ‘base’ functions with some basic known behavior.

Let’s take the previous adder closure and convert it to a more general form

```
def arith(math_func, x):
    def inner(y):
        return math_func(x, y)
    return inner

def adder(x, y):
    return x + y

>>> func = arith(adder, 91)
>>> func(5)
96
```

This technique is called currying - you’re now creating new function objects based on previous functions. The most common use for this is to create decorators. In Python, you can define special kinds of objects that wrap up your methods and add extra behavior. Some decorators are builtin already like ‘property’, ‘classmethod’ and ‘staticmethod’. Once you have a decorator, you can sprinkle it on to of another function to add new behavior.

Decorator syntax looks something like this

```
@decorator_func_name(arg1, arg2, arg3, ...)
def some_functions(x, y, z, ...):
    # Do something useful here
    pass
```

Suppose we have some method that requires intensive computational resoures to run, but the results do not vary much over time. Wouldn’t it be nice if we could cache the results so that the computation wouldn’t have to run each and every time?

Here’s our class with a slow computation method

```
import time
class Foobar(object):
    def slow_compute(self, *args, **kwargs):
        time.sleep(1)
        return args, kwargs, 42
```

Now let’s cache the value using a decorator function. Our strategy is that for any function named X with some argument list, we want to create a unique name and save the final computed value to that name. We want our cached value to have a human readable name, we we want to reuse the original function name, as well as the arguments that were passed in the first time.

Let’s get to some code!

```

import hashlib
def cache(func):
    """
    This decorator will add a _cache_functionName_HEXDIGEST
    attribute after the first invocation of an instance method to
    store cached values.
    """
    # Obtain the function's name
    func_name = func.func_name
    # Compute a unique value for the unnamed and named arguments
    arghash = hashlib.sha1(str(args) + str(kwargs)).hexdigest()
    cache_name = '_cache_%s_%s' % (func_name, arghash)
    def inner(self, *args, **kwargs):
        if hasattr(self, cache_name):
            # If we have a cached value, just use it
            print "Fetching cached value from : %s" % cache_name
            return getattr(self, cache_name)
        result = func(self, *args, **kwargs)
        setattr(self, cache_name, result)
        return result
    return inner

```

There are only two new tricks that are in this code.

1. I'm using the hashlib module to convert the arguments to the function into a unique single string.
2. The use of getattr, hasattr and setattr to manipulate the cached value on the instance object.

Now, if we want to cache the slow method, we just throw on a @cache line above the method declaration.

```

@cache
def slow_compute(self, *args, **kwargs):
    time.sleep(1)
    return args, kwargs, 42

```

Fantastic - we can reuse this cache decorator for any method we want now. Let's suppose now that we want our cache to invalidate itself after every 3 calls. This practical use of currying is only a slight modification to the original caching code.

```

import hashlib
def cache(loop_iter):
    def function_closure(func):
        func_name = func.func_name
        def closure(self, loop_iter, *args, **kwargs):
            arghash = hashlib.sha1(str(args) + str(kwargs)).hexdigest()
            cache_name = '_cache_%s_%s' % (func_name, arghash)
            counter_name = '_counter_%s_%s' % (func_name, arghash)
            if hasattr(self, cache_name):
                # If we have a cached value, just use it
                print "Fetching cached value from : %s" % cache_name
                loop_iter -= 1
                setattr(self, counter_name, loop_iter)
                result = getattr(self, cache_name)
                if loop_iter == 0:
                    delattr(self, counter_name)
                    delattr(self, cache_name)
                    print "Cleared cached value"
                return result
            result = func(self, *args, **kwargs)

```

```
        setattr(self, cache_name, result)
        setattr(self, counter_name, loop_iter)
        return result
    return closure
return function_closure
```

Now we're free to use @cache for any slow method and caching will come in for free - including automatic invalidation of the cached value. Just use it like this

```
@cache(10)
def slow_compute(self, *args, **kwargs):
    # TODO: stuff goes here...
    pass
```

Review - and a taste of how we could fit all of this together

Now - I'm going to ask you to use your imagination a little. We've covered quite a bit of ground really quickly.

We can :

- look up attributes in an object (use the `__dict__` attribute).
- check if an object belongs to a particular class hierarchy (use the `isinstance` function).
- build functions out of other functions using currying, and even bind those functions to arbitrary names

This is fantastic - we now have all the basic building blocks we need to generate complex methods based on the attributes of our class. Imagine a simplified addressbook application with a simple contact.

```
class Contact(object):
    first_name = str
    last_name = str
    date_of_birth = datetime.Date
```

Assuming we know how to save and load to a database, we can use the function generation techniques to automatically generate `load()` and `save()` methods and bind them into our `Contact` class. We can use our introspection techniques to determine what attributes need to be saved to our database. We could even grow special methods onto our `Contact` class so that we could iterate over all of the class attributes and magically grow `'searchby_first_name'` and `'searchby_last_name'` methods.

See how powerful this can be? We can write extremely minimal code, and we could code generate all of our required specialized behavior for saving, loading and searching for records in a database. Since we do all of that programmatically - we can dramatically reduce the amount of code that we have to write by hand and by doing so - we can reduce the chance that we introduce bugs into our system.

We're going to do exactly that in a later chapter. Build a simple database abstraction layer to demonstrate how to create your own object system that will automatically know how to read and write to a database.

Chapter 6: Exception Handling and Debugging

Any good program makes use of a language's exception handling mechanisms. There is no better way to frustrate an end-user than by having them run into an issue with your software and displaying a big ugly error message on the screen, followed by a program crash. Exception handling is all about ensuring that when your program encounters an issue, it will continue to run and provide informative feedback to the end-user or program administrator. Any Java programmer becomes familiar with exception handling on day one, as some Java code won't even compile unless there

is some form of exception handling put into place via the try-catch-finally syntax. Python has similar constructs to that of Java, and we'll discuss them in this chapter.

After you have found an exception, or preferably before your software is distributed, you should go through the code and debug it in order to find and repair the erroneous code. There are many different ways to debug and repair code; we will go through some debugging methodologies in this chapter. In Python as well as Java, the *assert* keyword can help out tremendously in this area. We'll cover *assert* in depth here and learn the different ways that it can be used to help you out and save time debugging those hard-to-find errors.

Exception Handling Syntax and Differences with Java

Java developers are very familiar with the *try-catch-finally* block as this is the main mechanism that is used to perform exception handling. Python exception handling differs a bit from Java, but the syntax is fairly similar. However, Java differs a bit in the way that an exception is *thrown* in code. Now, realize that I just used the term *throw* ... this is Java terminology. Python does not *throw* exceptions, but instead it *raises* them. Two different terms which mean basically the same thing. In this section, we'll step through the process of handling and raising exceptions in Python code, and show you how it differs from that in Java.

For those who are unfamiliar, I will show you how to perform some exception handling in the Java language. This will give you an opportunity to compare the two syntaxes and appreciate the flexibility that Python offers.:

```
try {
    // perform some tasks that may throw an exception
} catch (ExceptionType messageVariable) {
    // perform some exception handling
} finally {
    // execute code that must always be invoked
}
```

Now let's go on to learn how to make this work in Python. Not only will we see how to handle and raise exceptions, but you'll also learn some other great techniques later in the chapter.

Catching Exceptions

How often have you been working in a program and performed some action that caused the program to abort and display a nasty error message? It happens more often than it should because most exceptions can be caught and handled nicely. By nicely, I mean that the program will not abort and the end user will receive a descriptive error message stating what the problem is, and in some cases how it can be resolved. The exception handling mechanisms within programming languages were developed for this purpose.

Below is a table of all exceptions that are built into the Python language along with a description of each. You can write any of these into a clause and try to handle them. Later in this chapter I will show you how you and them if you'd like. Lastly, if there is a specific type of exception that you'd like to throw that does not fit any of these, then you can write your own exception type object.

Exception	Description
BaseException	This is the root exception for all others
GeneratorExit	Raised by close() method of generators for terminating iteration
KeyboardInterrupt	Raised by the interrupt key
SystemExit	Program exit
Exception	Root for all non-exiting exceptions
StopIteration	Raised to stop an iteration action
StandardError	Base class for all built-in exceptions

Continued on next page

Table 2.2 – continued from previous page

Exception	Description
ArithmeticError	Base for all arithmetic exceptions
FloatingPointError	Raised when a floating-point operation fails
OverflowError	Arithmetic operations that are too large
ZeroDivisionError	Division or modulo operation with zero is divisor
AssertionError	Used when an assert statement fails
AttributeError	Attribute reference or failure to assign correctly
EnvironmentError	An error occurred outside of Python
IOError	Error in Input/Output operation
OSError	An error occurred in the os module
EOFError	input() or raw_input() tried to read past the end of a file
ImportError	Import failed to find module or name
LookupError	Base class for IndexError and KeyError
IndexError	A sequence index goes out of range
KeyError	Referenced a non-existent mapping (dict) key
MemoryError	Memory exhausted
NameError	Failure to find a local or global name
UnboundLocalError	Unassigned local variable is referenced
ReferenceError	Attempt to access a garbage-collected object
RuntimeError	Obsolete catch-all error
NotImplementedError	Raised when a feature is not implemented
SyntaxError	Parser encountered a syntax error
IndentationError	Parser encountered an indentation issue
TabError	Incorrect mixture of tabs and spaces
SystemError	Non-fatal interpreter error
TypeError	Inappropriate type was passed to a built-in operator or function
ValueError	Argument error not covered by TypeError or a more precise error
Warning	Base for all warnings

The *try-except-finally* block is used in Python programs to perform the exception-handling task. Much like that of Java, code that may or may not raise an exception should be placed in the *try* block. Differently though, exceptions that may be caught go into an *except* block much like the Java *catch* equivalent. Any tasks that must be performed no matter if an exception is thrown or not should go into the *finally* block.

try-except-finally Logic

```
try:
    # perform some task that may raise an exception
except Exception, value:
    # perform some exception handling
finally:
    # perform tasks that must always be completed
```

Python also offers an optional *else* clause to create the *try-except-else* logic. This optional code placed inside the *else* block is run if there are no exceptions found in the block.

try-finally logic:

```
try:
    # perform some tasks that may raise an exception
finally:
    # perform tasks that must always be completed
```

try-except-else logic:

```

try:
    # perform some tasks that may raise an exception
except:
    # perform some exception handling
else:
    # perform some tasks that should only be performed if no exceptions are thrown

```

You can name the specific type of exception to catch within the *except* block, or you can generically define an exception handling block by not naming any exception at all. Best practice of course states that you should always try to name the exception and then provide the best possible handling solution for the case. After all, if the program is simply going to spit out a nasty error then the exception handling block does not help resolve the issue at all. However, there are some rare cases where it would be advantageous to not explicitly refer to an exception type when we simply wish to ignore errors and move on. The *except* block also allows us to define a variable to which the exception message will be assigned. This allows us the ability to store that message and display it somewhere within our exception handling code block. If you are calling a piece of Java code from within Jython and the Java code throws an exception, it can be handled within Jython in the same manner as Jython exceptions.

Example 5-1: Exception Handling in Python

```

# Code without an exception handler
>>> x = 10
>>> z = x / y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined

# The same code with an exception handling block
>>> x = 10
>>> try:
...     z = x / y
... except NameError, err:
...     print "One of the variables was undefined: ", err
...

One of the variables was undefined:  name 'y' is not defined

```

Take note of the syntax that is being used for defining the variable that holds the error message. Namely, the *except ExceptionType, value* statement syntax in Python and Jython 2.5 differs from that beyond 2.5. In Python 2.6, the syntax changes a bit in order to ready developers for Python 3, which exclusively uses the new syntax. Without going off topic too much, I think it is important to take note that this syntax will be changing in future releases of Jython.

Jython and Python 2.5 and Prior

```

try:
    // code
except ExceptionType, messageVar:
    // code

```

Jython 2.6 (Not Yet Implemented) and Python 2.6 and Beyond

```

try:
    // code
except ExceptionType as messageVar:
    // code

```

We had previously mentioned that it was simply bad programming practice to not explicitly name an exception type when writing exception handling code. This is true, however Python provides us with another means to obtain the type of exception that was thrown. There is a function provided in the *sys* package known as *sys.exc_info()* that will

provide us with both the exception type and the exception message. This can be quite useful if we are wrapping some code in a *try-except* block but we really aren't sure what type of exception may be thrown. Below is an example of using this technique.

Example 5-2: Using `sys.exc_info()`

```
# Perform exception handling without explicitly naming the exception type
>>> x = 10
>>> try:
...     z = x / y
... except:
...     print "Unexpected error: ", sys.exc_info()[0], sys.exc_info()[1]
...
Unexpected error: <type 'exceptions.NameError'> name 'y' is not defined
```

Sometimes you may run into a situation where it is applicable to catch more than one exception. Python offers a couple of different options if you need to do such exception handling. You can either use multiple *except clauses*, which does the trick and works well, but may become too wordy. The other option that you have is to enclose your exception types within parentheses and separated by commas on your *except* statement. Take a look at the following example that portrays the latter approach using the same example from *Example 5-1*.

Example 5-3: Handling Multiple Exceptions

```
# Catch NameError, but also a ZeroDivisionError in case a zero is used in the equation
>>> x = 10
>>> try:
...     z = x / y
... except (NameError, ZeroDivisionError), err:
...     print "One of the variables was undefined: ", err
...
One of the variables was undefined: name 'y' is not defined

# Using multiple except clauses
>>> x = 10
>>> y = 0
>>> try:
...     z = x / y
... except NameError, err1:
...     print err1
... except ZeroDivisionError, err2:
...     print 'You cannot divide a number by zero!'
...
You cannot divide a number by zero!
```

The *try-except* block can be nested as deep as you'd like. In the case of nested exception handling blocks, if an exception is thrown then the program control will jump out of the inner most block that received the error, and up to the block just above it. This is very much the same type of action that is taken when you are working in a nested loop and then run into a *break* statement, your code will stop executing and jump back up to the outer loop. The following example shows an example for such logic.

Example 5-4: Nested Exception Handling Blocks

```
# Perform some division on numbers entered by keyboard
try:
    # do some work
    try:
        x = raw_input('Enter a number for the dividend: ')
        y = raw_input('Enter a number to divisor: ')

```



```

    x = int(x)
    y = int(y)
    except ValueError, err2:
        # handle exception and move to outer try-except
        print 'You must enter a numeric value!'
    z = x / y
    except ZeroDivisionError, err1:
        # handle exception
        print 'You cannot divide by zero!'
    except TypeError, err3:
        print 'Retry and only use numeric values this time!'
    else:
        print 'Your quotient is: %d' % (z)

```

Raising Exceptions

Often times you will find reason to raise your own exceptions. Maybe you are expecting a certain type of keyboard entry, and a user enters something incorrectly that your program does not like. This would be a case when you'd like to raise your own exception. The *raise* statement can be used to allow you to raise an exception where you deem appropriate. Using the *raise* statement, you can cause any of the Python exception types to be raised, you could raise your own exception that you define (discussed in the next section), or you could raise a string exception. The *raise* statement is analogous to the *throw* statement in the Java language. In Java we may opt to throw an exception if necessary. However, Java also allows you to apply a *throws* clause to a particular method if an exception may possibly be thrown within instead of using try-catch handler in the method. Python does not allow you to perform such techniques using the *raise* statement.

raise Statement Syntax

```
raise ExceptionType or String[, message[, traceback]]
```

As you can see from the syntax, using *raise* allows you to become creative in that you could use your own string when raising an error. However, this is not really looked upon as a best practice as you should try to raise a defined exception type if at all possible. You can also provide a short message explaining the error. This message can be any string. Lastly, you can provide a *traceback* via use of *sys.exc_info()*. Now you've surely seen some exceptions raised in the Python interpreter by now. Each time an exception is raised, a message appears that was created by the interpreter to give you feedback about the exception and where the offending line of code may be. There is always a *traceback* section when any exception is raised. This really gives you more information on where the exception was raised.

Example 5-5: Using the raise Statement

```

>>> raise TypeError, "This is a special message"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: This is a special message

```

Defining Your Own Exceptions

You can define your own exceptions in Python by creating an exception class. Now classes are a topic that we have not yet covered, so this section gets a little ahead, but it is fairly straightforward. You simply define a class using the *class* keyword and then give it a name. An exception class should inherit from the base exception class, *Exception*. The easiest defined exception can simply use a pass statement inside the class. More involved exception classes can accept parameters and define an initializer. It is also a good practice to name your exception giving it a suffix of *Error*.

Example 5-6: Defining an Exception Class

```
class MyNewError(Exception):  
    pass
```

The example above is the simplest type of exception you can create. This exception that was created above can be raised just like any other exception now.

```
raise MyNewError, "Something happened in my program"
```

A more involved exception class may be written as follows.

Example 5-7: Exception Class Using Initializer

```
class MegaError(Exception):  
    """ This is raised when there is a huge problem with my program"""  
    def __init__(self, val):  
        self.val = val  
    def __str__(self):  
        return repr(self.val)
```

Issuing Warnings

Warnings can be raised at any time in your program and can be used to display some type of warning message, but they do not necessarily cause execution to abort. A good example is when you wish to deprecate a method or implementation but still make it usable for compatibility. You could create a warning to alert the user and let them know that such methods are deprecated and point them to the new definition, but the program would not abort. Warnings are easy to define, but they can be complex if you wish to define rules on them using filters. Much like exceptions, there are a number of defined warnings that can be used for categorizing. In order to allow these warnings to be easily converted into exceptions, they are all instances of the *Exception* type.

Table 5-2. Python Warning Categories

Warning	Description
Warning	Root warning class
UserWarning	A user-defined warning
DeprecationWarning	Warns about use of a deprecated feature
SyntaxWarning	Syntax issues
RuntimeWarning	Runtime issues
FutureWarning	Warns that a particular feature will be changing in a future release

Table 5-1: Exceptions

To issue a warning, you must first import the *warnings* module into your program. Once this has been done then it is as simple as making a call to the *warnings.warn()* function and passing it a string with the warning message. However, if you'd like to control the type of warning that is issued, you can also pass the warning category.

```
import warnings  
...  
warnings.warn("this feature will be deprecated")  
warnings.warn("this is a more involved warning", RuntimeWarning)
```

Importing the *warnings* module into your code gives you access to a number of built-in warning functions that can be used. If you'd like to filter a warning and change its behavior then you can do so by creating a filter. The following is a list of functions that come with the *warnings* module.

Function and Description	
<code>warn(message[, category[, stacklevel]])</code>	Issues a warning. Parameters include a message string, the optional category of warning, and the optional stacklevel that tells which stack frame the warning should originate from.
<code>warn_explicit(message, category, filename, lineno[, module[, registry]])</code>	This offers a more detailed warning message and makes category a mandatory parameter. filename, lineno, and module tell where the warning is located. registry represents all of the current warning filters that are active.
<code>showwarning(message, category, filename, lineno[, file])</code>	Gives you the ability to write the warning to a file.
<code>formatwarning(message, category, filename, lineno)</code>	Creates a formatted string representing the warning.
<code>resetwarnings()</code>	Resets all of the warning filters.
<code>filterwarnings(action[, message[, category[, module[, lineno[, append]]]])</code>	

This adds an entry into a warning filter list. Warning filters allow you to modify the behavior of a warning. The action in the warning filter can be one from the following table of actions, message is a regular expression, category is the type of a warning to be issued, module can be a regular expression, lineno is a line number to match against all lines, append specifies whether the filter should be appended to the list of all filters.

Filter Actions	Description
'always'	Always print warning message
'default'	Print warning once for each location where warning occurs
'error'	Converts a warning into an exception
'ignore'	Ignores the warning
'module'	Print warning once for each module in which warning occurs
'once'	Print warning only one time

Table 5-3. Warning Functions

Warning filters are used to modify the behavior of a particular warning. There can be many different warning filters in use, and each call to the `filterwarnings()` function will append another warning to the list of filters if so desired. In order to see which filters are currently in use, issue the command `print warnings.filters`. One can also specify a warning filter from the command line by use of the `-W` option. Lastly, all warnings can be reset to defaults by using the `resetwarnings()` function.:

```
-Waction:message:category:module:lineno
```

Assertions and Debugging

Debugging can be an easy task in Python via use of the `assert` statement and the `__debug__` variable. Assertions are statements that can print to indicate that a particular piece of code is not behaving as expected. The assertion checks an expression for a True or False value, and if False then it issues an `AssertionError` along with an optional message. If the expression evaluates to True then the assertion is ignored completely.

```
assert expression [, message]
```

By effectively using the `assert` statement throughout your program, you can easily catch any errors that may occur and make debugging life much easier. The following example will show you the use of the `assert` statement.:

```
# The following example shows how assertions are evaluated
>>> x = 5
>>> y = 10
>>> assert x < y, "The assertion is ignored"
```

```
>>> assert x > y, "The assertion works"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: The assertion works
```

You can make use of the internal `*__debug__*` variable by placing entire blocks of code that should be run for debugging purposes only inside a conditional based upon value of the variable.

Example 5-10: Making Use of `__debug__`

```
if __debug__:
    # perform some debugging tasks
```

Context Managers

Ensuring that code is written properly in order to manage resources such as files or database connections is an important topic. If files or database connections are opened and never closed then our program could incur issues. Often times, developers elect to make use of the issues. Often times, developers elect to make use of the *try-finally* blocks to ensure that such resources are handled properly. While this is an acceptable method for resource management, it can sometimes be misused and lead to problems when exceptions are raised in programs. For instance, if we are working with a database connection and an exception occurs after we've opened the connection, the program control may break out of the current block and skip all further processing. The connection may never be closed in such a case. That is where the concept of context management becomes an important new feature in Jython. Context management via the use of the *with* statement is new to Jython 2.5, and it is a very nice way to ensure that resources are managed as expected.

In order to use the *with* statement, you must import from `__future__`. The *with* statement basically allows you to take an object and use it without worrying about resource management. For instance, let's say that we'd like to open a file on the system and read some lines from it. To perform a file operation you first need to open the file, perform any processing or reading of file content, and then close the file to free the resource. Context management using the *with* statement allows you to simply open the file and work with it in a concise syntax.

Example 5-11: Python with Statement Example

```
# Read from a text file named players.txt
>>> from __future__ import with_statement
>>> with open('players.txt','r') as file:
...     x = file.read()
...
>>> print x
This is read from the file
```

In the example above, we did not worry about closing the file because the context took care of that for us. This works with object that extends the context management protocol. In other words, any object that implements two methods named `__enter__()` and `__exit__()` adhere to the context management protocol. When the *with* **statement begins*, the **__enter__()* method is executed. Likewise, as the last action performed when the *with* statement is ending, the **__exit__()* method is executed. The `__enter__()` method takes no arguments, whereas the `__exit__()` method takes three optional arguments *type*, *value*, **and* *traceback*. The **__exit__()* method returns a *True* or *False* value to indicate whether an exception was thrown. The *as variable* clause on the *with* statement is optional as it will allow you to make use of the object from within the code block. If you are working with resources such as a lock then you may not the optional clause.

If you follow the context management protocol, it is possible to create your own objects that can be used with this technique. The `__enter__()` method should create whatever object you are trying to work if needed. If you are working with an immutable object then you'll need to create a copy of that object to work with in the `__enter__()` method. The

`__exit__()` method on the other hand can simply return *False* unless there is some other type of cleanup processing that needs to take place.

Summary

In this chapter, we discussed many different topics regarding exceptions and exception handling within a Python application. First, you learned the exception handling syntax of the *try-except-finally* code block and how it is used. We then discussed why it may be important to *raise* your own exceptions at times and how to do so. That topic led to the discussion of how to define an exception and we learned that in order to do so we must define a class that extends the *Exception* type object.

After learning about exceptions, we went into the warnings framework and discussed how to use it. It may be important to use warnings in such cases where code may be deprecated and you want to warn users, but you do not wish to *raise* any exceptions. That topic was followed by assertions and how assertion statement can be used to help us debug our programs. Lastly, we touched upon the topic of context managers and using the *with* statement that is new in Jython 2.5.

In the next chapter you will delve into creating classes and learning about object-oriented programming in Python. Hopefully if there were topics discussed in this chapter or previously in the book that may have been unclear due to unfamiliarity with object orientation, they will be clarified in Chapter 6.

Chapter 7: Modules and Packages

Up until this chapter we have been looking at code at the level of the interactive console and simple scripts. This works well for small examples, but when your program gets larger, it becomes necessary to break programs up into smaller units. In Python, the basic building block for these units in larger programs is the module.

Imports For Re-Use

Breaking code up into modules helps to organize large code bases. Modules can be used to logically separate code that belongs together, making programs easier to understand. Modules are helpful for creating libraries that can be imported and used in different applications that share some functionality. Jython's standard library comes with a large number of modules that can be used in your programs right away.

Import Basics

The following discussion will use the a silly example file called `breakfast.py`:

```
class Spam(object):

    def order(self, number):
        print "spam " * number

def order_eggs():
    print " and eggs!"

s = Spam()
s.order(3)
order_eggs()
```

We'll start with a couple of definitions. A *namespace* is a logical grouping of unique identifiers. In other words, a namespace is that set of names that can be accessed from a given bit of code in your program. For example, if you open up a Jython prompt and type `dir()`, the names in the interpreter's namespace will be displayed.

```
>>> dir()
['__doc__', '__name__']
```

The interpreter namespace contains `__doc__` and `__name__`. The `__doc__` property contains the top level docstring, which is empty in this case. We'll get to the `__name__` property in a moment. First we need to talk about Jython *modules*. A *module* in Jython is a file containing Python definitions and statements which in turn define a namespace. The module name is the same as the file name with the suffix `.py` removed, so in our current example the Python file `"breakfast.py"` defines the module `"breakfast"`.

Now we can talk about the `__name__` property. When a module is run directly, as in `"jython breakfast.py"`, `__name__` will contain `'__main__'`. If a module is imported, `__name__` will contain the name of the module, so `"import breakfast"` results in the `breakfast` module containing a `__name__` of `"breakfast"`.

```
>>> __doc__
>>> __name__
'__main__'
```

Let's see what happens when we import breakfast: ::

```
>>> import breakfast
spam spam spam
    and eggs!
>>> dir()
['__doc__', '__name__', 'breakfast']
```

Checking the `doc()` after the import shows that `breakfast` has been added to the top level namespace. Notice that the act of importing actually executed the code in `breakfast.py`. Most of the time, we wouldn't want a module to execute in this way on import. To avoid this, but allow the code to execute when it is called directly, we typically check the `__name__` property:

```
class Spam(object):

    def order(self, number):
        print "spam " * number

def order_eggs():
    print " and eggs!"

if __name__ == '__main__':
    s = Spam()
    s.order(3)
    order_eggs()
```

Now if we import breakfast, we will not get the output: ::

```
>>> import breakfast
```

This is because in this case the `__name__` property will contain `'breakfast'`, the name of the module. If we call `breakfast.py` from the commandline like `"jython breakfast.py"` we would then get the output again, because `breakfast` would be executing as `__main__`.

In languages like Java, the import statement is strictly a compiler directive that must occur at the top of the source file. In Jython, the import statement is an expression that can occur anywhere in the source file, and can even be conditionally executed.

As an example, a common idiom is to attempt to import something that may not be there in a try block, and in the except block import a module that is known to be there.

```
>>> try:
...     from blah import foo
... except ImportError:
...     def foo():
...         return "hello from backup foo"
...
>>> foo()
'hello from backup foo'
>>>
```

If a module named `blah` had existed, the definition of `foo` would have been taken from there. Since no such module existed, `foo` was defined in the except block, and when we called `foo`, the ‘hello from backup foo’ string was returned.

I should point out that `dir()` does not actually print out the entire namespace for the top level of the interpreter. There are a large number of names that are omitted since the `dir()` output would not be as useful. The special `__builtin__` module can be imported to see the rest:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', ...]
```

Packages

Unfortunately, Jython must contend with two very different definitions of “Package”. In the Python world, a *Python package* is a directory containing an `__init__.py` file. The directory usually contains some Python modules which are said to be contained in the package. The `__init__.py` file is executed before any contained modules are imported.

In the Java world, a *Java package* organizes Java classes into a namespace using nested directories. Java packages do not require an `__init__.py` file. Also unlike Python packages, Java packages are explicitly referenced in each Java file with a package directive at the top.

```
chapter7/
  searchdir.py
  search/
    __init__.py
    walker.py
    scanner.py
```

The example contains one package: `search`, which is a package because it is a directory containing the special `__init__.py` file. In this case `__init__.py` is empty and so only serves as a marker that `search` is a package. If `__init__.py` contained code, it would be executed before any of its containing modules could be imported. Note that the directory `chapter7` itself is not a package because it does not contain an `__init__.py`. There are three modules in the example program: `searchdir`, `search.input` and `search.scanner`. The code for this program can be downloaded at XXX.

`searchdir.py` ~~~~~

```
import search.scanner as scanner
import sys

help = """
Usage: search.py directory terms...
"""

args = sys.argv
```

```
if args == None or len(args) < 2:
    print help
    exit()

dir = args[1]
terms = args[2:]
scan = scanner.scan(dir, terms)
scan.display()
```

scanner.py ———:

```
from search.walker import DirectoryWalker
from javax.swing import JFrame, JTable, WindowConstants

class ScanResults(object):
    def __init__(self):
        self.results = []

    def add(self, file, line):
        self.results.append((file, line))

    def display(self):
        colnames = ['file', 'line']
        table = JTable(self.results, colnames)
        frame = JFrame("%i Results" % len(self.results))
        frame.getContentPane().add(table)
        frame.size = 400, 300
        frame.defaultCloseOperation = WindowConstants.EXIT_ON_CLOSE
        frame.visible = True

    def scan(dir, terms):
        results = ScanResults()
        for filename in DirectoryWalker(dir):
            for line in open(filename):
                for term in terms:
                    if term in line:
                        results.add(filename, line)
        return results
```

walker.py ———:

```
import os

class DirectoryWalker:
    # A forward iterator that traverses a directory tree. Adapted from an
    # example in the eff-bot library guide: os-path-walk-example-3.py

    def __init__(self, directory):
        self.stack = [directory]
        self.files = []
        self.index = 0

    def __getitem__(self, index):
        while 1:
            try:
                file = self.files[self.index]
                self.index = self.index + 1
```



```

except IndexError:
    # pop next directory from stack
    self.directory = self.stack.pop()
    self.files = os.listdir(self.directory)
    self.index = 0
else:
    # got a filename
    fullname = os.path.join(self.directory, file)
    if (os.path.isdir(fullname) and not
        os.path.islink(fullname)):
        self.stack.append(fullname)
    else:
        return fullname

```

If you run `searchdir.py` on it's own directory like this:

Trying out the Example Code _____:

```
$ jython scanner.py . terms
```

You will get a swing table titled “5 Results” (possibly more if `.class` files are matched). Let’s examine the import statements used in this program. The module `searchdir` contains two import statements::

```

import search.scanner as scanner
import sys

```

The first imports the module “`search.scannar`” and renames the module “`scannar`”. The second imports the builtin module “`sys`” and leaves the name as “`sys`”. The module “`search.scannar`” has two import statements:

```

from search.walker import DirectoryWalker
from javax.swing import JFrame, JTable, WindowConstants

```

The first imports `DirectoryWalker` from the “`search.walker`” module. Note that we had to do this even though `search.walker` is in the same package as `search.scanner`. The last import is interesting because it imports the java classes like `JFrame` from the java package `javax.swing`. Jython makes this sort of import look the same as other imports. This simple example shows how you can import code from different modules and packages to modularize your programs.

Types of import statements

The import statement comes in a variety of forms that allow much finer control over how importing brings named values into your current module.

Basic import Statements _____

```

import module
from module import submodule
from . import submodule

```

I will discuss each of the import statement forms in turn starting with:

```
import module
```

This most basic type of import imports a module directly. Unlike Java, this form of import binds the leftmost module name, so If you import a nested module like:

```
import javax.swing.JFrame
```

You would need to refer to it as “javax.swing.JFrame” in your code. In Java this would have imported “JFrame”.

from import Statements —————

```
from module import name
```

This form of import allows you to import modules, classes or functions nested in other modules. This allows you to achieve the result that a typical Java import gives. To get a JFrame in your Jython code you issue:

```
from javax.swing import JFrame
```

You can also use the from style of import to import all of the names in a module directly into your current module using a “*”. This form of import is discouraged in the Python community, and is particularly troublesome when importing from Java packages (in some cases it does not work, see chapter 10 for details) so you should avoid its use. It looks like this:

```
from module import *
```

Relative import Statements

A new kind of import introduced in Python 2.5 is the explicit relative import. These import statements use dots to indicate how far back you will walk from the current nesting of modules, with one dot meaning the current module.

```
from . import module
from .. import module
from .module import submodule
from ..module import submodule
```

Even though this style of importing has just been introduced, its use is discouraged. Explicit relative imports are a reaction to the demand for implicit relative imports. If you look at the search.scanner package, you will see the import statement:

```
from search.walker import DirectoryWalker
```

Because search.walker sits in the same package as search.scanner, the import statement could have been:

```
from walker import DirectoryWalker
```

Some programmers like to use relative imports like this so that imports will survive module restructuring, but these relative imports can be error prone because of the possibility of name clashes. The new syntax provides an explicit way to use relative imports, though they too are still discouraged. The import statement above would look like this:

```
from .walker import DirectoryWalker
```

Aliasing import Statements

Any of the above imports can add an “as” clause to change import a module but give it a new name.

```
import module as alias
from module import submodule as alias
from . import submodule as alias
```

This gives you enormous flexibility in your imports, so to go back to the JFrame example, you could issue:

```
import javax.swing.JFrame as Foo
```

And instantiate a JFrame object with a call to Foo(), something that would surprise most Java developers coming to Jython.

Hiding Module Names

Typically when a module is imported, all of the names in the module are available to the importing module. There are a couple of ways to hide these names from importing modules. Starting any name with an underscore (_) which is the Python convention for marking names as private is the first way. The second way to hide module names is to define a list named `__all__`, which should contain only those names that you wish to have your module to expose. As an example here is the value of `__all__` at the top of Jython's `os` module:

```
__all__ = ["altsep", "curdir", "pardir", "sep", "pathsep",
          "linesep", "defpath", "name", "path",
          "SEEK_SET", "SEEK_CUR", "SEEK_END"]
```

Note that you can add to `__all__` inside of a module to expand the exposed names of that module. In fact, the `os` module in Jython does just this to conditionally expose names based on the operating system that Jython is running on.

Module Search Path, Compilation, and Loading

Compilation

Despite the popular belief that Jython is an “interpreted, not compiled”, in reality all Jython code is turned into Java bytecodes before execution. These bytecodes are not always saved to disk, but when you see Jython execute any code, even in an `eval` or an `exec`, you can be sure that bytecodes are getting fed to the JVM. The sole exception to this that I know of is the experimental `pycimport` module that I will describe in the section on `sys.meta_path` below, which interprets CPython bytecodes instead of producing Java bytecodes.

Module search Path and Loading

Understanding the process of module search and loading is more complicated in Jython than in either CPython or Java because Jython can search both Java's `CLASSPATH` and Python's `path`. We'll start by looking at Python's `path` and `sys.path`. When you issue an `import`, `sys.path` defines the path that Jython will use to search for the name you are trying to import. The objects within the `sys.path` list tell Jython where to search for modules. Most of these objects point to directories, but there are a few special items that can be in `sys.path` for Jython that are not just pointers to directories. Trying to import a file that does not reside anywhere in the `sys.path` (and also cannot be found in the `CLASSPATH`) raises an `ImportError` exception. Let's fire up a command line and look at `sys.path`.

```
>>> import sys
>>> sys.path
['', '/Users/frank/jython/Lib', '__classpath__', '__pyclasspath__',
'/Users/frank/jython/Lib/site-packages']
```

The first blank entry (‘’) tells Jython to look in the current directory for modules. The second entry points to Jython's `Lib` directory that contains the core Jython modules. The third and forth entries are special markers that we will discuss later, and the last points to the `site-packages` directory where new libraries can be installed when you issue `setuptools` directives from Jython (see Chapter XXX for more about `setuptools`).

Import Hooks

To understand the way that Jython imports Java classes we have to understand a bit about the Python import protocol. I won't get into every detail, for that you would want to look at PEP 302 .

Briefly, we first try any custom importers registered on `sys.meta_path`. If one of them is capable of importing the requested module, allow that importer to handle it. Next, we try each of the entries on `sys.path`. For each of these, we find the first hook registered on `sys.path_hooks` that can handle the path entry. If we find an import hook and it successfully imports the module, we stop. If this did not work, we try the builtin import logic. If that also fails, an `ImportError` is thrown. So let's look at Jython's `path_hooks`.

`sys.path_hooks` —————

```
>>> import sys
>>> sys.path_hooks
[<type 'org.python.core.JavaImporter'>, <type 'zipimport.zipimporter'>,
<type 'ClasspathPyImporter'>]
```

Each of these `path_hooks` entries specifies a `path_hook` that will attempt to import special files. `JavaImporter`, as its name implies, allows the dynamic loading of Java packages and classes that are specified at runtime. For example, if you want to include a jar at runtime you can execute the following code, which will then get picked up by the `JavaImporter` hook the next time that an import is attempted:

```
>>> import sys
>>> sys.path.append("/Users/frank/lib/mysql-connector-java-5.1.6.jar")
>>> import com.mysql
*sys-package-mgr*: processing new jar, '/Users/frank/lib/mysql-connector-java-5.1.6.
↪ jar'
>>> dir(com.mysql)
['__name__', 'jdbc']
```

sys.meta_path

Adding entries to `sys.meta_path` allows you to add import behaviors that will occur before any other import is attempted, even the default builtin importing behavior. This can be a very powerful tool, allowing you to do all sorts of interesting things. As an example, I will talk about an experimental module that ships with Jython 2.5. That module is `pycimport`. If you start up jython and issue:

```
>>> import pycimport
```

Jython will start scanning for `.pyc` files in your path and if it finds one, will use the `.pyc` file to load your module. `.pyc` files are the files that CPython produces when it compiles Python source code. So, if you after you have imported `pycimport` (which adds a hook to `sys.meta_path`) then issue:

```
>>> import foo
```

Jython will scan your path for a file named `foo.pyc`, and if it finds one it will import the `foo` module using the CPython bytecodes. Here the code at the bottom of `pycimport.py` that makes defines the `MetaImporter` and adds it to `sys.meta_path`:

```
class __MetaImporter(object):
    def __init__(self):
        self.__importers = {}
    def find_module(self, fullname, path):
        if __debugging__: print "MetaImporter.find_module(%s, %s)" % (
            repr(fullname), repr(path))
```

```

    for _path in sys.path:
        if _path not in self.__importers:
            try:
                self.__importers[_path] = __Importer(_path)
            except:
                self.__importers[_path] = None
        importer = self.__importers[_path]
        if importer is not None:
            loader = importer.find_module(fullname, path)
            if loader is not None:
                return loader
        else:
            return None

sys.meta_path.insert(0, __MetaImporter())

```

The `find_module` method calls into other parts of `pycimport` and looks for `.pyc` files. If it finds one, it knows how to parse and execute those files and adds the corresponding module to the runtime. Pretty cool eh?

Java Package Scanning

Although you can ask the Java SDK to give you a list of all of the packages known to a `ClassLoader` using:

```
java.lang.ClassLoader#getPackages()
```

there is no corresponding

```
java.lang.Package#getClasses()
```

This is unfortunate for Jython, because Jython users expect to be able to introspect the code they use in powerful ways. For example, users expect to be able to call `dir()` on Java objects and packages to see what names they contain:

```

>>> import java.util.zip
>>> dir(java.util.zip)
['Adler32', 'CRC32', 'CheckedInputStream', 'CheckedOutputStream', 'Checksum',
↪ 'DataFormatException', 'Deflater', 'DeflaterOutputStream', 'GZIPInputStream',
↪ 'GZIPOutputStream', 'Inflater', 'InflaterInputStream', 'ZipEntry', 'ZipException',
↪ 'ZipFile', 'ZipInputStream', 'ZipOutputStream', '__name__']
>>> dir(java.util.zip.ZipInputStream)
['__class__', '__delattr__', '__doc__', '__eq__', '__getattr__', '__hash__', '__
↪ init__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__
↪ __', '__str__', 'available', 'class', 'close', 'closeEntry', 'equals', 'getClass',
↪ 'getNextEntry', 'hashCode', 'mark', 'markSupported', 'nextEntry', 'notify',
↪ 'notifyAll', 'read', 'reset', 'skip', 'toString', 'wait']

```

To make this sort of introspection possible in the face of merged namespaces requires some major effort the first time that Jython is started (and when jars or classes are added to Jython's path at runtime). If you have ever run a new install of Jython before, you will recognize the evidence of this system at work:

```

*sys-package-mgr*: processing new jar, '/Users/frank/jython/jython.jar'
*sys-package-mgr*: processing new jar, '/System/Library/Frameworks/JavaVM.framework/
↪ Versions/1.5.0/Classes/classes.jar'
*sys-package-mgr*: processing new jar, '/System/Library/Frameworks/JavaVM.framework/
↪ Versions/1.5.0/Classes/ui.jar'
*sys-package-mgr*: processing new jar, '/System/Library/Frameworks/JavaVM.framework/
↪ Versions/1.5.0/Classes/laf.jar'

```

```
...
*sys-package-mgr*: processing new jar, '/System/Library/Frameworks/JavaVM.framework/
↳Versions/1.5.0/Home/lib/ext/sunjce_provider.jar'
*sys-package-mgr*: processing new jar, '/System/Library/Frameworks/JavaVM.framework/
↳Versions/1.5.0/Home/lib/ext/sunpkcs11.jar'
```

This is Jython scanning all of the jar files that it can find to build an internal representation of the package and classes available on your JVM. This has the unfortunate side effect of making the first startup on a new Jython installation painfully slow.

How Jython Finds the Jars and Classes to scan

There are two properties that Jython uses to find jars and classes. These settings can be given to Jython using commandline settings or the registry (see Chapter XXX). The two properties are:

```
python.packages.paths
python.package.directories
```

These properties are comma separated lists of further registry entries that actually contain the values the scanner will use to build its listing. You probably should not change these properties. The properties that get pointed to by these properties are more interesting. The two that potentially make sense to manipulate are:

```
java.class.path
java.ext.dirs
```

For the `java.class.path` property, entries are separated as the classpath is separated on the operating system you are on (that is, “;” on Windows and “:” on most other systems). Each of these paths are checked for a `.jar` or `.zip` and if they have these suffixes they will be scanned.

For the `java.ext.dirs` property, entries are separated in the same manner as `java.class.path`, but these entries represent directories. These directories are searched for any files that end with `.jar` or `.zip`, and if any are found they are scanned.

To control the jars that are scanned, you need to set the values for these properties. There are a number of ways to set these property values, see Chapter XXX for more.

If you only use full class imports, you can skip the package scanning altogether. Set the system property `python.cachedir.skip` to true or (again) pass in your own `postProperties` to turn it off.

Python Modules and Packages vs. Java Packages

The basic semantics of importing Python modules and packages versus the semantics of importing Java packages into Jython differ in some important respects that need to be kept carefully in mind.

`sys.path`

When Jython tries to import a module, it will look in its `sys.path` in the manner described in the previous section until it finds one. If the module it finds represents a Python module or package, this import will display a “winner take all” semantic. That is, the first python module or package that gets imported blocks any other module or package that might subsequently get found on any lookups. This means that if you have a module `foo` that contains only a name `bar` early in the `sys.path`, and then another module also called `foo` that only contains a name `baz`, then executing “import `foo`” will only give you `foo.bar` and not `foo.baz`.

This differs from the case when Jython is importing Java packages. If you have a Java package `org.foo` containing `bar`, and a Java package `org.foo` containing `baz` later in the path, executing “`import org.foo`” will merge the two namespaces so that you will get both `org.foo.bar` and `org.foo.baz`.

Just as important to keep in mind, if there is a Python module or package of a particular name in your path that conflicts with a Java package in your path this will also have a winner take all effect. If the Java package is first in the path, then that name will be bound to the merged Java packages. If the Python module or package wins, no further searching will take place, so the Java packages with the clashing names will never be found.

Naming Python Modules and Packages —————

Developers coming from Java will often make the mistake of modeling their Jython package structure the same way that they model Java packages. Do not do this. The reverse url convention of Java is a great, I would even say a brilliant convention for Java. It works very well indeed in the world of Java where these namespaces are merged. In the Python world however, where modules and packages display the winner take all semantic, this is a disastrous way to organize your code.

If you adopt this style for Python, say you are coming from “`acme.com`” so you would set up a package structure like “`com.acme`”. If you try to use a library from your vendor `xyz` that is set up as “`com.xyz`”, then the first of these on your path will take the “`com`” namespace, and you will not be able to see the other set of packages.

Proper Python Naming —————

The Python convention is to keep namespaces as shallow as you can, and make your top level namespace reasonably unique, whether it be a module or a package. In the case of `acme` and `company xyz` above, you might start your package structures with “`acme`” and “`xyz`” if you wanted to have these entire codebases under one namespace (not necessarily the right way to go – better to organize by product instead of by organization, as a general rule).

Note: There are at least two sets of names that are particularly bad choices for naming modules or packages in Jython. The first is any top level domain like `org`, `com`, `net`, `us`, `name`. The second is any of the domains that Java the language has reserved for its top level namespaces: `java`, `javax`.

Java Import Example

We’ll start with a Java class which is on the CLASSPATH when Jython is started:

```
package com.foo;
public class HelloWorld {
    public void hello() {
        System.out.println("Hello World!");
    }
    public void hello(String name) {
        System.out.printf("Hello %s!", name);
    }
}
```

Here we manipulate that class from the Jython interactive interpreter:

```
>>> from com.foo import HelloWorld
>>> h = HelloWorld()
>>> h.hello()
Hello World!
>>> h.hello("frank")
Hello frank!
```

It’s important to note that, because the `HelloWorld` program is located on the Java CLASSPATH, it did not go through the `sys.path` process we talked about before. In this case the Java class gets loaded directly by the `ClassLoader`. Discussions of Java `ClassLoaders` are beyond the scope of this book. To read more about `ClassLoader` see (citation? Perhaps point to the Java Language Specification section)

Conclusion

In this chapter we have learned how to divide code up into modules to for the purpose of organization and re-use. We have learned how to write modules and packages, and how the Jython system interacts with Java classes and packages. This ends Part I. We have now covered the basics of the Jython language and are now ready to learn how to use Jython.

Chapter 8: Scripting With Jython

In this chapter we will look at scripting with jython. For our purposes, I will define “scripting” as the writing of small programs to help out with daily tasks. These tasks are things like deleting and creating directories, managing files and programs, or anything else that feels repetitive that you might be able to express as a small program. In practice however, scripts can become so large that the line between a script and a full sized program can blur.

We’ll start with an overview of some of the most helpful modules that come with jython for these tasks. These modules are `os`, `shutil`, `getopt`, `optparse`, `subprocess`. We will just be giving you a quick feel for these modules. For details you should look at reference documentation like <http://jython.org/currentdocs>. Then we’ll cover a medium sized task to show the use of a few of these modules together.

Parsing Commandline Options

Many scripts are simple one-offs that you write once, use, and forget. Others become part of your weekly or even daily use over time. When you find that you are using a script over and over again, you often find it helpful to pass in command line options. There are three main ways that this is done in jython. The first way is to hand parse the arguments, the second is the `getopt` module, and the third is the newer, more flexible `optparse` module.

Let’s say we have a script called `foo.py` and you want to be able to give it some parameters when you invoke it the name of the script and the arguments passed can be examined by importing the `sys` module and inspecting `sys.argv` like so:

```
# script foo.py import sys print sys.argv
```

If you run the above script with `a`, `b`, and `c` as arguments:

```
$ jython foo.py a b c
$ ['foo.py', 'a', 'b', 'c']
```

The name of the script ended up in `sys.argv[0]`, and the rest in `sys.argv[1:]`. Often you will see this instead in jython programs:

```
# script foo2.py import sys

args = sys.argv[1:] print args
```

which will result in:

```
$ jython foo2.py a b c
$ ['a', 'b', 'c']
```

If you are going to do more than just feed the arguments to your script directly, than parsing these arguments by hand can get pretty tedious. The jython libraries include two modules that you can use to avoid tedious hand parsing. Those modules are `getopt` and `optparse`. The `optparse` module is the newer, more flexible option, so I'll cover that one. The `getopt` module is still useful since it requires a little less code for simpler expected arguments. Here is a basic `optparse` script:

```
# script foo3.py
from optparse import optionparser
parser = optionparser()
parser.add_option("-f", "--foo" help="set foo option")
parser.add_option("-b", "--bar" help="set bar option")
(options, args) = parser.parse_args()
print "options: %s" % options
print "args: %s" % args
```

running the above:

```
$ jython foo3.py -b a --foo b c d
$ options: {'foo': 'b', 'bar': 'a'}
$ args: ['c', 'd']
```

I'll come back to the `optparse` module with a more concrete example later in this chapter.

Scripting The Filesystem

We'll start with what is probably the simplest thing that you can do to a filesystem, and that is listing the file contents of a directory.

```
>>> import os
>>> os.listdir('.')
['ast', 'doc', 'grammar', 'lib', 'license.txt', 'news', 'notice.txt', 'src']
```

First we imported the `os` module, and then we executed `listdir` on the current directory, indicated by the `'.'`. Of course your output will reflect the contents of your local directory. The `os` module contains many of the sorts of functions that you would expect to see for working with your operating system. The `os.path` module contains functions that help in working with filesystem paths.

Compiling Java Source

While compiling java source is not strictly a typical scripting task, it is a task that I'd like to show off in my bigger example starting in the next section. The api I am about to cover was introduced in jdk 6, and is optional for jvm vendors to implement. I know that it works on the jdk 6 from sun and on the jdk 6 that ships with mac os x. For more details of the `javacompiler` api, a good starting point is here: <http://java.sun.com/javase/6/docs/api/javax/tools/javacompiler.html>. The following is a simple example of the use of this api from jython

```

compiler = toolprovider.getsystemjavacompiler()
diagnostics = diagnosticcollector()
manager = compiler.getstandardfilemanager(diagnostics, none, none)
units = manager.getjavafileobjectsfromstrings(names)
comp_task = compiler.gettask(none, manager, diagnostics, none, none, units)
success = comp_task.call()
manager.close()

```

Example Script: builder.py

So I've discussed a few of the modules that tend to come in handy when writing scripts for jython. Now I'll put together a simple script to show off what can be done. I've chosen to write a script that will help handle the compilation of java files to .class files in a directory, and clean the directory of .class files as a separate task. I will want to be able to create a directory structure, delete the directory structure for a clean build, and of course compile my java source files.

```

import os
import sys
import glob

from javax.tools import (forwardingjavafilemanager, toolprovider,
                          diagnosticcollector,)

tasks = {}

def task(func):
    tasks[func.func_name] = func

@task
def clean():
    files = glob.glob("*.class")
    for file in files:
        os.unlink(file)

@task
def compile():
    files = glob.glob("*.java")
    _log("compiling %s" % files)
    if not _compile(files):
        quit()
    _log("compiled")

def _log(message):
    if options.verbose:
        print message

def _compile(names):
    compiler = toolprovider.getsystemjavacompiler()
    diagnostics = diagnosticcollector()
    manager = compiler.getstandardfilemanager(diagnostics, none, none)
    units = manager.getjavafileobjectsfromstrings(names)
    comp_task = compiler.gettask(none, manager, diagnostics, none, none, units)
    success = comp_task.call()
    manager.close()
    return success

if __name__ == '__main__':

```

```
from optparse import optionparser
parser = optionparser()
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print out task messages.")
parser.add_option("-p", "--projecthelp",
                  action="store_true", dest="projecthelp",
                  help="print out list of tasks.")
(options, args) = parser.parse_args()

if options.projecthelp:
    for task in tasks:
        print task
    sys.exit(0)

if len(args) < 1:
    print "usage: jython builder.py [options] task"
    sys.exit(1)
try:
    current = tasks[args[0]]
except KeyError:
    print "task %s not defined." % args[0]
    sys.exit(1)
current()
```

The script defines a “task” decorator that gathers the names of the functions and puts them in a dictionary. We have an optionparser class that defines two options `--projecthelp` and `--quiet`. By default the script logs its actions to standard out. `--quiet` turns this logging off. `--projecthelp` lists the available tasks. We have defined two tasks, “compile” and “clean”. The “compile” task globs for all of the `.java` files in your directory and compiles them. The “clean” task globs for all of the `.class` files in your directory and deletes them. Do be careful! The `.class` files are deleted without prompting!

So let's give it a try. If you create a Java class in the same directory as `builder.py`, say the classic “Hello World” program:

HelloWorld.java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

You could then issue these commands to `builder.py` with these results:

```
[frank@pacman chapter8]$ jython builder.py --help
Usage: builder.py [options]

Options:
  -h, --help            show this help message and exit
  -q, --quiet            Don't print out task messages.
  -p, --projecthelp     Print out list of tasks.
[frank@pacman chapter8]$ jython builder.py --projecthelp
compile
clean
[frank@pacman chapter8]$ jython builder.py compile
compiling ['HelloWorld.java']
```

```

compiled
[frank@pacman chapter8]$ ls
DEBUG.classicHelloWorld.java
HelloWorld.classicHelloWorldbuilder.py
[frank@pacman chapter8]$ jython builder.py clean
[frank@pacman chapter8]$ ls
HelloWorld.javabuilder.py
[frank@pacman chapter8]$ jython builder.py --quiet compile
[frank@pacman chapter8]$ ls
DEBUG.classicHelloWorldHelloWorld.java
HelloWorld.classicHelloWorldHelloWorldbuilder.py
[frank@pacman chapter8]$

```

Chapter 10: Jython and Java Integration

Java integration is the heart of Jython application development. Most Jython developers are either Python developers that are looking to make use of the vast library that the JVM has to offer, or Java developers that would like to utilize the Python language semantics without migrating to a completely different platform. The fact is that most Jython developers are using it so that they can take advantage of the vast libraries available to the Java world, and in order to do so there needs to be a certain amount of Java integration in the application. Whether you plan to use some of the existing Java libraries in your application, or your interested in mixing some great Python code into your Java application, this chapter is geared to help with the integration.

This chapter will focus on integrating Java and Python, but it will take several different vantage points on the topic. You will learn several techniques to make use Jython code within your Java applications. Perhaps you'd like to simplify your code a bit, this chapter will show you how to write certain portions of your code in Jython and others in Java so that you can make code as simple as possible. You'll learn how to make use of the many Java libraries within your Jython applications using Pythonic syntax! Forget about coding those programs in Java, why not use Jython so that the Java implementations in the libraries are behind the scenes, this chapter will show how to write in Python and use the libraries directly.

Using Jython within Java Applications

Often times, it is handy to have the ability to make use of Jython from within a Java application. Perhaps there is a class that would be better implemented in the Python syntax, such as a Javabean. Or maybe there is a handy piece of Jython code that would be useful within some Java logic. Whatever the case may be, there are several ways that can be used in order to achieve this combination of technologies. In this section, we'll cover some of the older techniques for using Jython within Java, and then go into the current and future best practices for doing this. In the end, you should have a good understanding for how to use a module, script, or even just a few lines of Jython within your Java application. You will also have an overall understanding for the way that Jython has evolved in this area.

A Bit of History

Prior to Jython 2.5, the standard distribution of Jython included a utility known as `jythonc`. It's main purpose was to provide the ability to convert Python modules into Java classes so that Java applications could seamlessly make use of Python code, albeit in a roundabout fashion. `jythonc` actually compiles the Jython code down into Java `.class` files and then the classes are utilized within the Java application. This utility could also be used to freeze code modules, create jar files, and to perform other tasks depending upon which options were used. This technique is no longer the recommended approach for utilizing Jython within Java applications, but I will briefly cover it over the next couple of paragraphs for good historical reference. As a matter of fact, `jythonc` is no longer packaged with the Jython distribution

beginning with the 2.5 release. For those who aren't interested in the older methodology, please feel free to jump ahead to the next section that covers one of the currently preferred methods for performing such tasks.

In order for jythonc to take a Jython class and turn it into a corresponding Java class, it had to adhere to a few standards. First, the Jython class had to subclass a Java object, either a class or interface. It also had to do one of the following: override a Java method, implement a Java method, or create a new method using a signature. We will go through a brief example utilizing this technique with Jython 2.2.1. As this is no longer the preferred or accepted approach, we will not delve deep into the topic.:

```
Fish.py
class Fish(object):
    """ Fish Object """
    def __init__(self, type, salt_or_fresh):
        self.type = type
        self.salt_or_fresh = salt_or_fresh
    def getType(self):
        "@sig public String getType()"
        return self.type
    def getSaltOrFresh(self):
        "@sig public String getSaltOrFresh()"
        return self.salt_or_fresh
```

While this method worked well and did what it was meant to do, it caused a separation between the Jython code and the Java code. The step of using jythonc to compile Jython into Java is clean, yet, it creates a rift in the development process. Code should seamlessly work together without a separate compilation procedure. One should have the ability to seamlessly utilize Jython classes and modules from within a Java application by reference only, and without a special compiler in between. There have been some significant advances in this area, and I will discuss some of those techniques in the next few sections. After seeing these next few concepts in action, you will see why jythonc is no longer needed.

Object Factories

Perhaps the most widely used technique used today for incorporating Jython code within Java applications is the object factory design pattern. This idea basically enables seamless integration between Java and Jython via the use of object factories. There are various different implementations of the logic, but all of them do have the same result in the end. Implementations of the object factory paradigm allow one to include Jython modules within Java applications without the use of an extra compilation step. Moreover, this technique allows for a clean integration of Jython and Java code through usage of Java interfaces. In this section, I will explain the main concept of the object factory technique and then I will show you various implementations.

The key to this design pattern is the creation of a factory method that utilizes `PythonInterpreter` in order to load the desired Jython module. Once the factory has loaded the module via `PythonInterpreter`, it creates a `PyObject` instance of the module. Lastly, the factory coerces the `PyObject` into Java code using the `PyObject __tojava__` method. Overall, the idea is not very difficult to implement and relatively straightforward. However, the different implementations come into play when it comes to passing references for the Jython module and a corresponding Java interface. It is important to note that the factory takes care of instantiating the Jython object and translating it into Java. All work that is performed against the resulting Java object is coded against a corresponding Java interface. This is a great design because it allows us to change the Jython code implementation if we wish without altering the definition contained within the interface. The Java code can be compiled once and we can change the Jython code at will without breaking the application.

Let's take a look at an overview of the entire procedure from a high level. Say that you'd like to use one of your existing Jython modules as an object container within a Java application. Begin by coding a Java interface that contains definitions for those methods contained in the module that you'd like to expose to the Java application. Next, you would modify the Jython module to implement the newly coded Java interface. After this, code a Java factory class that would make the necessary conversions of the module from a `PyObject` into a Java object. Lastly, take the

newly created Java object and use it as you wish. It may sound like a lot of steps in order to perform a simple task, but I think you'll agree that it is not very difficult once you've seen it in action.

Over the next few sections, I will take you through different examples of the implementation. The first example is a simple and elegant approach that involves a one-to-one Jython object and factory mapping. In the second example, we'll take a look at a very loosely coupled approach for working with object factories that basically allows one factory to be used for all Jython objects. Each of these methodologies has its own benefit and you can use the one that works best for you.

One-to-One Jython Object Factories

We will first discuss the notion of creating a separate object factory for each Jython object we wish to use. This one-to-one technique can prove to create lots of boilerplate code, but it has some advantages that we'll take a closer look at later on. In order to utilize a Jython module using this technique, you must either ensure that the .py module is contained within your sys.path, or hard code the path to the module within your Java code. Let's take a look at an example of this technique in use with a Java application that uses a Jython class representing a building.:

```
from org.jython.book.interfaces import BuildingType
```

```
class Building(BuildingType):
    def __init__(self, name, address, id):
        self.name = name
        self.address = address
        self.id = id

    def getBuildingName(self):
        return self.name

    def getBuildingAddress(self):
        return self.address

    def getBuildingId(self):
        return self.id
```

```
package org.jython.book.interfaces;
```

```
public interface BuildingType {

    public String getBuildingName();
    public String getBuildingAddress();
    public String getBuildingId();

}
```

```
package org.jython.book.util;
```

```
import org.jython.book.interfaces.BuildingType;
import org.python.core.PyObject;
import org.python.core.PyString;
import org.python.util.PythonInterpreter;
```

```
public class BuildingFactory {

    private PyObject buildingClass;
```

```
public BuildingFactory() {
    PythonInterpreter interpreter = new PythonInterpreter();
    interpreter.exec("from Building import Building");
    buildingClass = interpreter.get("Building");
}

public BuildingType create(String name, String location, String id) {
    PyObject buildingObject = buildingClass.__call__(new PyString(name),
new PyString(location),
new PyString(id));
    return (BuildingType)buildingObject.__tojava__(BuildingType.class);
}
}

package org.jython.book;

import org.jython.book.util.BuildingFactory;
import org.jython.book.interfaces.BuildingType;

public class Main {

    private static void print(BuildingType building) {
        System.out.println("Building Info: " +
            building.getBuildingId() + " " +
            building.getBuildingName() + " " +
            building.getBuildingAddress());
    }

    public static void main(String[] args) {
        BuildingFactory factory = new BuildingFactory();
        print(factory.create("BUILDING-A", "100 WEST MAIN", "1"));
        print(factory.create("BUILDING-B", "110 WEST MAIN", "2"));
        print(factory.create("BUILDING-C", "120 WEST MAIN", "3"));
    }
}
```

Let's perform a play-by-play analysis of what goes on in this code. We begin with a Jython module named `Building.py` that is placed somewhere on our `sys.path`. Now, we must first ensure that there are no name conflicts before doing so or we could see some quite unexpected results. It is usually a safe bet to place this file at the source root for your application unless you explicitly place the file in your `sys.path` elsewhere. You can see that our `Building.py` object is a simple container for holding building information. We must explicitly implement a Java interface within our Jython class. This will allow the `PythonInterpreter` to coerce our object later. Our second piece of code is the Java interface that we implemented in `Building.py`. As you can see from the code, the returning Jython types are going to be coerced into Java types, so we define our interface methods using the eventual Java types.

The third piece of code in the example above plays the most important role in the game, this is the object factory that will coerce our Jython code into a resulting Java class. In the constructor, a new instance of the `PythonInterpreter` is created which we then utilize the interpreter to obtain a reference to our Jython object and stores it into our `PyObject`. Next, there is a static method named `create` that will be called in order to coerce our Jython object into Java and return the resulting class. It does so by actually performing a `__call__` on the `PyObject` wrapper itself, and as you can see we have the ability to pass parameters to it if we like. The parameters must also be wrapped by `PyObject`s. The coercion takes place when the `__tojava__` method is called on the `PyObject` wrapper. In order to make object implement our Java interface, we must pass the interface `EmployeeType.class` to the `__tojava__` call.

The last bit of provided code, `Main.java`, shows how to make use of our factory. You can see that the factory takes care

of all the heavy lifting and our implementation in `Main.java` is quite small. Simply call the `factory.create()` method to instantiate a new `PyObject` and coerce it into Java.

This procedure for using the object factory design has the benefit of maintaining complete awareness of the Jython object from within Java code. In other words, creating a separate factory for each Jython object allows for the use of passing arguments into the constructor of the Jython object. Since the factory is being designed for a specific Jython object, we can code the `__call__` on the `PyObject` with specific arguments that will be passed into the new constructor of the coerced Jython object. Not only does this allow for passing arguments into the constructor, but also increases the potential for good documentation of the object since the Java developer will know exactly what the new constructor will look like. Although in most cases the developer using this technique will be the person writing the Jython objects as well. The procedures performed in this subsection are probably the most frequently used throughout the Jython community. In the next section, we'll take a look at the same technique applied to a generic object factory that can be used by any Jython object.

Making Use of a Loosely Coupled Object Factory

The object factory design does not have to be implemented using a one to one strategy such as that depicted in the example above. It is possible to design the factory in such a way that it is generic enough to be utilized for any Jython object. This technique allows for less boilerplate coding as you only require one Singleton factory that can be used for all Jython objects, and it also allows for ease of use as you can separate the object factory logic into it's own project and then apply it wherever you'd like. For instance, I've created a project in my environment that basically contains a Jython object factory that can be used in any Java application in order to create Jython objects from Java without worrying about the factory. You can create a similar project or use the one that I've created and linked to the source for this book. In this section we'll take a look at the design behind this project and how it works.

Let's take a look at the same example from above and apply the loosely coupled object factory design. You will notice that this technique forces the Java developer to do a bit more work when creating the object from the factory, but it has the advantage of saving the time that is spent to create a separate factory for each Jython object. You can also see that now we need to code setters into our Jython object and expose them via the Java interface as we can no longer make use of the constructor for passing arguments into the object since the loosely coupled factory makes a generic `__call__` on the `PyObject`:

```
from org.jython.book.interfaces import BuildingType

class Building(BuildingType):
    def __init__(self):
        self.name = None
        self.address = None
        self.id = -1

    def getBuildingName(self):
        return self.name

    def setBuildingName(self, name):
        self.name = name;

    def getBuildingAddress(self):
        return self.address

    def setBuildingAddress(self, address):
        self.address = address

    def getBuildingId(self):
        return self.id

    def setBuildingId(self, id):
```

```
        self.id = id
package org.jython.book.interfaces;

public interface BuildingType {

    public String getBuildingName();
    public String getBuildingAddress();
    public int getBuildingId();
    public void setBuildingName(String name);
    public void setBuildingAddress(String address);
    public void setBuildingId(int id);
}

import java.util.logging.Level;
import java.util.logging.Logger;
import org.python.core.PyObject;
import org.python.util.PythonInterpreter;

public class JythonObjectFactory {
    private static JythonObjectFactory instance = null;
    private static PyObject pyObject = null;

    protected JythonObjectFactory() {

    }

    public static JythonObjectFactory getInstance(){
        if(instance == null){
            instance = new JythonObjectFactory();
        }

        return instance;
    }

    public static Object createObject(Object interfaceType, String moduleName){
        Object javaInt = null;
        PythonInterpreter interpreter = new PythonInterpreter();
        interpreter.exec("from " + moduleName + " import " + moduleName);

        pyObject = interpreter.get(moduleName);

        try {

            PyObject newObj = pyObject.__call__();

            javaInt = newObj.__tojava__(Class.forName(interfaceType.toString().
↪substring(
                interfaceType.toString().indexOf(" ") + 1, interfaceType.toString().
↪length())));
        } catch (ClassNotFoundException ex) {
            Logger.getLogger(JythonObjectFactory.class.getName()).log(Level.SEVERE,
↪null, ex);
        }
    }
}
```

```

        return javaInt;
    }
}

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.jythonbook.interfaces.BuildingType;
import org.jythonbook.factory.JythonObjectFactory;

public class Main {

    public static void main(String[] args) {

        JythonObjectFactory factory = JythonObjectFactory.getInstance();

        BuildingType building = (BuildingType) factory.createObject(
            BuildingType.class, "Building");
        building.setBuildingName("BUILDING-A");
        building.setBuildingAddress("100 MAIN ST.");
        building.setBuildingId(1);
        System.out.println(building.getBuildingId() + " " + building.
↪getBuildingName() + " " +
            building.getBuildingAddress());

    }

}

```

If we follow this paradigm then you can see that our Jython module must be coded a bit differently than it was in our one-to-one example, but not much. The main differences are in the initializer as it no longer takes any arguments, and we also now have coded setter methods into our object. The rest of the concept still holds true in that we must implement a Java interface that will expose those methods we wish to invoke from within our Java application. In this case, we coded the `BuildingType.java` interface and included the necessary setter definitions so that we have a way to load our class with values.

Our next step is to either code a loosely coupled object. If you take a look at the code in the `JythonObjectFactory.java` class you will see that it is a singleton; that is it can only be instantiated one time. The important method to look at is `createObject()` as it does all of the work. As you can see from the code, the `PythonInterpreter` is responsible for obtaining a reference to the Jython object name that we pass as a String value into the method. Once the `PythonInterpreter` has obtained the object and stored it into a `PyObject`, its `__call__()` method is invoked without any parameters. This will retrieve an empty object that is then stored into another `PyObject` that is referenced by `newObj`. Lastly, our newly obtained object is coerced into Java code by calling the `__tojava__()` method which takes the fully qualified name of the Java interface we've implemented with our Jython object. The new Java object is then returned.

Taking a look at the `Main.java` code, you can see that the factory is instantiated or referenced via the use of the `JythonObjectFactory.getInstance()`. Once we have an instance of the factory, the `createObject(Interface, String)` is called passing the interface and a string representation of the module name we wish to use. The code must cast the coerced object using the interface as well. This example assumes that the object resides somewhere on your `sys.path`, otherwise you can use the `createObjectFromPath(Interface, String)` that accepts the string representation for the path to the module we'd like to coerce. This is of course not a preferred technique since it will now include hard-coded paths, but it can be useful to use this technique for testing purposes. Say you've got two Jython modules coded and one of them contains a different object implementation for testing purposes, this technique will allow you to point to the test module.

Another similar, yet, more refined implementation omits the use of `PythonInterpreter` and instead makes use of `PySystemState`. Why would we want another implementation that produces the same results? Well, there are a couple of reasons worth noting. The loosely-coupled object factory design I described in the beginning of this section instantiates the `PythonInterpreter` and then makes calls against it. This can cause a decrease in performance, as it is quite expensive to use the interpreter. On the other hand, we can make use of `PySystemState` and save ourselves the trouble of incurring extra overhead making calls to the interpreter. Not only does the next example show how to utilize this technique, but it also shows how we can make calls upon the coerced object and pass arguments at the same time. This is a limitation of the object factory we showed previously.:

```
JythonObjectFactory.java

package org.jython.book.util;

import org.python.core.Py;

import org.python.core.PyObject;
import org.python.core.PySystemState;

public class JythonObjectFactory {

    private final Class interfaceType;
    private final PyObject klass;

    // likely want to reuse PySystemState in some clever fashion since expensive to
    ↪ setup...
    public JythonObjectFactory(PySystemState state, Class interfaceType, String
    ↪ moduleName, String className) {
        this.interfaceType = interfaceType;
        PyObject importer = state.getBuiltins().__getitem__(Py.newString("__import__
    ↪"));
        PyObject module = importer.__call__(Py.newString(moduleName));
        klass = module.__getattr__(className);
        System.err.println("module=" + module + ",class=" + klass);
    }

    public JythonObjectFactory(Class interfaceType, String moduleName, String
    ↪ className) {
        this(new PySystemState(), interfaceType, moduleName, className);
    }

    public Object createObject() {
        return klass.__call__().__tojava__(interfaceType);
    }

    public Object createObject(Object arg1) {
        return klass.__call__(Py.java2py(arg1)).__tojava__(interfaceType);
    }

    public Object createObject(Object arg1, Object arg2) {
        return klass.__call__(Py.java2py(arg1), Py.java2py(arg2)).__tojava__
    ↪ (interfaceType);
    }

    public Object createObject(Object arg1, Object arg2, Object arg3) {
        return klass.__call__(Py.java2py(arg1), Py.java2py(arg2), Py.java2py(arg3)).
    ↪ __tojava__(interfaceType);
    }
}
```

```

    public Object createObject(Object args[], String keywords[]) {
        PyObject convertedArgs[] = new PyObject[args.length];
        for (int i = 0; i < args.length; i++) {
            convertedArgs[i] = Py.java2py(args[i]);
        }
        return klass.__call__(convertedArgs, keywords).__tojava__(interfaceType);
    }

    public Object createObject(Object... args) {
        return createObject(args, Py.NoKeywords);
    }
}

import org.jython.book.interfaces.BuildingType;
import org.jython.book.util.JythonObjectFactory;

public class Main{

    public static void main(String args[]) {
        // what other control options should we provide to the factory?
        // jsr223 might have some good ideas, but also let's keep some simple code_
↪usage
        // for now, let's just try out and refine
        JythonObjectFactory factory = new JythonObjectFactory(
            BuildingType.class, "building", "Building");

        BuildingType building = (BuildingType) factory.createObject();

        building.setBuildingName("BUIDING-A");
        building.setBuildingAddress("100 MAIN ST.");
        building.setBuildingId(1);

        System.out.println(building.getBuildingId() + " " + building.
↪getBuildingName() + " " +
            building.getBuildingAddress());
    }
}

```

As you can see from the code above, it has quite a few differences from the object factory implementation shown previously. First, you can see that the instantiation of the object factory requires some different arguments. In this case, we pass in the interface, module, and class name. Next, you can see that the `PySystemState` obtains a reference to the importer `PyObject`. The importer then makes a `__call__` to the module we've requested. By the way, the requested module must be contained somewhere on the `sys.path`. Lastly, we obtain a reference to our class by calling the `__getattr__` method on the module. We can now use the returned class to perform the coercion of our Jython object into Java. As mentioned previously, you'll note that this particular implementation includes several `createObject()` variations allowing one to pass arguments to the module when it is being called. This, in effect, gives us the ability to pass arguments into the initializer of the Jython object.

Which object factory is best? Your choice, depending upon the situation your application is encountering. Bottom line is that there are several ways to perform the object factory design and they all allow seamless use of Jython objects from within Java code.

Now that we have a coerced Jython object, we can go ahead and utilize the methods that have been defined in the Java interface. As you can see, the simple example above sets a few values and then prints the object values out. Hopefully you can see how easy it is to create a single object factory that can be used for any Jython object rather than just one.

Returning `__doc__` Strings

It is also very easy to obtain the `__doc__` string from any of your Jython classes by coding an accessor method on the object itself. We'll add some code to the building object that was used in the previous examples. It doesn't matter what type of factory you decide to work with, this trick will work with both.:

```
from org.jython.book.interfaces import BuildingType

class Building(BuildingType):
    ''' Class to hold building objects '''

    def __init__(self):
        self.name = None
        self.address = None
        self.id = -1

    def getBuildingName(self):
        return self.name

    def setBuildingName(self, name):
        self.name = name;

    def getBuildingAddress(self):
        return self.address

    def setBuildingAddress(self, address):
        self.address = address

    def getBuildingId(self):
        return self.id

    def setBuildingId(self, id):
        self.id = id

    def getDoc(self):
        return self.__doc__

package org.jython.book.interfaces;

public interface BuildingType {

    public String getBuildingName();
    public String getBuildingAddress();
    public int getBuildingId();
    public void setBuildingName(String name);
    public void setBuildingAddress(String address);
    public void setBuildingId(int id);
    public String getDoc();

}

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.jython.book.interfaces.BuildingType;
import org.plyjy.factory.JythonObjectFactory;
```

```

public class Main {

    public static void main(String[] args) {

        JythonObjectFactory factory = JythonObjectFactory.getInstance();
        BuildingType building = (BuildingType) factory.createObject(
            BuildingType.class, "Building");
        building.setBuildingName("BUIDING-A");
        building.setBuildingAddress("100 MAIN ST.");
        building.setBuildingId(1);
        System.out.println(building.getBuildingId() + " " + building.
↪getBuildingName() + " " +
            building.getBuildingAddress());
        System.out.println(building.getDoc());

    }
}

1 BUIDING-A 100 MAIN ST.
Class to hold building objects

```

Applying the Design to Different Object Types

This design will work with all object types, not just plain old Jython objects. In the following example, the Jython module is a class containing a simple calculator method. The factory coercion works the same way, the result is a Jython class that is converted into Java.:

```

from org.jython.book.interfaces import CostCalculatorType

class CostCalculator(CostCalculatorType, object):
    ''' Cost Calculator Utility '''

    def __init__(self):
        print 'Initializing'
        pass

    def calculateCost(self, salePrice, tax):
        return salePrice + (salePrice * tax)

package org.jython.book.interfaces;

public interface CostCalculatorType {

    public double calculateCost(double salePrice, double tax);

}

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.plyjy.factory.JythonObjectFactory;

public class Main {

    public static void main(String[] args) {

```

```
JythonObjectFactory factory = JythonObjectFactory.getInstance();
CostCalculatorType costCalc = (CostCalculatorType) factory.createObject(
    CostCalculatorType.class, "CostCalculator");
System.out.println(costCalc.calculateCost(25.96, .07));

}
}
```

Initializing
27.7772

We can also use Python properties in our Jython objects along with Java. As a matter of fact, the Java interface is coded with the same definitions as it would be for a regular accessor-style object. In this example, we'll populate a Jython object that contains automobile properties. Although the automobile object in this example is not very life-like as it does not contain make, model, year, etc. for various legal reasons, I think you'll get the idea.:

```
from org.jython.book.interfaces import AutomobileType

class Automobile(AutomobileType, object):
    ''' Bean to hold automobile objects '''

    def __init__(self):
        print 'Initializing Now'
        pass

    def setType(self, value):
        self.__type = value
        print 'setting object: ' + value

    def getType(self):
        print 'getting object'
        return self.__type

    type = property(fget=getType,
                    fset=setType,
                    doc="The Type of the Automobile.")

    def setColor(self, value):
        self.__color = value

    def getColor(self):
        return self.__color

    color = property(fget=getColor,
                     fset=setColor,
                     doc="Color of Automobile.")

    def getDoc(self):
        return self.__doc__

package org.jython.book.interfaces;

public interface AutomobileType {
    public String getType();
    public void setType(String value);
    public String getColor();
    public void setColor(String value);
    public String getDoc();
}
```



```

}

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.jython.book.interfaces.AutomobileType;
import org.plyjy.factory.JythonObjectFactory;

public class Main {

    private static void print(AutomobileType auto) {
        System.out.println("Doc: " + auto.getDoc());
        System.out.println("Type: " + auto.getType());
        System.out.println("Color: " + auto.getColor());
    }

    public static void main(String[] args) {

        JythonObjectFactory factory = JythonObjectFactory.getInstance();
        AutomobileType automobile = (AutomobileType) factory.createObject(
            AutomobileType.class, "Automobile");
        automobile.setType("Sport");
        automobile.setColor("red");
        print (automobile);
    }
}

```

Initializing Now
 setting object: Sport
 Doc: Bean to hold automobile objects
 getting object
 Type: Sport
 Color: red

JSR-223

Along with the release of Java SE 6 came a new advantage for dynamic languages on the JVM. JSR-223 enables dynamic languages to be callable via Java in a seamless manner. Although this method of accessing Jython code is not quite as flexible as using an object factory, it is quite useful for running short Jython scripts from within Java code. The scripting project (<https://scripting.dev.java.net/>) contains many engines that can be used to run different languages within Java. In order to run the Jython engine, you must obtain jython-engine.jar from the scripting project and place it into your classpath. You must also place jython.jar in the classpath, and it does not yet function with Jython 2.5 so Jython 2.2.1 must be used.

Below is a small example showing the utilization of the scripting engine.

```

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class Main {

```

```
public static void main(String[] args) throws ScriptException {
    ScriptEngine engine = new ScriptEngineManager().getEngineByName("python");
    engine.eval("import sys");
    engine.eval("print sys");
    engine.put("a", 42);
    engine.eval("print a");
    engine.eval("x = 2 + 2");
    Object x = engine.get("x");
    System.out.println("x: " + x);
}

}

*sys-package-mgr*: processing new jar, '/jsr223-engines/jython/build/jython-engine.jar
↪'
*sys-package-mgr*: processing modified jar, '/System/Library/Java/Extensions/QTJava.
↪zip'
sys module
42
x: 4
```

Utilizing PythonInterpreter

A similar technique to JSR-223 for embedding Jython is making use of the `PythonInterpreter` directly. This style of embedding code is very similar to making use of a scripting engine, but it has the advantage of working with Jython 2.5. Another advantage is that the `PythonInterpreter` enables you to make use of `PyObject`s directly. In order to make use of the `PythonInterpreter` technique, you only need to have `jython.jar` in your classpath, there is no need to have an extra engine involved.:

```
import org.python.core.PyException;
import org.python.core.PyInteger;
import org.python.core.PyObject;
import org.python.util.PythonInterpreter;
public class Main {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws PyException {
        PythonInterpreter interp = new PythonInterpreter();
        interp.exec("import sys");
        interp.exec("print sys");
        interp.set("a", new PyInteger(42));
        interp.exec("print a");
        interp.exec("x = 2+2");
        PyObject x = interp.get("x");
        System.out.println("x: " + x);
    }
}

<module 'sys' (built-in)>
42
x: 4
```

Using Java within Jython Applications

Making use of Java from within Jython applications is about as seamless as using external Jython modules within a Jython script. You simply import the required Java classes and use them directly. Java classes can be called in the same fashion as Jython classes, and the same goes for method calling. You simply call a class method and pass parameters the same way you'd do in Python.

Type coercion occurs much as it does when using Jython in Java in order to seamlessly integrate the two languages. In the following table, you will see the Java types that are coerced into Python types and how they match up. This table was taken from the Jython user guide.

Java Type			Python Type	
char				String(length of 1)
boolean				Integer(true = not zero)
byte, short, int, long		Integer		
java.lang.String, byte[], char[]	String			
java.lang.Class			JavaClass	
Foo[]				Array(containing objects of class or subclass of Foo)
java.lang.Object			String	
orb.python.core.PyObject	Un-changed			
Foo				JavaInstance representing Java class Foo

Table 10.1- Python and Java Types

Another thing to note about the utilization of Java within Jython is that there may be some naming conflicts that arise. If a Java object conflicts with a Python object name, then you can simply fully qualify the Java object in order to ensure that the conflict is resolved.

In the next couple of examples, you will see some Java objects being imported and used from within Jython.

```
>>> from java.lang import Math
>>> Math.max(4, 7)
7L
>>> Math.pow(10,5)
100000.0
>>> Math.round(8.75)
9L
>>> Math.abs(9.765)
9.765
>>> Math.abs(-9.765)
9.765
>>> from java.lang import System as javasystem
>>> javasystem.out.println("Hello")
Hello
```

Now let's create a Java object and use it from within a Jython application.:

```
public class Beach {
    private String name;
    private String city;
    private String state;
    private boolean publicBeach;
```

```
public Beach(String name, String city, String state, boolean publicBeach){
    this.name = name;
    this.city = city;
    this.state = state;
    this.publicBeach = publicBeach;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}

public String getState() {
    return state;
}

public void setState(String state) {
    this.state = state;
}

public boolean isPublicBeach() {
    return publicBeach;
}

public void setPublicBeach(boolean publicBeach) {
    this.publicBeach = publicBeach;
}
}

>>> import Beach
>>> beach = Beach("Cocoa Beach", "Cocoa Beach", "FL", True)
>>> beach.getName()
u'Cocoa Beach'
>>> print beach.getName()
Cocoa Beach
>>> print beach.isPublicBeach()
True
```

One thing you'll need to do is ensure that the Java class you wish to use resides within your CLASSPATH. In the example above, I created a JAR file that contained the Beach class and then put that JAR on the CLASSPATH.

It is also possible to extend Java classes via Jython classes. This allows us to extend the functionality of a given Java class using Jython objects, which can be quite helpful at times. The next example shows a Jython class extending a Java class that includes some calculation functionality. The Jython class then adds another calculation method and

makes use of the calculation methods from both the Java class and the Jython class.:

```
public class Calculator {

    public Calculator(){

    }

    public double calculateTip(double cost, double tipPercentage){
        return cost * tipPercentage;
    }

    public double calculateTax(double cost, double taxPercentage){
        return cost * taxPercentage;
    }

}

import Calculator
from java.lang import Math

class JythonCalc(Calculator):
    def __init__(self):
        pass

    def calculateTotal(self, cost, tip_and_tax):
        return cost + tip_and_tax

if __name__ == "__main__":
    calc = JythonCalc()
    cost = 23.75
    tip = .15
    tax = .07
    print "Starting Cost: ", cost
    print "Tip Percentage: ", tip
    print "Tax Percentage: ", tax
    print Math.round(calc.calculateTotal(cost,
        (calc.calculateTip(cost, .15) + calc.calculateTax(cost, .07))))

Starting Cost:  23.75
Tip Percentage:  0.15
Tax Percentage:  0.07
29
```

Conclusion

Integrating Jython and Java is really at the heart of the language. Using Java within Jython works just as adding other Jython modules, a very seamless integration. What makes this nice is that now we can utilize the full set of libraries and APIs available to Java from our Jython applications. Having the ability of using Java within Jython also provides the advantage of writing Java code in the Python syntax... something we all enjoy.

Utilizing design patterns such as the Jython object factory, we can also harness our Jython code from within Java applications. Although jythonc is no longer part of the Jython distribution, we can still effectively use Jython from within Java. There are many examples of the object factory available, as well as projects such as PlyJy (<http://kenai.com/projects/plyjy/>).

[com/projects/plyjy](#)) that give the ability to use object factories by simply including a JAR in your Java application.

There are more ways to use Jython from within Java as well. The Java language added scripting language support with JSR-223 with the release of Java 6. Using a jython engine, we can make use of the JSR-223 dialect to sprinkle Jython code into our Java applications. Similarly, the `PythonInterpreter` can be used from within Java code to invoke Jython. Also keep an eye on projects such as Clamp (<http://github.com/groves/clamp/tree/master>) that are on the horizon. The Clamp project has the goal to make use of annotations in order to create Java classes from Jython classes. It will be exciting to see where this project goes, and it will be documented once the project has been completed.

In the next chapter, you will see how we can use Jython within some integrated development environments. Specifically, we will take a look at developing Jython with Eclipse and Netbeans. Utilizing an IDE can greatly increase developer productivity, and also assist in subtleties such as adding modules and JAR files to the classpath.

Chapter 11: Using Jython in an IDE

In this chapter, we will discuss developing Jython applications using two of the most popular integrated development environments, Eclipse and Netbeans. There are many other development environments available for Python and Jython today, however, these two are perhaps the most popular and contain the most Jython-specific tools. Eclipse has had a plugin known as PyDev for a number of years, and this plugin provides rich support for developing and maintaining Python and Jython applications alike. Netbeans began to include Python and Jython support with version 6.5 and beyond. The Netbeans IDE also provides rich support for development and maintenance of Python and Jython applications.

Please note that in this chapter we will refer to Python/Jython as Jython. All of the IDE options discussed are available for both Python and Jython unless otherwise noted. For readability and consistency sake, we'll not refer to both Python and Jython throughout this chapter unless there is some feature that is not available for Python or Jython specifically. Also note that we will call the plugins discussed by their names, so in the case of Netbeans the plugin is called *Netbeans Python Plugin*. This plugin works with both Python and Jython in all cases.

Eclipse

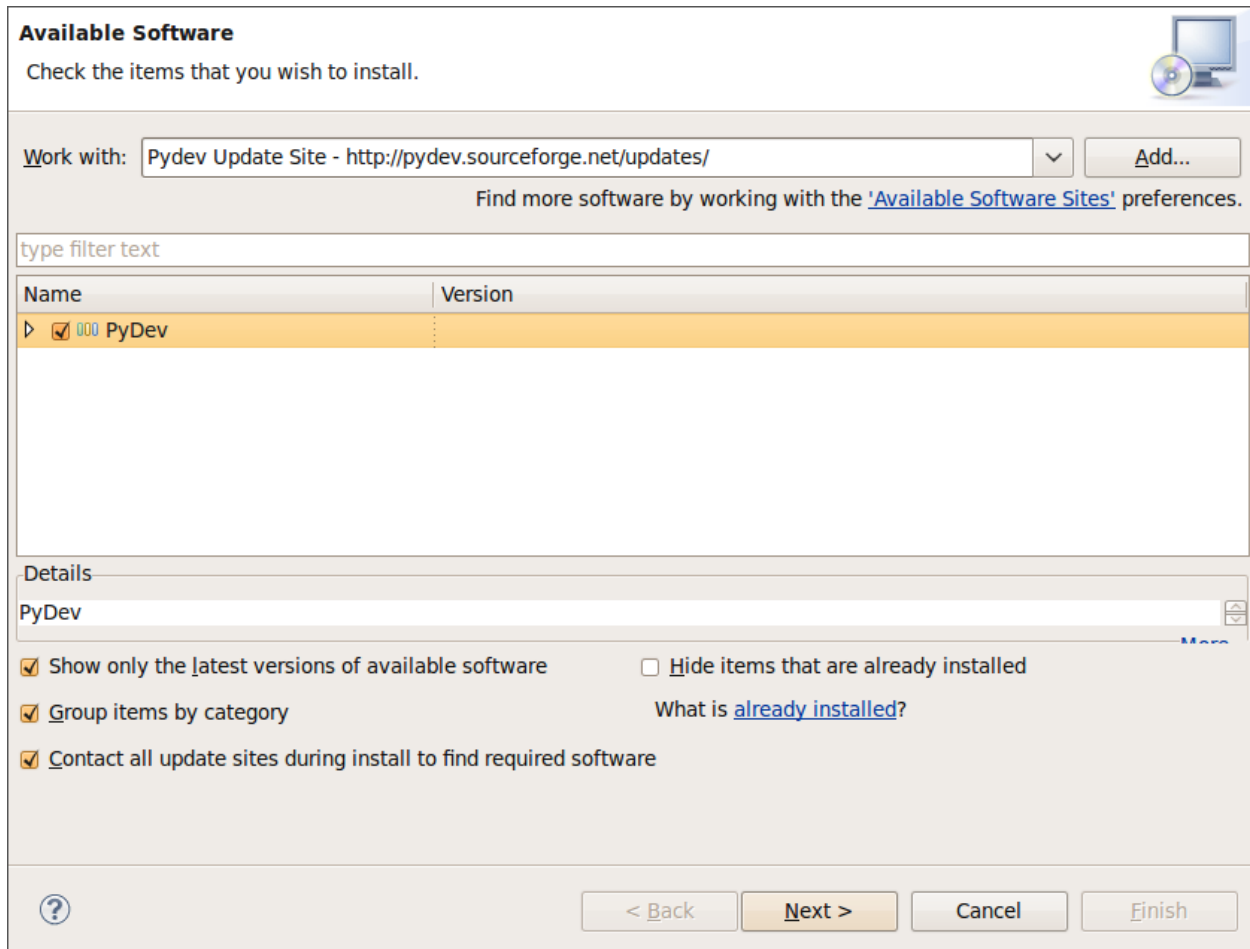
Naturally, you will need to have Eclipse installed on your machine to use Jython with it. The latest available version when this book is being written is Eclipse 3.5 (also known as Eclipse Galileo) and it is the recommended version to use to follow this section. Versions 3.2, 3.3 and 3.4 will work too, although there will be minor user interface differences which may confuse you while following this section.

If you don't have Eclipse installed on your machine, go to <http://www.eclipse.org/downloads/> and download the version for Java developers.

Installing PyDev

Eclipse doesn't include Jython support built-in. Thus, we will use PyDev, an excellent plugin which adds support for the Python language and includes specialized support for Jython. PyDev's home page is <http://pydev.sourceforge.net/> but you won't need to manually download and install it.

To install the plugin, start Eclipse and go to the menu *Help* → *Install new Software...*, type <http://pydev.sourceforge.net/updates/> into the "Work with" input box and press enter. After a short moment you will see an entry for PyDev in the bigger box below. Just select it, clicking on the checkbox which appears at the left of "PyDev" (see the image which follows, as reference) and finally click the "Next" button.



After this, just follow the wizard, accept the license agreement and then click the “Finish” button.

Once the plugin has been installed by Eclipse, you will be asked if you want to restart the IDE to enable the plugin. As that is the recommended option, do so, answering “Yes” to the dialog. Once Eclipse reboots itself, you will enjoy full Python support on the IDE.

Minimal Configuration

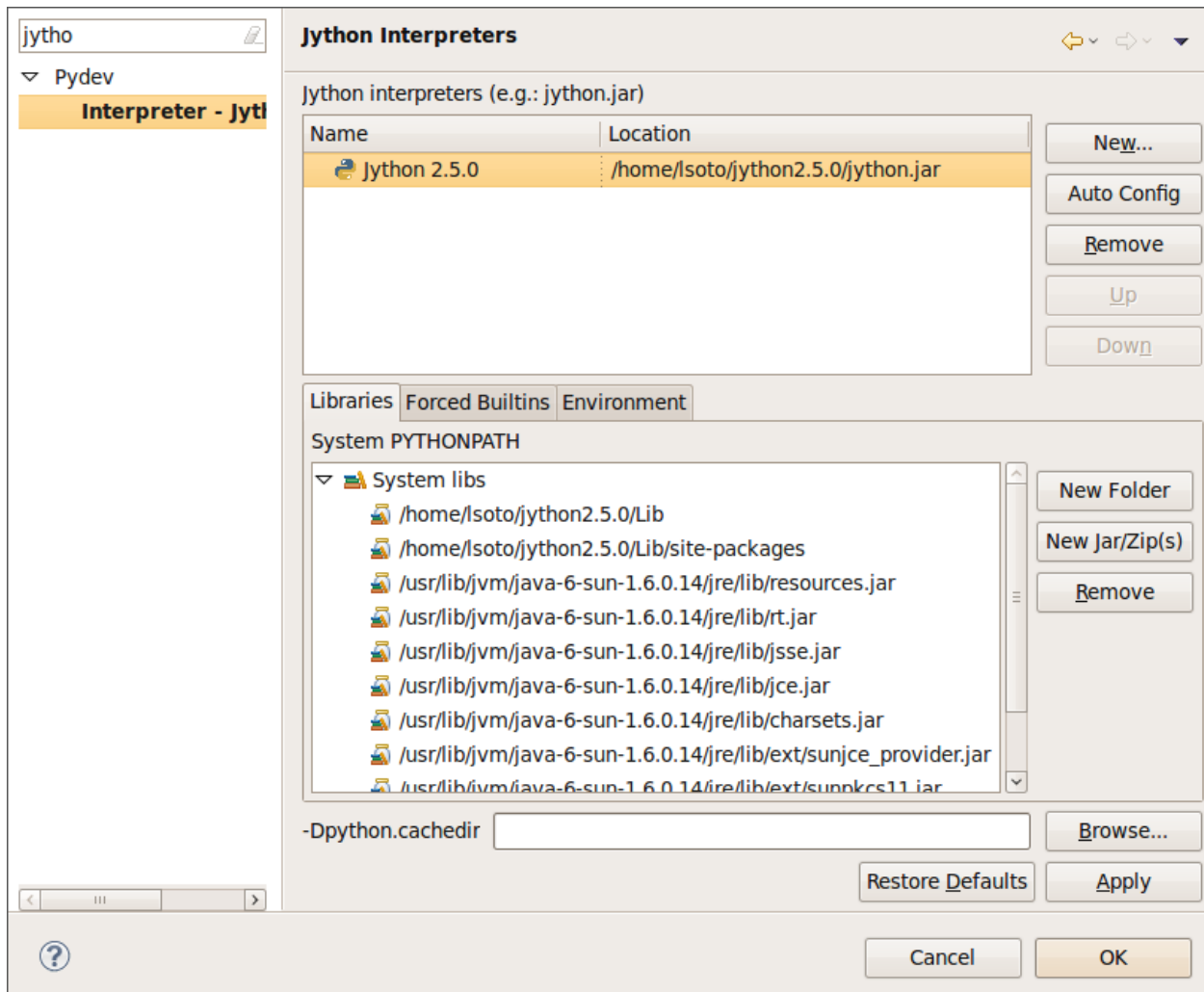
Before starting a PyDev project you must tell PyDev which Python interpreters are available. In this context, a interpreter is just a particular installation of some implementation of Python. When starting you will normally only need one interpreter and for this chapter we will only use Jython 2.5.0. To configure it, open the Eclipse Preferences dialog (via *Window* → *Preferences* in the main menu bar). On the text box located at the top of the left panel, type “Jython”. This will filter the myriad of Eclipse (and PyDev!) options and will present us with a much simplified view, in which you will spot the “Interpreter - Jython” section on the left.

Once you selected the “Interpreter - Jython” section, you will be presented with an empty list of Jython interpreters at the top of the right side. We clearly need to fix that! So, click the “New...” button, enter “Jython 2.5.0” as the “Interpreter Name”, click the “Browse...” button and find the `jython.jar` inside your Jython 2.5.0 installation.

Note: Even if this is the only runtime we will use on this chapter, I recommend you to use a naming schema like the one proposed here, including both the implementation name (e.g.: “Jython”) and the full version (e.g.: “2.5.0”) on the interpreter name. This will avoid confusion and name clashing when adding new interpreters in the future.

After selecting the `jython.jar` file, PyDev will automatically detect the default, *global* `sys.path` entries. PyDev always infer the right values, so unless you have very special needs, just accept the default selection and click “OK”.

If all has gone well, you will now see an entry on the list of Jython interpreters, representing the information you just entered. It will be similar to the following picture (of course, your filesystem paths will differ):



That’s all. Click “OK” and you will be ready to develop with Jython while enjoying the support provided by a modern IDE.

If you are curious, you may want to explore the other options found on the “Preferences” window, below the “PyDev” section (after clearing the search filter we used to quickly go to the Jython interpreter configuration). But in my experience, it’s rarely needed to change most of the other options available.

In the next sections we will take a look to the more important PyDev features to have a more pleasant learning experience and make you more productive.

Hello PyDev!: Creating Projects and Executing Modules

Once you see the first piece of example code on this chapter, it may seem over simplistic. It is, indeed, a very dumb example. The point is to keep the focus on the basic steps you will perform for the lifecycle of any Python-based project inside the Eclipse IDE, and which will apply on simple and complex projects. So, as you probably guessed it, our first project will be a dumb “Hello World”. Let’s start it!

Go to *File* → *New* → *Project...*. You will be presented with a potentially long list with all the kind of projects you can create with Eclipse. Select “PyDev Project”, under the “PyDev” group (you can also use the filter text box at the top and just type “PyDev Project” if it’s faster for you).

The next dialog will ask you for your project properties. As the “Project name”, we will use “LearningPyDev”. On “Project contents”, we will let checked the “Use default” checkbox, so PyDev will create a directory with the same name as the project inside the Eclipse workspace (which is the root path of your eclipse projects). Since we are using Jython 2.5.0, we will change the “Project type” to “Jython” and the “Grammar Version” to “2.5”. We will let alone the “Interpreter”, which will default to the Jython interpreter we just defined on the [Minimal Configuration](#) section. We will also left checked the “Create default ‘src’ folder and add it to the pythonpath” option since it’s a common convention on Eclipse projects.

After clicking “Finish” PyDev will create your project, which will only contain an empty `src` directory and a reference to the interpreter being used. Let’s create our program now!

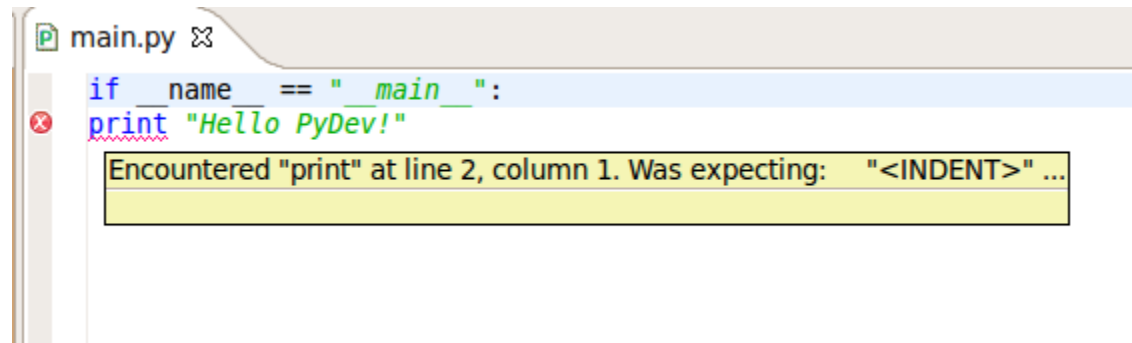
Right click on the project, and select *New* → *PyDev Module*. Let the “Package” blank and enter “main” as the “Name”. PyDev offers some templates to speed up the creation of new modules, but we won’t use them, as our needs are rather humble. So let the “Template” as empty and click “Finish”.

PyDev will present you an editor for the `main.py` file it just created. It’s time to implement our program. Write the following code at the editor:

```
if __name__ == "__main__":
    print "Hello PyDev!"
```

And then press `Ctrl + F11` to run this program. Select “Jython Run” from the dialog presented and click OK. The program will run and the text “Hello PyDev!” will appear on the console, located on the bottom area of the IDE.

If you manually typed the program, you probably noted that the IDE knows that in Python a line ending in “:” marks the start of a block and will automatically put your cursor at the appropriate level of indentation in the next line. See what happens if you manually override this decision and put the print statement at the same indentation level of the if statement and save the file. The IDE will highlight the line flagging the error. If you hover at the error mark, you will see the explanation of the error, as seen in the image:



Expect the same kind of feedback for whatever syntax error you made. It helps to avoid the frustration of going on edit-run loops only to find further minor syntax errors.

Passing Command-line Arguments and Customizing Execution

Command line arguments may seem old-fashioned, but are actually a very simple and effective way to let programs interact with the outside. Since you have learned to use Jython as a scripting language, it won’t be uncommon to write scripts which will take its input from the command line (note that for unattended execution, reading input from the command line is way more convenient than obtaining data from the standard input, let alone using a GUI).

As you have probably guessed, we will make our toy program to take a command line argument. The argument will represent the name of the user to greet, to build a more personalized solution. Here is how our `main.py` should look

like:

```
import sys
if __name__ == "__main__":
    if len(sys.argv) < 2:
        print "Sorry, I can't greet you if you don't say your name"
    else:
        print "Hello %s!" % sys.argv[1]
```

If you hit `Ctrl + F11` again, you will see the “Sorry I can’t greet you...” message on the console. It makes sense, since you didn’t pass the name. Not to say that it was your fault, as you didn’t have any chance to say your name either.

To specify command line arguments, go to the *Run → Run Configurations...* menu, and you will find an entry named “LearningPyDev main.py” under the “Jython Run” section in the left. It will probably be already selected, but if it’s not, select it manually. Then, on the main section of the dialog you will find ways to customize the execution of our script. You can change aspects like the current directory, pass special argument to the JVM, change the interpreter to use, set environment variables, among others. We just need to specify an argument so let’s type “Bob” on the “Program arguments” box and click the “Run” button.

As you’d expect, the program now prints “Hello Bob!” on the console. Note that the value you entered is remembered, that is, if you press `Ctrl + F11` now, the program will print “Hello Bob!” again. Some people may point out that this behavior makes testing this kind of programs very awkward, since the “Run Configurations” dialog will have to be opened each time the arguments need to be changed. But if we really want to test our programs (which *is* a good idea), we should do it in the right way. We will look into that soon, but first let’s finish our tour on basic IDE features.

Playing with the Editor

Let’s extend our example code a bit more, providing different ways to greet our users, in different languages. We will use the `optparse` to process the arguments this time. Refer to Chapter 8 if you want to remember how to use `optparse`. We will also use decorators (seen in Chapter 6) to make it trivial to extend our program with new ways to greet our users. So, our little `main.py` has grown a bit now:

```
# -*- coding: utf-8 -*-
import sys
from optparse import OptionParser

greetings = dict(en=u'Hello %s!',
                 es=u'Hola %s!',
                 fr=u'Bonjour %s!',
                 pt=u'Alò %s!')

uis = {}
def register_ui(ui_name):
    def decorator(f):
        uis[ui_name] = f
        return f
    return decorator

def message(ui, msg):
    if ui in uis:
        uis[ui](msg)
    else:
        raise ValueError("No greeter named %s" % ui)

def list_uis():
    return uis.keys()
```

```

@register_ui('console')
def print_message(msg):
    print msg

@register_ui('window')
def show_message_as_window(msg):
    from javax.swing import JFrame, JLabel
    frame = JFrame(msg,
                    defaultCloseOperation=JFrame.EXIT_ON_CLOSE,
                    size=(100, 100),
                    visible=True)
    frame.contentPane.add(JLabel(msg))

if __name__ == "__main__":
    parser = OptionParser()
    parser.add_option('--ui', dest='ui', default='console',
                    help="Sets the UI to use to greet the user. One of: %s" %
                    ", ".join("%s" % ui for ui in list_uis()))
    parser.add_option('--lang', dest='lang', default='en',
                    help="Sets the language to use")
    options, args = parser.parse_args(sys.argv)
    if len(args) < 2:
        print "Sorry, I can't greet you if you don't say your name"
        sys.exit(1)

    if options.lang not in greetings:
        print "Sorry, I don't speak '%s'" % options.lang
        sys.exit(1)

    msg = greetings[options.lang] % args[1]

    try:
        message(options.ui, msg)
    except ValueError, e:
        print "Invalid UI name\n"
        print "Valid UIs:\n\n" + "\n".join(' * ' + ui for ui in list_uis())
        sys.exit(1)

```

Take a little time to play with this code in the editor. Try pressing Ctrl + Space, which is the shortcut for automatic code completion (also known as “Intellisense” on Microsoft’s parlance) on different locations. It will provide completion for import statements (try completing that line just after the `import` token, or in the middle of the `OptionParser` token) and attribute or method access (like on `sys.exit` or `parser.add_option` or even in `JFrame.EXIT_ON_CLOSE` which is accessing a Java class!). It also provides hints about the parameters in the case of methods.

In general, every time you type a dot, the automatic completion list will pop out, if the IDE knows enough about the symbol you just typed to provide help. But you can also call for help at any given point. For example, go to the bottom of the code and type `message(`. Suppose you just forgot the order of the parameters to that function. Solution: Press Ctrl + Space and PyDev will “complete” the statement, using the name of the formal parameters of the function.

Also try Ctrl + Space on keywords like `def`. PyDev will provide you little templates which may save you some typing. You can customize the templates on the *PyDev* → *Editor* → *Templates* section of the Eclipse Preferences window (available on the *Window* → *Preferences* main menu).

The other thing you may have noted now that we have a more sizable program with some imports, functions and global variables is the “Outline” panel in the right side of the IDE window shows a tree-structure view of code being edited showing such features. It also displays classes, by the way.

And don't forget to run the code! Of course, it's not much spectacular to see that after pressing `Ctrl + F11` we still get the same boring "Hello Bob!" text on the console. But if you edit the command line argument (as seen recently, via the "Run Configurations..." dialog) to the following: `Bob --lang es --ui window`, you will get a nice window greeting Bob in Spanish. Also see what happens if you specify a non supported UI (say, `--ui speech`) or a unsupported language. We even support the `--help`! So we have a generic, polyglot greeter which also happens to be reasonably robust and user friendly (for command line program standards, that is).

At this point you are probably tired of manually testing the program editing the command line argument on that dialog. Just one more extra section and we will get into a better way to test our program using the IDE. Actually, part of the next section will help us towards the solution.

A Bit of Structure: Packages, Modules and Navigation

If you like simplicity you may be asking (or swearing, depending on your character) why I over-engineered the last example. There are simpler (in the sense of a more concise and understandable code) solutions to the same problem statement. But I needed to grow the our toy code to explore another aspect of IDEs, which for some people are a big reason to use them: Organizing complex code bases. And you don't expect me to put a full blown Pet Store example on this book to get to that point, do you? ;-)

So, let's suppose that the complications I introduced (mainly the registry of UIs exposed via decorators) are perfectly justified, because we are working on a slightly complicated problem. In other words: Let's extrapolate.

The point is, we know that the great majority of our projects can't be confined to just one file (i.e., one python module). Even our very dumb example is starting to get unpleasant to read. And, when we realize that we need more than one module, we also realize we need to group our modules under a common umbrella, to keep it clear that our modules form a coherent thing together and to lower the chances of name clashing with other projects. So, as seen on Chapter 7, the Python solution to this problem are modules and packages.

Our plan is to organize the code as follows. Everything will go under the package `hello`. The core logic, including the language support, will go into the package itself (i.e., into its `__init__.py` file) and each UI will go into its own module under the `hello` package. The `main.py` script will remain as the command line entry point.

Right click on the project and select *New → PyDev Package*. Enter "hello" as the "Name" and click "Finish". PyDev will create the package and open an editor for its `__init__.py` file. As I said, we will move the core logic to this package, so this file will contain the following code, extracted from our previous version of the main code:

```
# -*- coding: utf-8 -*-
greetings = dict(en=u'Hello %s!',
                 es=u'Hola %s!',
                 fr=u'Bonjour %s!',
                 pt=u'Alô %s!')

class LanguageNotSupportedException(ValueError):
    pass

class UINotSupportedExeption(ValueError):
    pass

uis = {}
def register_ui(ui_name):
    def decorator(f):
        uis[ui_name] = f
        return f
    return decorator

def message(ui, msg):
    '''
    Displays the message `msg` via the specified UI which has to be
```

```

    previously registered.
    '''
    if ui in uis:
        uis[ui](msg)
    else:
        raise UINotSupportedException(ui)

def list_uis():
    return uis.keys()

def greet(name, lang, ui):
    '''
    Greets the person called `name` using the language `lang` via the
    specified UI which has to be previously registered.
    '''
    if lang not in greetings:
        raise LanguageNotSupportedException(lang)
    message(ui, greetings[lang] % name)

```

Note that I embraced the idea of modularizing our code, providing exceptions to notify clients of problems when calling the greeter, instead of directly printing messages on the standard output.

Now we will create the `hello.console` module containing the console UI. Right click on the project, select *New* → *PyDev Module*, Enter “hello” as the “Package” and “console” as the “Name”. You can avoid to type the package name if you right click on the package instead of the project. Click “Finish” and copy the `print_message` function there:

```

from hello import register_ui

@register_ui('console')
def print_message(msg):
    print msg

```

Likewise, create the `window` module inside the `hello` package, and put there the code for `show_message_as_window`:

```

from javax.swing import JFrame, JLabel
from hello import register_ui

@register_ui('window')
def show_message_as_window(msg):
    frame = JFrame(msg,
                    defaultCloseOperation=JFrame.EXIT_ON_CLOSE,
                    size=(100, 100),
                    visible=True)
    frame.contentPane.add(JLabel(msg))

```

Finally, the code for our old `main.py` is slightly reshaped into the following:

```

import sys
import hello, hello.console, hello.window
from optparse import OptionParser

def main(args):
    parser = OptionParser()
    parser.add_option('--ui', dest='ui', default='console',
                    help="Sets the UI to use to greet the user. One of: %s" %
                    ", ".join("%s" % ui for ui in list_uis()))

```

```
parser.add_option('--lang', dest='lang', default='en',
                  help="Sets the language to use")
options, args = parser.parse_args(args)
if len(args) < 2:
    print "Sorry, I can't greet you if you don't say your name"
    return 1
try:
    hello.greet(args[1], options.lang, options.ui)
except hello.LanguageNotSupportedException:
    print "Sorry, I don't speak '%s'" % options.lang
    return 1
except hello.UINotSupportedExeption:
    print "Invalid UI name\n"
    print "Valid UIs:\n\n" + "\n".join(' * ' + ui for ui in hello.list_uis())
    return 1
return 0

if __name__ == "__main__":
    main(sys.argv)
```

Tip: Until now, we have used PyDev’s wizards to create new modules and packages. But, as you saw on Chapter 7, modules are just files with the `.py` extension located on the `sys.path` or inside packages, and packages are just directories that happen to contain a `__init__.py` file. So you may want to create modules using *New → File* and packages using *New → Folder* if you don’t like the wizards.

Now we have our code split over many files. On a small project navigating through it using the left-side project tree (called the “PyDev Package Explorer”) isn’t difficult, but you can imagine that on a project with dozens of files it will be difficult. So we will see some ways to ease the navigation of a code base.

First, let’s suppose you are reading `main.py` and want to jump to the definition of the `hello.greet` function, called on the line 17. Instead of manually changing to such file and scanning until finding the function, just press `Ctrl` and click `greet`. PyDev will automatically move you into the definition. Also works on most variables and modules (try it on the import statements, for example).

Another good way to quickly jump between files without having to resort to the Package Explorer is to use `Ctrl + Shift + R`, which is the shortcut for “Open Resource”. Just type (part of) the file name you want to jump to and PyDev will search on every package and directory of your open projects.

Now that you have many files, note that you don’t need to necessarily have the file you want to run opened and active on the editor. For every script you run (using the procedure in which you need to be editing the program and then press `Ctrl + F11`) the IDE will remember that such script is something you are interested in running and will add it to the “Run History”. You can access the “Run History” on the main menu under *Run -> Run History*, or in the dropdown button located in the main toolbar, along the green “play” icon. In both places you will find the latest programs you ran, and many times using this list and selecting the script you want to re-run will be more convenient than jumping to the script on the editor and then pressing `Ctrl + F11`.

Finally, the IDE internally records an history of your “jumps” between files, just like a web browser do for web pages you visit. And just like a web browser you can go back and forward. To do this, use the appropriate button on the toolbar or the default shortcuts which are `Ctrl + Left` and `Ctrl + Right`.

Testing

OK, it’s about time to explore our options to test our code, without resorting to the cumbersome manual black box testing we have been done changing the command line argument and observing the output.

PyDev supports running PyUnit tests from the IDE, so we will write them. Let's create a module named `tests` on the `hello` package with the following code:

```
import unittest
import hello

class UIMock(object):
    def __init__(self):
        self.msgs = []
    def __call__(self, msg):
        self.msgs.append(msg)

class TestUIs(unittest.TestCase):
    def setUp(self):
        global hello
        hello = reload(hello)
        self.foo = UIMock()
        self.bar = UIMock()
        hello.register_ui('foo')(self.foo)
        hello.register_ui('bar')(self.bar)
        hello.message('foo', "message using the foo UI")
        hello.message('foo', "another message using foo")
        hello.message('bar', "message using the bar UI")

    def testBarMessages(self):
        self.assertEqual(["message using the bar UI"], self.bar.msgs)

    def testFooMessages(self):
        self.assertEqual(["message using the foo UI",
                          "another message using foo"],
                          self.foo.msgs)

    def testNonExistentUI(self):
        self.assertRaises(hello.UINotSupportedExeption,
                          hello.message, 'non-existent-ui', 'msg')

    def testListUIs(self):
        uis = hello.list_uis()
        self.assertEqual(2, len(uis))
        self.assert_('foo' in uis)
        self.assert_('bar' in uis)
```

As you can see, the test covers the functionality of the dispatching of messages to different UIs. A nice feature of PyDev is the automatic discovery of tests, so you don't need to code anything else to run the tests above. Just right click on the `src` folder on the Package Explorer and select *Run As* → *Jython unit-test*. You will see the output of the test almost immediately on the console:

```
Finding files...
['/home/lsoto/eclipse3.5/workspace-jythonbook/LearningPyDev/src/'] ... done
Importing test modules ... done.

testBarMessages (hello.tests.TestUIs) ... ok
testFooMessages (hello.tests.TestUIs) ... ok
testListUIs (hello.tests.TestUIs) ... ok
testNonExistentUI (hello.tests.TestUIs) ... ok

-----
Ran 4 tests in 0.064s
```

OK

Python's unittest is not the only testing option on the Python world. A convenient way to do tests which are more black-box-like than unit test, though equally automated is doctest.

Note: We will cover testing tools in much greater detail in Chapter 19, so take a look at that chapter if you feel too disoriented.

The nice thing about doctests is that they look like a interactive session with the interpreter, which makes them quite legible and easy to create. We will test our console module using a doctest.

First, click the rightmost button on the console's toolbar (you will recognize it as the one with a plus sign on its upper left corner, which has the "Open Console" tip when you pass the mouse over it). From the menu, select "PyDev Console". To the next dialog answer "Jython Console". After doing this you will get an interactive interpreter embedded on the IDE.

Let's start exploring our own code using the interpreter:

```
>>> from hello import console
>>> console.print_message("testing")
testing
```

I highly encourage you to type those two commands yourself. You will note how code completion also works on the interactive interpreter!

Back to the topic, we just interactively checked that our console module works as expected. The cool thing is that we can copy and paste this very snippet as a doctest which will serve to automatically check that the behavior we just tested will stay the same in the future.

Create a module named `doctests` inside the `hello` package, and paste those three lines from the interactive console, surrounding them by triple quotes (since they are not syntactically correct python code after all). After adding a little of boilerplate to make this file executable, it will look like this:

```
"""
>>> from hello import console
>>> console.print_message("testing")
testing
"""

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

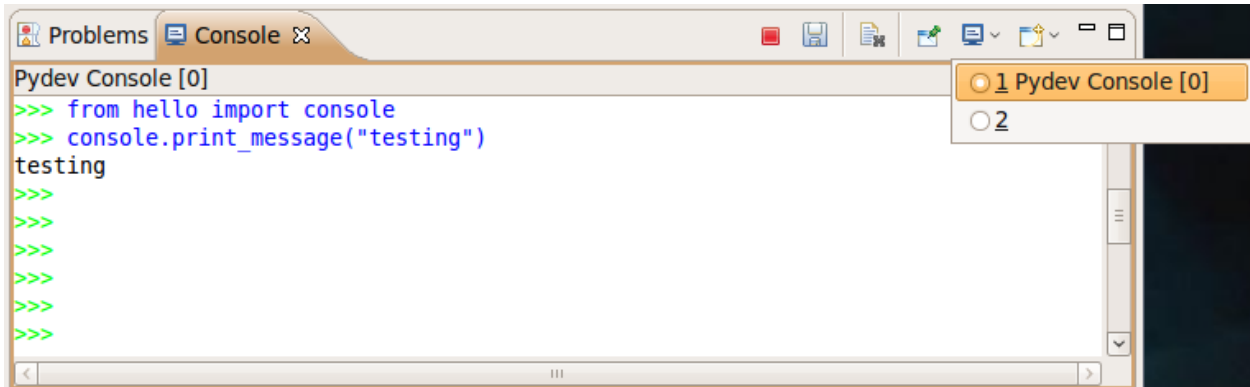
After doing this, you can run this test via the *Run* → *Jython run* menu while `doctests.py` is the currently active file on the editor. If all goes well, you will get the following output:

```
Trying:
      from hello import console
Expecting nothing
ok
Trying:
      console.print_message("testing")
Expecting:
      testing
ok
1 items passed all tests:
   2 tests in __main__
```



```
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

After running the doctest you will notice that your interactive console has gone away, replaced by the output console showing the test results. To go back to the interactive console, look for the console button in the console tab toolbar, exactly at the left of the button you used to spawn the console. Then, on the dropdown menu select the “PyDev Console” as shown in the next image.



As you can see, you can use the interactive console to play with your code, try ideas and test them. And later a simple test can be made just by copying and pasting text from the same interactive console session. Of special interest is the fact that, since Jython code can access Java APIs quite easily, you can also test classes written with Java in this way!

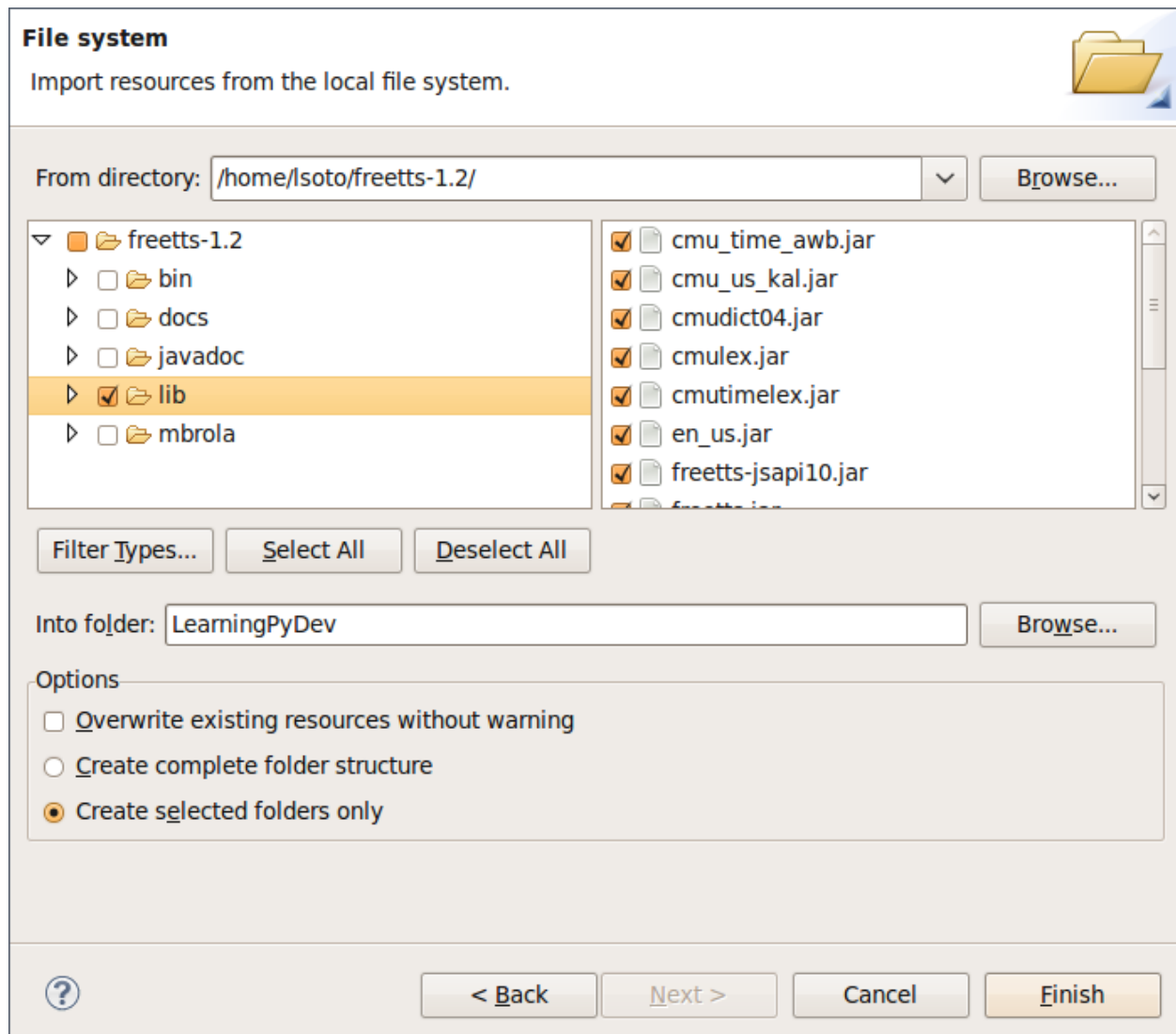
Adding Java libraries to the project

Finally, I will show you how to integrate Java libraries into your project. When testing the command line switches some pages ago, I hinted that we could have an “speech” interface for our greeter. It doesn’t sound like a bad idea after all, since (like on almost any aspect) the Java world has good libraries to solve that problem.

We will use the FreeTTS library, which can be downloaded from <http://freetts.sourceforge.net/docs/index.php>. (You should download the binary version)

After downloading FreeTTS you will have to extract the archive on some place on your hard disk. Then, we will import a JAR file from FreeTTS into our PyDev project.

Right click the project and select “Import...”. Then choose *General* → *File System* and browse to the directory in which you expanded FreeTTS and select it. Finally, expand the directory on the left side panel and check the `lib` subdirectory. See the following image as reference.



After clicking finish you will see that the file is now part of your project.

Tip: Alternatively, and depending on your operating system, the same operation can be performed copying the folder from the file manager and pasting it into the project (either via menu, keyboard shortcuts or drag & drop).

Now, the file is part of the project, but we need to tell PyDev that the file is a JAR file and should be added to the `sys.path` of our project environment. To do this right click on the project and select “Properties”. Then on the left panel of the dialog select “PyDev - PYTHONPATH”. Then click the “Add zip/jar/egg” button and select the `lib/freetts.jar` file on the right side of the dialog that will pop out. Click OK on both dialogs and you are ready to use this library from Python code.

The code for our new `hello.speech` module is as follows:

```
from com.sun.speech.freetts import VoiceManager
from hello import register_ui

@register_ui('speech')
def speech_message(msg):
    voice = VoiceManager().getVoice("kevin16")
```

```
voice.allocate()  
voice.speak(msg)  
voice.deallocate()
```

If you play with the code on the editor you will notice that PyDev also provides completion for imports statement referencing the Java library we are using.

Finally, we will change the second line of `main.py` from:

```
import hello, hello.console, hello.window
```

to:

```
import hello, hello.console, hello.window, hello.speech
```

In order to load the speech UI too. Feel free to power on the speakers and use the `--ui speech` option to let the computer greet yourself and your friends!

There you go, our humble greeter has finally evolved into a quite interesting, portable program with speech synthesis abilities. It's still a toy, but one which shows how quick you can move with the power of Jython, the diversity of Java and the help of an IDE.

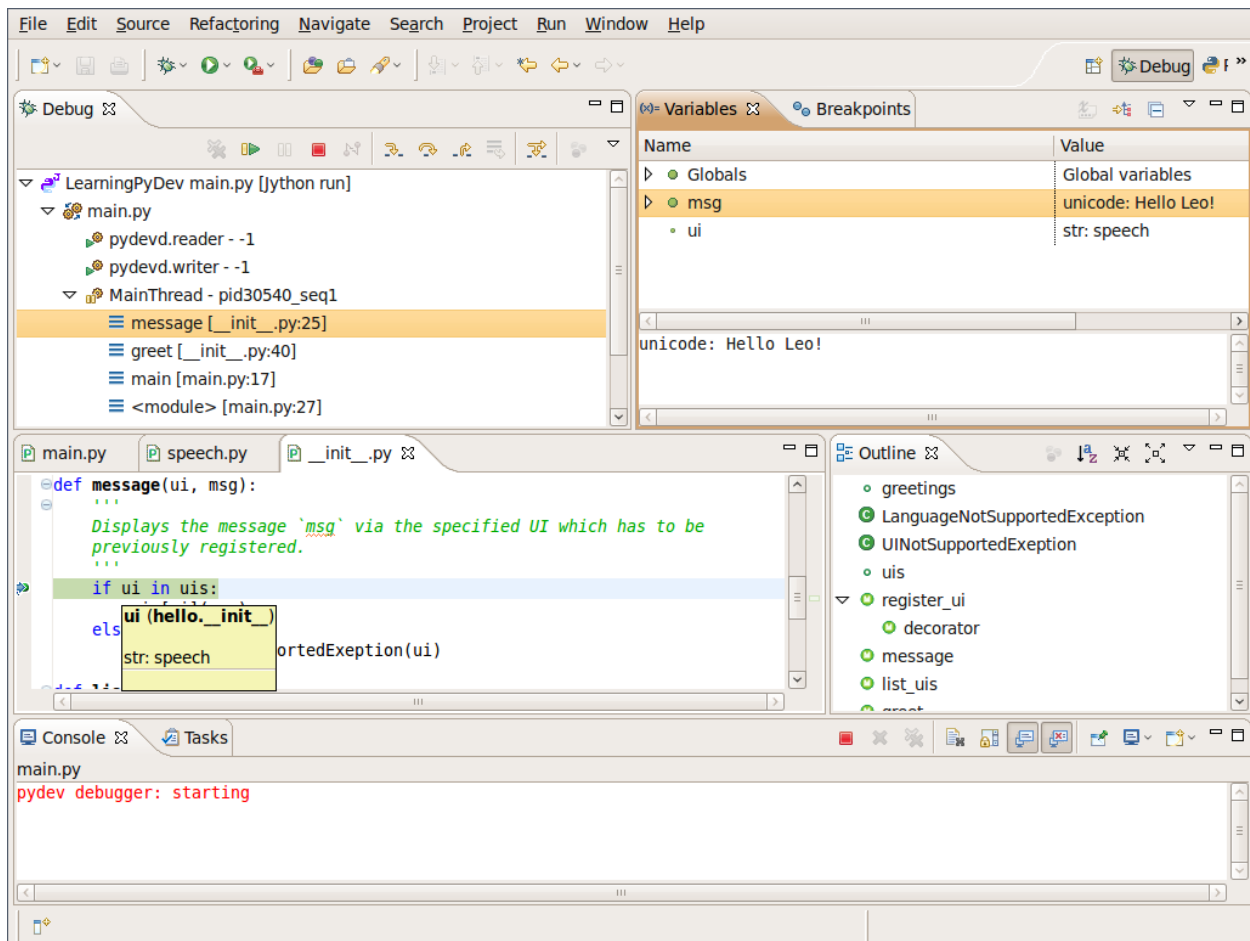
Other topics

I have covered most of the PyDev features, but I've left a few unexplored. We will take a look at what we've missed before ending this half-chapter dedicated to PyDev.

Debugging

PyDev offers full debugging abilities for your Jython code. To try it just put some breakpoints on your code double clicking on the left margin of the editor, and then start your program using the `F11` shortcut instead of `Ctrl + F11`.

Once the debugger hits your breakpoint, the IDE will ask you to change its perspective. It means that it will change to a different layout, better suited for debugging activities. Answer yes to such dialog and you will find yourself on the debugging perspective which looks like the following image:



In few words, the perspective offers the typical elements of a debugger: the call stack, the variables for each frame of the call stack, a list of breakpoints, and the ability to “Step Into” (F5), “Step Over” (F6) and “Resume Execution” (F8) among others.

Once you finish your debugging session, you can go back to the normal editing perspective by selecting “PyDev” on the upper right area of the main IDE Window (which will have the “Debug” button pushed while staying in the debugging perspective).

Refactoring

PyDev also offers some basic refactoring abilities. Some of them are limited to CPython, but others, like “Extract Method” work just fine with Jython projects. I encourage you to try them to see if they fit your way of work. Sometimes you may prefer to refactor manually since the task tend do not be as painful as in Java (or any other statically typed language without type inference). On the other hand, when the IDE can do the right thing for you and avoid some mechanical work, you will be more productive.

(Half-)Conclusion

PyDev is a very mature plugin for the Eclipse platform which can be an important element in your toolbox. Automatic completion and suggestions helps a lot when learning new APIs (both Python APIs and Java APIs!) specially if paired with the interactive console. It is also a good way to introduce a whole team into Jython or into an specific Jython project, since the project-level configuration can be shared via normal source control system. Not to mention that programmers coming from the Java world will find themselves much more comfortable on a familiar environment.

To me, IDEs are a useful part of my toolbox, and tend to shine on big codebases and/or complex code which I don't completely understand yet. Powerful navigation and refactoring abilities are key on the process of understanding such kind of projects and are features that should only improve in the future.

Finally, the debugging capabilities of PyDev are superb and will end your days of using `print` as a poor man's debugger (Seriously, I did that for a while!). Even more advanced Python users who master the art of `import pdb; pdb.set_trace()` should give it a try.

Now, this is a “half-conclusion” because PyDev isn't the only IDE available for Jython. If you are already using the Netbeans IDE or didn't like Eclipse or PyDev for some reason, take a look at the rest of this chapter in which we will cover the Netbeans plugin for Python development.

Netbeans

The Netbeans integrated development environment has been serving the Java community well for over ten years now. During that time, the tool has matured quite a bit from what began as an ordinary Java development tool into what is today an advanced development and testing environment for Java and other languages alike. As Java and JavaEE application development still remain an integral part of the tool, other languages such as JRuby, Jython, Groovy, and Scala have earned themselves a niche in the tool as well. Most of these languages are supported as plugins to the core development environment, which is what makes Netbeans such an easy IDE to extend as it is very easy to build additional features to distribute. The Python support within Netbeans began as a small plugin known as nbPython, but it has grown into a fully-featured Python development environment and it continues to grow.

The Netbeans Python support provides developers with all of the expected IDE features such as code completion, color coding, and easy runtime development. It also includes some nice advanced features for debugging applications and the like.

IDE Installation and Configuration

The first step for installing the Netbeans Python development environment is to download the current release of the Netbeans IDE. At the time of this writing, Netbeans 6.7 was the most recent release, hot off the presses in fact. You can find the IDE download by going to the website <http://www.netbeans.org> and clicking on the download link. Once you do so, you'll be presented with plenty of different download options. These are variations of the IDE that are focused on providing different features for developers depending upon what they will use the most. Nobody wants a bulky, memory hungry development tool that will overhaul a computer to the extreme. By providing several different configuration of the IDE, Netbeans gives you the option to leave off the extras and only install those pieces that are essential to your development. The different flavors for the IDE include Java SE, Java, Ruby, C/C++, PHP, and All. For those developers only interested in developing core Java applications, the Java SE download would suffice. Likewise, someone interested in any of the other languages could download the IDE configuration specific to that language. For the purposes of this book and in my everyday development, I use the “All” option as I enjoy having all of the options available. However, there are options available for adding features if you download only the Java SE or another low-profile build and wish to add more later.

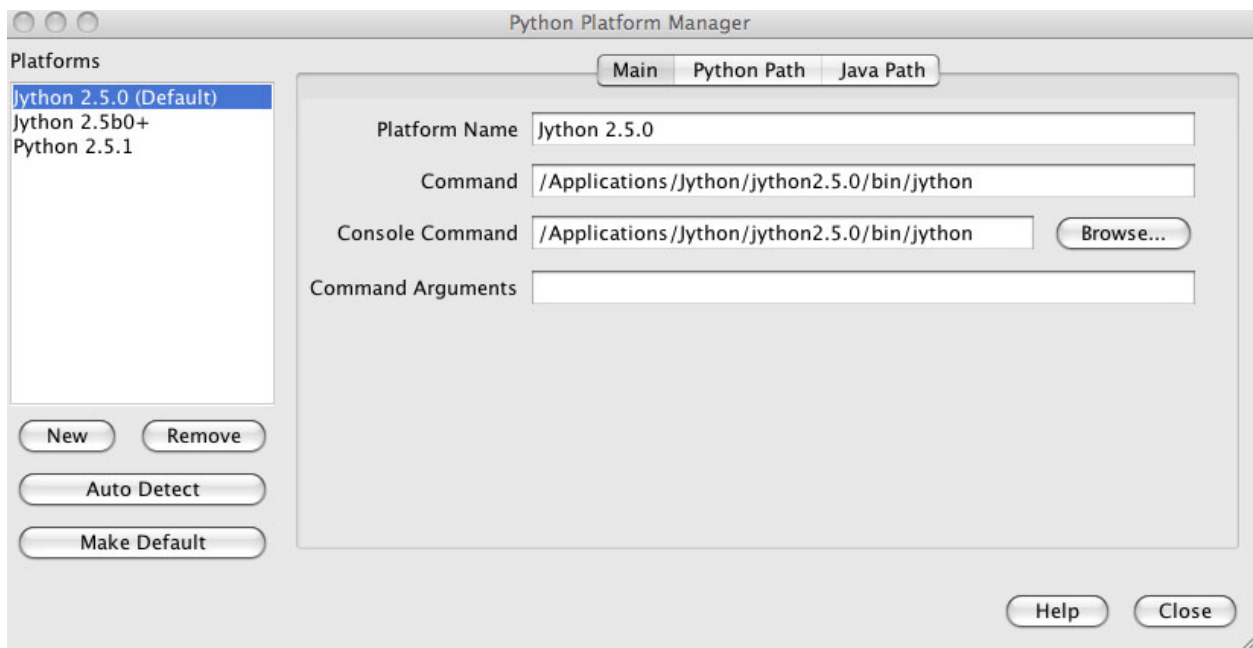
At the time of this writing, there was also a link near the top of the downloads page for PythonEA distribution. If that link or a similar Python Netbeans distribution link is available then you can use it to download and install just the Jython-specific features of the Netbeans IDE. I definitely do not recommend taking this approach unless you plan to purely code Jython applications alone. It seems to me that a large population of the Jython developer community also codes some Java, and may even integrate Java and Jython within their applications. If this is the case, you will want to have the Java-specific features of Netbeans available as well. That is why I do not recommend the Python-only distribution for Jython developers, but the choice is there for you to make.

Now that you've obtained the IDE, it is easy to install in any environment using the intuitive Netbeans installer. Perhaps the most daunting task when using a new IDE is configuring it for your needs. This should not be the case with Netbeans though because the configuration for Java and Python alike are quite simple. For instance, if you working with the fully-featured installation, you will already have application servers available for use as Netbeans

installs Glassfish by default. Note that it is a smart idea to change that admin password very soon after installation in order to avoid any potentially embarrassing security issues.

When the IDE initially opens up, you are presented with a main window that includes links to blogs and articles pertaining to Netbeans features. You also have the standard menu items available such as File, Edit, Tools, and so on. In this chapter we will specifically cover the configuration and use of the Jython features, however, there are very useful tutorials available online and in book format for covering other Netbeans features. One thing you should note at this point is that with the initial installation, Python/Jython development tools are not yet installed unless you chose to install the *PythonEA* distribution. Assuming that you have installed the full Netbeans distribution, you will need to add the Python plugin via the Netbeans plugin center. You will need to go to the *Tools* menu and then open the *Plugins* *submenu. From there, you should choose the **Available Plugins* tab and sort by category. Select all of the plugins in the *Python* category and then install. This option will install the Python plugin as well as a distribution of Jython. You will need to follow on-screen directions to complete the installation.

Once the plugin has been successfully installed then it is time to configure your Python and Jython homes. To do so, go to the *Tools* menu and then open the *Python Platforms* menu as this will open the platform manager for Python/Jython. At the time of this writing, the default Jython version that was installed with the Python plugin was 2.5b0+, even though 2.5.0 final has been release. As this is the case, go ahead and add your Jython 2.5.0 final installation as a platform option and make it the default.



To do so, click on the *New* button underneath the platform listing. You can try to select the *Auto Detect* option, but I did not have luck with Netbeans finding my Jython installation for 2.5.0 final using it. If you choose the *New* button then you will be presented with a file chooser window. You should choose the Jython executable that resides in the area <JYTHON_HOME>/bin and all of the other necessary fields will auto-populate with the correct values. Once completed, choose the *Close* button near the bottom of the *Python Platform Manager* window. You are now ready to start programming with Python and Jython in Netbeans.

Advanced Python Options

If you enter the Netbeans preferences window then you will find some more advanced options for customizing your Python plugin. If you go to the *Editor* tab, you can set up Python specific options for formatting, code templates, and hints. In doing so, you can completely customize the way that Netbeans displays code and offers assistance when working with Jython. You can also choose to set up different fonts and coloring for Python code by selecting the *Fonts*

and *Colors* tab. This is one example of just how customizable Netbeans really is because you can set up different fonts and colors for each language type.

If you choose the *Miscellaneous* tab then you can add different file types to the Netbeans IDE and associate them with different IDE features. If you look through the pull-down menu of files, you can see that files with the extension of *py* or *pyc* are associated as Python files. This ensures that files with the associated extensions will make use of their designated Netbeans features. For instance, if we wanted to designate our Jython files with the extension of *jy*, we could easily do so and associate this extension with Python files in Netbeans. Once we've made this association then we can create files with an extension of *jy* and use them within Netbeans just as if they were Python files. Lastly, you can alter a few basic options such as enabling prompting for python program arguments, and changing debugger port and shell colors from the *Python* tab in Netbeans preferences.

General Jython Usage

As stated previously in the chapter, there are a number of options when using the Netbeans Python solution. There are a few different selections that can be made when creating a new Jython project. You can either choose to create a *Python Project* or *Python Project with Existing Sources*. These two project types are named quite appropriately as a *Python Project* will create an empty project, and

Once created it is easy to develop and maintain applications and scripts alike. Moreover, you can debug your application and have Netbeans create tests if you choose to do so. One of the first nice features you will notice is the syntax coloring in the editor. There is also testing available via the debugging

Stand Alone Jython Apps

In this section, I will discuss how to develop a stand-alone Jython application within Netbeans. We will use a variation of the standard *HockeyRoster* application that I have used in other places throughout the book. Overall, the development of a stand alone Jython application in Netbeans differs very little from a stand alone Java application. The main difference is that you will have different project properties and other options available that pertain to creating Jython. And obviously you will be developing in Jython source files along with all of the color coding and code completion that the Python plugin has to offer.

To get started, go ahead and create a new Python Project by using the *File* menu or the shortcut in the Netbeans toolbar. For the purposes of this section, name the new project *HockeyRoster*. Uncheck the option to *Create Main File* as we will do this manually. Once your project has been created, explore some of the options you have available by right-clicking (ctrl-click) on the project name. The resulting menu should allow you the option to create new files, run, debug, or test your application, build eggs, work with code coverage, and more. At this point you can also change the view of your Python packages within Netbeans by choosing the *View Python Packages as* option. This will allow you the option to either see the application in *list* or *tree* mode, your preference. You can search through your code using the *Find* option, share it on Kenai with the integrated Netbeans Kenai support, look at the local file history, or use your code with a version control system. Click on the *Properties* option and the *Project Properties* window should appear. From within the *Project Properties* window, there are options listed on the left-hand side including *Source*, *Python*, *Run*, and *Formatting*. The *Source* option provides the ability to change source location or add new source locations to your project. The *Test Root Folders* section within this option allows you to add a location where Python tests reside so that you can use them with your project. The *Python* option allows you to change your Python platform and add locations, JARs, and files to your Python path. Changing your Python platform provides a handy ability to test your program on Jython and Python alike, if you want to ensure that your code works on each platform. The *Run* option provides the ability to add or change the *Main* module, and add application arguments. Lastly, the *Formatting* option allows you to specify different formatting options in Netbeans for this particular project. This is great because each different project can have different colored text, etc. depending upon the options chosen.

At this point, create the *Main* module for the *HockeyRoster* application by using the *File* and then *New* drop-down menu, right-clicking (cntrl-click) on the project, or using the toolbar icon. From here you can either create an Executable Module, Module, Empty Module, Python Package, or Unit Test. Choose to create an Executable Module

and name the main file *HockeyRoster.py*, and keep in mind that when we created the project we had the ability to have the IDE generate this file for us but we chose to decline. Personally, I like to organize my projects using the Python packaging system. Create a some packages now using the same process that you used to create a file and name the package *org*. Add another package within the first and name it *jythonbook*. Once created, drag your *HockeyRoster.py* module into the *jythonbook* package to move it into place. Note that you can also create several packages at the same time by naming a package like *org.jythonbook*, which will create both of the resulting packages.

The *HockeyRoster.py* main module will be the implementation module for our application, but we still need somewhere to store each of the player's information. For this, we will create class object container named *Player.py*. Go ahead and create an "Empty Module" named *Player* within the same *jythonbook* package. Now we will code the *Player* class for our project. To do so, erase the code that was auto-generated by Netbeans in the *Player.py* module and type the following. Note that you can change the default code that is created when generating a new file by changing the template for Python applications.

```
# Player.py
#
# Class container to hold player information

class Player:

    # Player attributes

    id = 0
    first = None
    last = None
    position = None
    goals = 0
    assists = 0

    def create(self, id, first, last, position):
        self.id = id
        self.first = first
        self.last = last
        self.position = position

    def set_goals(self, goals):
        self.goals = goals

    def add_goal(self):
        self.goals = goals + 1

    def get_goals(self):
        return self.goals

    def set_assists(self, assists):
        self.assists = assists

    def add_assist(self):
        self.assists = assists + 1

    def get_assists(self):
        return self.assists
```

The first thing to note is that Netbeans will maintain your indentation level. It is also easy to tab backwards by using the SHIFT + TAB keyboard shortcut. Using the default environment settings, the keywords should be in a different color (blue by default) than the other code. Method names will be in bold, and references to *self* or variables will

be in a different color as well. You should notice some code completion, mainly the automatic *self* placement after you type a method name and then the right parentheses. Other subtle code completion features also help to make our development lives easier. If you make an error, indentation or otherwise, you will see a red underline near the error as well as a red error badge on the line number within the left-hand side of the editor. Netbeans will offer you some assistance in determining the cause of the error if you hover your mouse over the red error badge or underline.

Now that we have coded the first class in our stand-alone Jython application, it is time to take a look at the implementation code. The *HockeyRoster.py* module is the heart of our roster application as it controls what is done with the team. We will use the *shelve* technique to store our *Player* objects to disk for the roster application. As you can see from the code below, this is a very basic application and is much the same as the implementation that will be found in the next chapter using Hibernate persistence.

```
# HockeyRoster.py
#
# Implementation logic for the HockeyRoster application

# Import Player class from the Player module

from Player import Player
import shelve
import sys

# Define a list to hold each of the Player objects
playerList = []
factory = None

# Define shelve for storage to disk
playerData = None

# makeSelection()
#
# Creates a selector for our application. The function prints output to the
# command line. It then takes a parameter as keyboard input at the command line
# in order to choose our application option.

def makeSelection():
    validOptions = ['1', '2', '3', '4', '5']
    print "Please chose an option\n"

    selection = raw_input("Press 1 to add a player, 2 to print the roster, 3 to ↵
↳search for a player on the team, 4 to remove player, 5 to quit: ")
    if selection not in validOptions:
        print "Not a valid option, please try again\n"
        makeSelection()
    else:
        if selection == '1':
            addPlayer()
        elif selection == '2':
            printRoster()
        elif selection == '3':
            searchRoster()
        elif selection == '4':
            removePlayer()
        else:
            print "Thanks for using the HockeyRoster application."

# addPlayer()
#
```

```

# Accepts keyboard input to add a player object to the roster list. This function
# creates a new player object each time it is invoked and appends it to the list.
def addPlayer():
    addNew = 'Y'
    print "Add a player to the roster by providing the following information\n"
    while addNew.upper() == 'Y':
        first = raw_input("First Name: ")
        last = raw_input("Last Name: ")
        position = raw_input("Position: ")

        id = returnPlayerCount() + 1
        print id
        #set player and shelve
        player = Player(id, first, last, position)
        playerData[str(id)] = player

        print "Player successfully added to the roster\n"
        addNew = raw_input("Add another? (Y or N)")
    makeSelection()

# printRoster()
#
# Prints the contents of the list to the command line as a report
def printRoster():
    print "=====\n"
    print "Complete Team Roster\n"
    print "=====\n\n"
    playerList = returnPlayerList()
    for player in playerList.keys():
        print "%s %s - %s" % (playerList[player].first, playerList[player].last,
        ↪playerList[player].position)
    print "\n"
    print "=== End of Roster ===\n"
    makeSelection()

# searchRoster()
#
# Takes input from the command line for a player's name to search within the
# roster list. If the player is found in the list then an affirmative message
# is printed. If not found, then a negative message is printed.
def searchRoster():
    index = 0
    found = False
    print "Enter a player name below to search the team\n"
    first = raw_input("First Name: ")
    last = raw_input("Last Name: ")
    position = None
    playerList = returnPlayerList()
    for playerKey in playerList.keys():
        player = playerList[playerKey]
        if player.first.upper() == first.upper() and player.last.upper() == last.
        ↪upper():
            found = True
            position = player.position
    if found:
        print '%s %s is in the roster as %s' % (first, last, position)
    else:

```

```

        print '%s %s is not in the roster.' % (first, last)
    makeSelection()

def removePlayer():
    index = 0
    found = False
    print "Enter a player name below to remove them from the team roster\n"
    first = raw_input("First Name: ")
    last = raw_input("Last Name: ")
    position = None
    playerList = returnPlayerList()
    for playerKey in playerList.keys():
        player = playerList[playerKey]
        if player.first.upper() == first.upper() and player.last.upper() == last.
↪upper():
            found = True
            foundPlayer = player
        if found:
            print '%s %s is in the roster as %s, are you sure you wish to remove?' %_
↪(foundPlayer.first, foundPlayer.last, foundPlayer.position)
            yesno = raw_input("Y or N")
            if yesno.upper() == 'Y':
                # remove player from shelfe
                print 'The player has been removed from the roster', foundPlayer.id
                del (playerData[str(foundPlayer.id)])
            else:
                print 'The player will not be removed'
        else:
            print '%s %s is not in the roster.' % (first, last)
    makeSelection()

def returnPlayerList():
    playerList = playerData
    return playerList

def returnPlayerCount():
    return len(playerData.keys())

# main
#
# This is the application entry point. It simply prints the applicaion title
# to the command line and then invokes the makeSelection() function.
if __name__ == "__main__":
    print sys.path
    print "Hockey Roster Application\n\n"
    playerData = shelve.open("players")
    makeSelection()

```

The code should be relatively easy to follow at this point in the book. The *main* function initiates the process as expected, and as you see it either creates or obtains a reference to the shelfe or dictionary where the roster is stored. Once this occurs then the processing is forwarded to the *makeSelection()* function that drives the program. The important thing to note here is that when using Netbeans the code is layed out nicely, and that code completion will assist with imports and completion of various code blocks. To run your program, you can either right-click (CTRL+CLICK) on the project or set the project as the main project within Netbeans and use the toolbar or pull-down menus. If everything has been set up correctly then you should see the program output displaying in the Netbeans *output* window. You can interact with the output window just as you would with the terminal.

Jython and Java Integrated Apps

Rather than repeat the different ways in which Jython and Java can be intermixed within an application, this section will focus on how to do so from within the Netbeans IDE. There are various approaches that can be taken in order to perform integration, so this section will not cover all of them. However, the goal is to provide you with some guidelines and examples to use when developing integrated Jython and Java applications within Netbeans.

Using a JAR or Java Project in Your Jython App

Making use of Java from within a Jython application is all about importing and ensuring that you have the necessary Java class files and/or JAR files in your classpath. In order to achieve this technique successfully, you can easily ensure that all of the necessary files will be recognized by the Netbeans project. Therefore, the focus of this section is on using the Python project properties to set up the `sys.path` for your project. To follow along, go ahead and use your *HockeyRoster* Jython project that was created earlier in this section.

Let's say that we wish to add some features to the project that are implemented in a Java project named *HockeyIntegration* that we are coding in Netbeans. Furthermore, let's assume that the *HockeyIntegration* Java project compiles into a JAR file. In order to use this project from within our *HockeyRoster* project, you'll need to open up the project properties by right-clicking on your Jython project and choosing the *Properties* option. Once the window is open then click on the *Python* menu item on the left-hand side of the window. This will give you access to the `sys.path` so you can add other Python modules, eggs, Java classes, JAR files, etc. Click on the *Add* button and then traverse to the project directory for the Java application you are developing. Once there, go within the *dist* directory and select the resulting JAR file and click *OK*. You can now use any of the Java project's features from within your Jython application.

If you are interested in utilizing a Java API that exists within the standard Java library then you are in great shape. As you should know by now, Jython automatically provides access to the entire Java standard library. You merely import the Java that you wish to use within your Jython application and begin using, nothing special to set up within Netbeans. At the time of this writing, the Netbeans Python EA did not support import completion for the standard Java library. However, I suspect that this feature will be added in a subsequent release.

Using Jython in Java

If you are interested in using Jython or Python modules from within your Java applications, Netbeans makes it easy to do. As mentioned in Chapter 10, the most common method of utilizing Jython from Java is to use the object factory pattern. However, there are other ways to do this such as using the *clamp* project which is not yet production ready at the time of this writing. For the purposes of this section, we'll discuss how to utilize another Netbeans Jython project as well as other Jython modules from within your Java application using the object factory pattern.

In order to effectively demonstrate the use of the object factory pattern from within Netbeans, we'll be making use of the *PlyJy* project which provides object factory implementations that can be used out-of-the-box. If you haven't done so already, go to the *Project Kenai* site find the *PlyJy* project and download the provided JAR. We will use the Netbeans project properties window in our Java project to add this JAR file to our project. Doing so will effectively diminish the requirement of coding any object factory implementations by hand and we'll be able to directly utilize Jython classes in our project.

Create a Java project named *ObjectFactoryExample* by using the "New->Project->Java Application" selection. Once you've done so, right-click (CNTRL+CLICK) on the project and choose *Properties*. Once the project properties window appears, click on the *Libraries* option on the left-hand side. From there, add the *PlyJy* JAR file that you previously downloaded to your project classpath. You will also have to add the *jython.jar* file for the appropriate version of Jython that you wish to use. In our case, we will utilize the Jython 2.5.0 release.

The next step is to ensure that any and all Jython modules that you wish to use are in your CLASSPATH somewhere. This can be easily done by either adding them into your application as regular code modules somewhere and then going into the project properties window and including that directory in "Compile-Time Libraries" list contained the *Libraries* section by using the "Add JAR/Folder" button. Although this step may seem unnecessary because the

modules are already part of your project, it must be done in order to place them into your CLASSPATH. Once they've been added to the CLASSPATH successfully then you can begin to make use of them via the object factory pattern. Netbeans will seamlessly use the modules in your application as if all of the code was written in the same language.

Developing Web Apps (Django, etc)

As of the time of this writing, Netbeans has very little support for developing Jython web applications as far as frameworks go. Developing simple servlets and/or applets with Jython are easy enough with just creating a regular web application and setting it up accordingly. However, making full use of a framework such as Django from within Netbeans is not available as of version 6.7. There are many rumors and discussions in the realm of a Django plugin to become part of the Netbeans 7 release, but perhaps that will be covered in a future edition of this book. In the meantime we need to make use of Netbeans in it's current form, without a plugin specifically targeted for Jython web development. Although there are a few hurdles and none of the frameworks can be made completely functional from within the tool, there are some nice tricks that can be played in order to allow Jython web development worth executing within Netbeans.

In order to deploy a standard web application in Netbeans and make use of Jython servlets and/or applets, simply create a standard web application and then code the Jython in the standard servlet or applet manner. Since there are no plugins to support this work it is all a manual process. Something tells me that making use of the fine code completion and semantec code coloring is a nice perk even if there aren't any wizards to assist you in coding your *web.xml* configuration. Since there are not any wizards to help us out, we will only mention that Netbeans makes standard web Jython web development easier by utilizing the features of the IDE, not abstracting away the coding and instead completing wizards.

Using Django in Netbeans

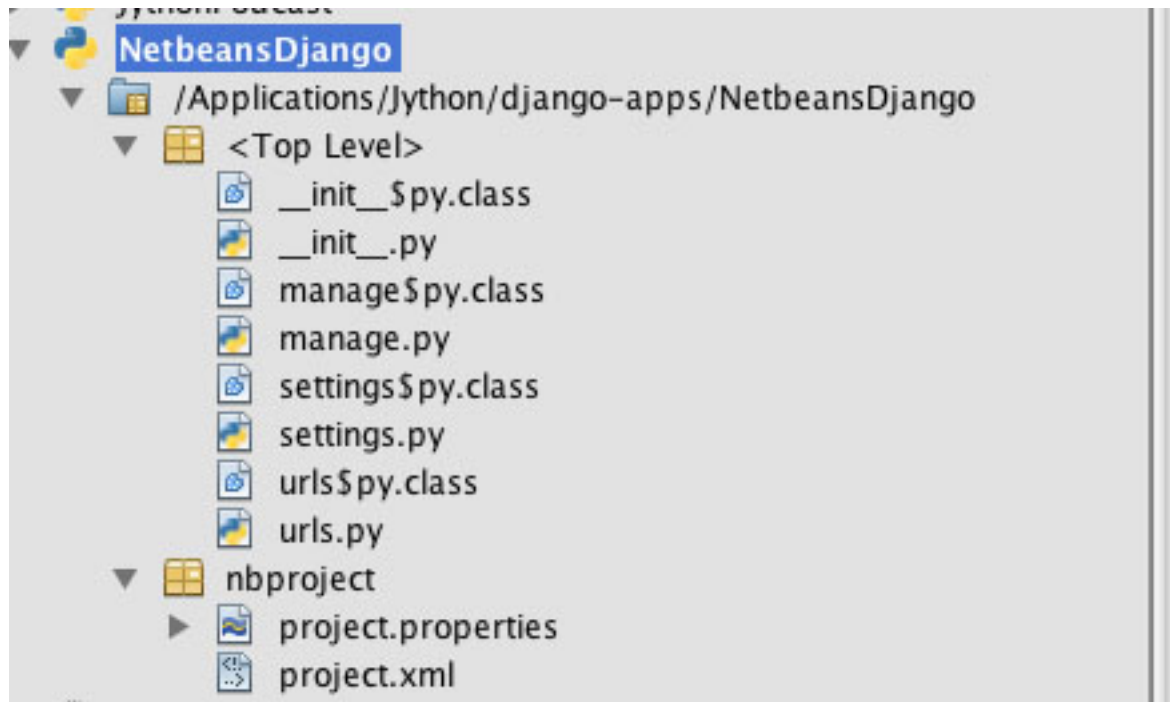
As stated at the beginning of this section, it is not a very straight forward task if you wish to develop Jython web applications utilizing a standard framework from within Netbeans. However, with a little extra configuration and some manual procedures it is easy enough to do. In this section I will demonstrate how we can make use of Netbeans for developing a Django application without using any Netbeans plugins above and beyond the standard Python support. You will see that Jython applicatons can be run, tested, and verified from within the IDE with very little work. Since there are a few steps in this section that may be more difficult to visualize, please use the provided screen shots to follow along if you are not using Netbeans while reading this text.

In order to effectively create and maintain a Django website, you need to have the ability to run commands against *manage.py*. Unfortunately, there is no built in way to easily do this within the IDE so we have to use the terminal or command line along with the IDE to accomplish things. Once we create the project and set it up within Netbeans then we can work with developing it from within Netbeans and you can also set up the project *Run* feature to startup the Django server.

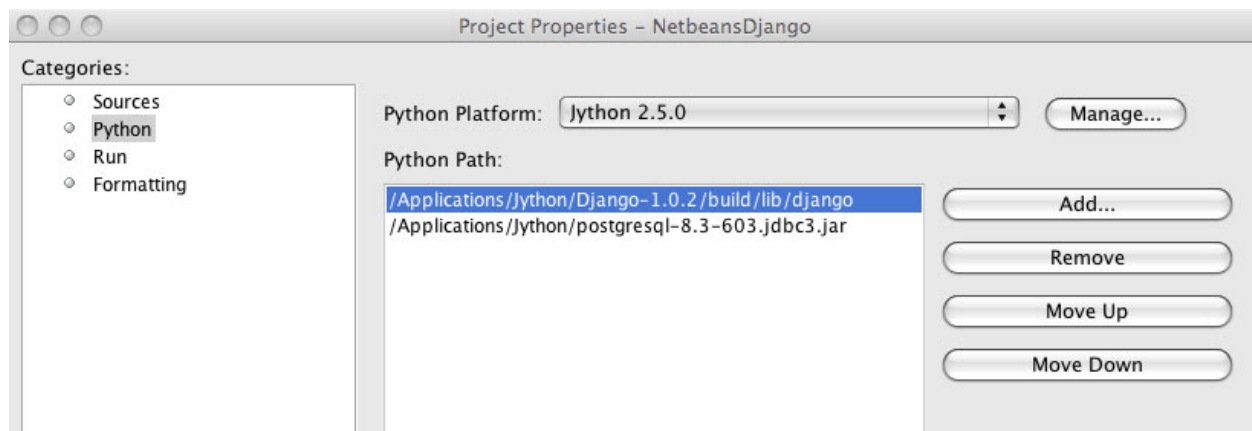
Assuming that you already have Django setup and configured along with the Django-Jython project on your machine, the first step in using a Django project from within Netbeans is actually creating the project. If you are working with a Django project that has already been created then you can skip this step, but if not then you will need to go to the terminal or command-line and create the project using *django-admin.py*. For the purposes of this tutorial, let's call our Django site *NetbeansDjango*.

```
django-admin.py startproject NetbeansDjango
```

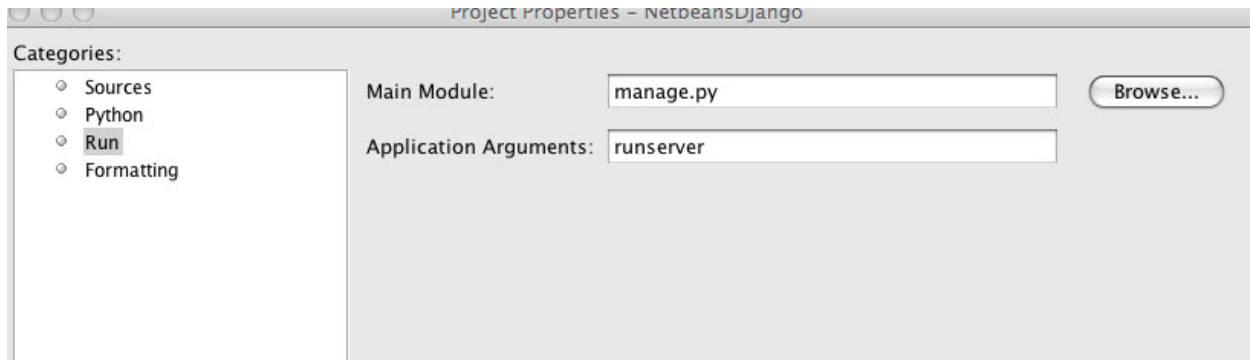
Now we should have the default Django site setup and we're ready to bring it into Netbeans. To do this, start a new Python project within Netbeans using the *Python Project with Existing Sources* option, and be sure to set your Python Platform to Jython 2.5.0 so we are using Jython. After hitting the *Next* button we have the ability to add sources to our project. Hit the *Add* button and choose the select the main project folder, so in our case select the *NetbeansDjango* folder. This will add our project root as the source root for our application. In turn, it adds our Django setup files such as *manage.py* to our project. After doing so your project should look something like the following screenshot.



In this next step, we will configure the Netbeans project *Run* option so that it starts up the Django web server for us. To do so, right-click (CNTRL+CLICK) on the newly created project and go to *Properties*. From there choose the *Python* option in the left-hand menu and add the Django directory (containing the bin, conf, contrib, core, etc. files) to your path. For this tutorial we will also make use of the PostgreSQL database, so you'll want to also add the *postgresql.jar* to your Python path.



Next, select the *Run* option from the left-hand menu and add *manage.py* as the main module, and add *runserver* as the application argument. This will essentially hook-up the *Run* project option to the Django *manage.py* such that it will invoke the Django webserver to start up.



At this point, we are ready to begin developing our Django application. So with a little minor setup and some use of the terminal or command-line we are able to easily use Netbeans for developing Django projects. There are a few minor inconsistencies with this process however, note that there is no real integrated way to turn off the webserver as yet so once it is started we can either leave it running or stop it via your system process manager. Otherwise you can hook up different options to the Netbeans *Run* project command such as *syncdb* by simply choosing a different application argument in the project properties. If you use this methodology, then you can simply start and stop the Django web server via the terminal as normal. I have also found that after running the Django web server you will have to manually delete the *settings\$.py.class* file that is generated before you can run the server again or else it will complain.

In future versions of Netbeans, namely the Netbeans 7 release, it is expected that Django functionality will be built into the Python support. We will have to take another look at using Django from within Netbeans at that time. For now, this procedure works and it does a fine job. You can make use of similar procedures to use other web frameworks such as Pylons from within Netbeans.

Conclusion

As with most other programming languages, you have several options to use for an IDE when developing Jython. In this chapter we covered two of the most widely used IDE options for developing Jython applications, Netbeans and Eclipse. Eclipse offers a truly complete IDE solution for developing Jython applications, both stand alone and web based. Along with the inclusion of the Django plugin for Eclipse, the IDE makes it very easy to get started with Jython development and also manage existing projects. PyDev is under constant development and always getting better, adding new features and streamlining existing features.

Netbeans Jython support is still in early development at the time of this writing. Many of the main features such as code completion and syntax coloring are already in place. It is possible to develop Jython applications including Jython and Java integration as well as web based applications. In the future, Netbeans Jython support will develop to include many more features and they will surely be covered in future releases of this book.

In the next chapter, we will take a look at developing some applications utilizing databases. The zxJDBC API will be covered and you'll learn how to develop Jython applications utilizing standard database transactions. Object relational mapping is also available for Jython in various forms, we'll discuss many of those options as well.

Chapter 12- Databases and Jython: Object Relational Mapping and Using JDBC

First, we will look at zxJDBC package, which is a standard part of Jython since version 2.1 and complies with the Python 2.0 DBI standard. zxJDBC can be an appropriate choice for simple one-off scripts where database portability is not a concern. In addition, it's (generally) necessary to use zxJDBC when writing a new dialect for SQLAlchemy or Django. [But not strictly true: you can use pg8000, a pure Python DBI driver, and of course write your own DBI drivers. But please don't do that.] So knowing how zxJDBC works can be useful when working with these packages.

However, it's too low level for us to recommend for more general usage. Use SQLAlchemy or Django if at all possible. Finally, JDBC itself is also directly accessible, like any other Java package from Jython. Simply use the `java.sql` package. In practice this should be rarely necessary.

The second portion of this chapter will focus on using object relational mapping with Jython. The release of Jython 2.5 has presented many new options for object relational mapping. In this chapter we'll focus on using SQLAlchemy with Jython, as well as using Java technologies such as Hibernate. In the end you should have a couple of different choices for using object relational mapping in your Jython applications.

zxJDBC – Using Python's DB API via JDBC

Introduction to zxJDBC

The zxJDBC package provides an easy-to-use Python wrapper around JDBC. zxJDBC bridges two standards:

- JDBC is the standard platform for database access in Java.
- DBI is the standard database API for Python apps.

zxJDBC, part of Jython, provides a DBI 2.0 standard compliant interface to JDBC. Over 200 drivers are available for JDBC [<http://developers.sun.com/product/jdbc/drivers>], and they all work with zxJDBC. High performance drivers are available for all major relational databases, including DB2, Derby, MySQL, Oracle, PostgreSQL, SQLite, SQLServer, and Sybase. And drivers are also available for non-relational and specialized databases too.

However, unlike JDBC, zxJDBC when used in the simplest way possible, blocks SQL injection attacks, minimizes overhead, and avoids resource exhaustion. In addition, zxJDBC defaults to using a transactional model (when available), instead of autocommit.

First we will look at connections and cursors, which are the key resources in working with zxJDBC, just like any other DBI package. Then we will look at what you can do them with them, in terms of typical queries and data manipulating transactions.

Getting Started

The first step in developing an application that utilizes a database back-end is to determine what database or databases the application will use. In the case of using zxJDBC or another JDBC implementation, the determination of what database the application will make use of is critical to the overall development process. Many application developers will choose to use an object relational mapper for this very reason. When an application is coded with a JDBC implementation whereas SQL code is hand-coded, the specified database of choice will cause different dialects of SQL to be used. One of the benefits of object relation mapping (ORM) technology is that the SQL is transparent to the developer. The ORM technology takes care of the different dialects behind the scenes. This is one of the reasons why ORM technology may be slower at implementing support for many different databases. Take SQLAlchemy or Django for instance, each of these technologies must have a different dialect coded for each database. Using an ORM can make an application more portable over many different databases. However, as stated in the preface using zxJDBC would be a fine choice if your application is only going to target one or two databases.

While using JDBC for Java, one has to deal with the task of finding and registering a driver for the database. Most of the major databases make their JDBC drivers readily available for use. Others may make you register prior to download of the driver, or in some cases purchase it. Since zxJDBC is an alternative implementation of JDBC, one must use a JDBC driver in order to use the API. Most JDBC drivers come in the format of a JAR file that can be installed to an application server container, and IDE. In order to make use of a particular database driver, it must reside within the CLASSPATH. As mentioned previously, to find a given JDBC driver for a particular database, take a look at the Sun Microsystems JDBC Driver search page (<http://developers.sun.com/product/jdbc/drivers>) as it contains a listing of different JDBC drivers for *most* of the databases available today.

Note: examples in this section are for Jython 2.5.1 and later. Jython 2.5.1 introduced some simplifications for working with connections and cursors. In addition, we assume PostgreSQL for most examples, using the world sample database (also available for MySQL). In order to follow along with the examples in the following sections, you should have a PostgreSQL database available with the *world* database example. Please go to the PostgreSQL homepage at <http://www.postgresql.org> to download the database. The world database sample is available with the source for this book. It can be installed into a PostgreSQL database by opening psql and initiating the following command:

```
postgres=# \i <path to world sql>/world.sql
```

As stated previously, once a driver has been obtained it must be placed into the classpath. Below are a few examples for adding JDBC drivers to the CLASSPATH for a couple of the most popular databases.

```
# Oracle

# Windows
set CLASSPATH=<PATH TO JDBC>\ojdbc14.jar;%CLASSPATH%

# OS X
export CLASSPATH=<PATH TO JDBC>/ojdbc14.jar:$CLASSPATH

# PostgreSQL

# Windows
set CLASSPATH=<PATH TO JDBC>\postgresql-x.x.jdbc4.jar;%CLASSPATH%

# OS X
export CLASSPATH<PATH TO JDBC>/postgresql-x.x.jdbc4.jar:$CLASSPATH
```

After the appropriate JAR for the target database has been added to the CLASSPATH, development can commence. It is important to note that zxJDBC (and all other JDBC implementations) use a similar procedure for working with the database. One must perform the following tasks to use a JDBC implementation:

- Create a connection
- Create a query or statement
- Obtain results of query or statement
- If using a query, obtain results in a cursor and iterate over data to perform tasks
- Close cursor
- Close connection (If not using the with_statement syntax in versions of Jython prior to 2.5.1)

Over the next few sections, we'll take a look at each of these steps and how zxJDBC can make them easier than using JDBC directly.

Connections

A database connection is simply a resource object that manages access to the database system. Because database resources are generally expensive objects to allocate, and can be readily exhausted, it is important to close them as soon as you're finished using them. There are two ways to create database connections:

- Direct creation. Standalone code, such as a script, will directly create a connection.
- JNDI. Code managed by a container should use JNDI for connection creation. Such containers include GlassFish, JBoss, Tomcat, WebLogic, and WebSphere. Normally connections are pooled when run in this context and are associated with a given security context.

Below is an example of the best way to create a database connection outside of a managed container using Jython 2.5.1. It is important to note that prior to 2.5.1, the `with_statement` syntax was not available. This is due to the underlying implementation of PyConnection in versions of Jython prior to 2.5.1. As a rule, any object that can be used via the `with_statement` must implement certain functionality, including the `__exit__` method. Please see the note below to find out how to implement this functionality in versions prior to 2.5.1. Another thing to notice is that in order to connect, we must use a JDBC url which conforms to the standards of a given database...in this case PostgreSQL.

```
from __future__ import with_statement
from com.ziclix.python.sql import zxJDBC

# for example
jdbc_url = "jdbc:postgresql:test"
username = "postgres"
password = "jython25"
driver = "org.postgresql.Driver"

with zxJDBC.connect(jdbc_url, username, password, driver) as conn:
    do_something(conn)
    # there is no need to close the cursor as the with statement takes care of this,
    ↪for us
```

Walking through the steps, you can see that the `with_statement` and `zxJDBC` are imported as we will use them to obtain our connection. The next step is to define a series of string values that will be used for the connection activity. Note that these only need to be defined once if set up as globals. Lastly, the connection is obtained and some work is done. Now let's take a look at this same procedure coded in Java for comparison.

```
import java.sql.*;
import org.postgresql.Driver;

...
// In some method
Connection conn = null;
String jdbc_url = "jdbc:postgresql:test";
String username = "postgres";
String password = "jython25";
String driver = "org.postgresql.Driver";
try {
    DriverManager.registerDriver(new org.postgresql.Driver());
    conn = DriverManager.getConnection(jdbc_url,
                                     username, password);
    // do something using statement and resultset
    conn.close();
}
catch(Exception e) {
    logWriter.error("getBeanConnection ERROR: ",e);
    return conn;
}
```

Note: In versions of Jython prior to 2.5.1, the `with_statement` syntax is not available. For this reason, we must work directly with the connection (ie. close it when finished). Take a look at the following code for an example of using `zxJDBC` connections without the `with_statement` functionality.

```
from __future__ import with_statement
from com.ziclix.python.sql import zxJDBC

# for example jdbc_url = "jdbc:postgresql:test" username = "postgres" password = "jython25" driver =
"org.postgresql.Driver"

conn = zxJDBC.connect(jdbc_url, username, password, driver) do_something(conn) # Be sure to clean up by closing
```

the connection (and cursor) `conn.close()`

The `with` statement ensures that the connection is immediately closed following the work. The alternative is to use `finally` to perform the close. Using the latter technique allows for more tightly controlled exception handling technique, but also adds a considerable amount of code. As noted previously, the `with` statement is not available in versions of Jython prior to 2.5.1, so this is the recommended approach when using those versions:

```
try:
    conn = zxJDBC.connect(jdbc_url, username, password, driver)
    do_something(conn)
finally:
    conn.close()
```

The connection (PyConnection) object in `zxJDBC` has a number of methods and attributes that can be used to perform various functions and obtain metadata information. For instance, the `close` method can be used to close the connection. The following tables are listings of all available methods and attributes for a connection and what they do.

Table 12-1: Connection Methods

Method	Functionality
<code>close</code>	Close the connection now (rather than whenever <code>__del__</code> is called).
<code>commit</code>	Commits all work that has been performed against a connection
<code>cursor</code>	Returns a new cursor object from the connection
<code>rollback</code>	In case a database does provide transactions this method causes the database to roll back to the start of any pending transaction.
<code>nativesql</code>	Converts the given SQL statement into the system's native SQL grammar

Table 12-2: Connection Attributes

Attribute	Functionality
<code>autocommit</code>	Enable or disable autocommit on a connection. Default is disabled.
<code>dbname</code>	Returns the name of the database
<code>dbversion</code>	Returns the version of database
<code>drivername</code>	Returns the database driver name
<code>driverversion</code>	Returns the database driver version
<code>url</code>	Returns the database URL in use
<code>__connection__</code>	Returns the type of connection in use
<code>__cursors__</code>	Returns a listing of all open cursors on the connection
<code>__statements__</code>	Returns a listing of all open statements on the connection
<code>closed</code>	Returns a boolean stating whether connection is closed

Of course, we can always use the connection to obtain a listing of all methods and attributes using the following syntax.:

```
>>> conn.__methods__
['close', 'commit', 'cursor', 'rollback', 'nativesql']
>>> conn.__members__
['autocommit', 'dbname', 'dbversion', 'drivername', 'driverversion', 'url', '__connection__', '__cursors__', '__statements__', 'closed']
```

Note: Connection pools help ensure for more robust operation, by providing for reuse of connections while ensuring the connections are in fact valid. Often naive code will hold a connection for a very long time, to avoid the overhead of creating a connection, and then go to the trouble of managing reconnecting in the event of a network or server failure. It's better to let that be managed by the connection pool infrastructure instead of reinventing it.

All transactions, if supported, are done within the context of a connection. We will be discussing transactions further in the subsection on data modification, but this is the basic recipe:

```
try:
    # Obtain a connection that is not using auto-commit (default for zxJDBC)
    conn = zxJDBC.connect(jdbc_url, username, password, driver)
    # Perform all work on connection
    do_something(conn)
    # After all work is complete, commit
    conn.commit
except:
    # If a failure occurs along the way, rollback all previous work
    conn.rollback()
```

zxJDBC.lookup

In a managed container, you would use `zxJDBC.lookup` instead of `zxJDBC.connect`. If you have code that needs to run both inside and outside containers, we recommend you use a factory to abstract this. Inside a container, like an app server, you should use JNDI to allocate the resource. Generally the connection will be managed by a connection pool:

```
factory = "com.sun.jndi.fscontext.RefFSContextFactory"
db = zxJDBC.lookup('jdbc/postgresDS'),
    INITIAL_CONTEXT_FACTORY=factory)
```

The above example assumes that the datasource defined in the container is named “jdbc/postgresDS”, and it uses the Sun FileSystem JNDI reference implementation. This lookup process does not require knowing the JDBC URL or the driver factory class. These aspects, as well as possibly the user name and password, are configured by the administrator of the container using tools specific to that container. Most often by convention you will find that JNDI names typically resemble a *jdbc/NAME* format.

Cursors

Once you have a connection, you probably want to do something with it. Since you can do multiple things within a transaction - query one table, update another - you need one more resource, which is a cursor. A cursor in `zxJDBC` is a wrapper around the JDBC statement and resultSet objects that provides a very *Pythonic* syntax for working with the database. The result, an easy to use and extremely flexible API. Cursors are used to hold data that has been obtained via the database, and they can be used in a variety of fashions which we will discuss. There are two types of cursors available for use, static and dynamic. A static cursor is the default type, and it basically performs an iteration of an entire resultSet at once. The latter dynamic cursor is known as a lazy cursor and it only iterates through the resultSet on an as-needed basis. Here are some examples of creating each type of cursor.

```
# Assume that necessary imports have been performed
# and that a connection has been obtained and assigned
# to a variable 'conn'

cursor = conn.cursor() # static cursor creation

cursor = conn.cursor(1) # dynamic cursor creation with the boolean argument
```

Dynamic cursors tend to perform better due to memory constraints, however, in some cases they are not as convenient as working with a static cursor. For example, if you’d like to query the database to find a row count it is very easy with

a static cursor because all rows are obtained at once. This is not possible with a dynamic cursor and one must perform two queries in order to achieve the same result.

```
# Using a static cursor to obtain rowcount
>>> cursor = conn.cursor()
>>> cursor.execute("select * from country")
>>> cursor.rowcount
239

# Using a dynamic cursor to obtain rowcount
>>> cursor = conn.cursor(1)
>>> cursor.execute("select * from country")
>>> cursor.rowcount
0

# Since rowcount does not work with dynamic, we must
# perform a separate count query to obtain information
>>> cursor.execute("select count(*) from country")
>>> cursor.fetchone()
(239L,)
```

Cursors are used to execute queries, inserts, updates, deletes, and/or issue database commands. Like connections, cursors have a number of methods and attributes that can be used to perform actions or obtain metadata information.

Table 12-3: Cursor Methods

Method	Functionality
tables	Retrieves a list of tables. (catalog, schema-pattern, table-pattern, types)
columns	Retrieves a list of columns. (catalog, schema-pattern, table-name-pattern, column-name-pattern)
primarykeys	Retrieves a list of primary keys. (catalog, schema, table)
foreignkeys	Retrieves a list of foreign keys. (primary-catalog, primary-schema, primary-table, foreign-catalog, foreign-schema, foreign-table)
procedures	Retrieves a list of procedures. (catalog, schema, tables)
procedurecolumns	Retrieves a list of procedure columns. (catalog, schema-pattern, procedure-pattern, column-pattern)
statistics	Obtains statistics on the query. (catalog, schema, table, unique, approximation)
bestrow	Optimal set of columns that uniquely identifies a row
version-columns	Columns that are automatically updated when any value in a row is updated
close	Closes the cursor
execute	Executes code contained within the cursor
executemany	Used to execute prepared statements or sql with a parameter list
fetchone	Fetch the next row of a query result set, returning a single sequence, or None if no more data exists
fetchall	Fetch all (remaining) rows of a query result, returning them as a sequence of sequences
fetchmany	Fetch the next set of rows of a query result, returning a sequence of sequences
callproc	Executes a stored procedure
next	Moves to the next row in the cursor
write	Execute the sql written to this file-like object

Table 12-4: Cursor Attributes

Attribute	Functionality
arraysize	Number of rows <i>fetchmany()</i> should return without any arguments
rowcount	Returns the number of resulting rows
rownumber	Returns the current row number
description	Returns information regarding each column in the query
datahandler	Returns the specified datahandler
warnings	Returns all warnings on the cursor
lastrowid	Returns the rowid of the last row fetched
updatecount	Returns the number of updates that the current cursor has performed
closed	Returns a boolean representing whether the cursor has been closed
connection	Returns the connection object that contains the cursor

A number of the methods and attributes above cannot be used until a cursor has been executed with a query or statement of some kind. Most of the time, the particular method or attribute name will provide a good enough description of it's functionality.

Creating and Executing Queries

As you've seen previously, it is quite easy to initiate a query against a given cursor. Simply provide a *select* statement in string format as a parameter to the cursor *execute()* or *executemany()* methods and then use one of the *fetch* methods to iterate over the returned results. In the following examples we query the world data and display some cursor data via the associated attributes and methods.

```
>>> cursor = conn.cursor()
>>> cursor.execute("select country, region from country")

# Fetch next record
>>> cursor.fetchone()
((AFG,Afghanistan,Asia,"Southern and Central Asia",652090,1919,22720000,45.9,5976.00,,
↳Afganistan/Afqanestan,"Islamic Emirate","Mohammad Omar",1,AF), u'Southern and
↳Central Asia')

# Calling fetchmany() without any parameters returns next record
>>> cursor.fetchmany()
[((NLD,Netherlands,Europe,"Western Europe",41526,1581,15864000,78.3,371362.00,360478.
↳00,Nederland,"Constitutional Monarchy",Beatrix,5,NL), u'Western Europe')]
```

```
# Fetch the next two records
>>> cursor.fetchmany(2)
[((ANT,"Netherlands Antilles","North America",Caribbean,800,,217000,74.7,1941.00,,
↳"Nederlandse Antillen","Nonmetropolitan Territory of The Netherlands",Beatrix,33,
↳AN), u'Caribbean'), ((ALB,Albania,Europe,"Southern Europe",28748,1912,3401200,71.6,
↳3205.00,2500.00,Shqip?ria,Republic,"Rexhep Mejdani",34,AL), u'Southern Europe')]
```

```
# Calling fetchall() would retrieve the rest of the records
>>> cursor.fetchall()
...
```

```
# Using description provides data regarding the query in the cursor
>>> cursor.description
[('country', 1111, 2147483647, None, None, None, 2), ('region', 12, 2147483647, None,
↳None, None, 0)]
```

Creating a cursor using the `with_statement` syntax is easy, please take a look at the following example for use with Jython 2.5.1 and beyond.

```
with conn.cursor() as c:
    do_some_work(c)
```

Like connections, you need to ensure the resource is appropriately closed. Of course if you are using Jython from the shell, there's generally no need to worry about resource allocations. So you can just do this to follow the shorter examples we will look at:

```
>>> c = conn.cursor()
>>> # work with cursor
```

As you can see, queries are easy to work with using cursors. In the example above, we used the *fetchall()* method to retrieve all of the results of the query. However, there are other options available for cases where all results are not desired including the *fetchone()* and *fetchmany()* options. Sometimes it is best to iterate over results of a query in order to work with each record separately. The following example iterates over the countries contained within the country table.

```
>>> from com.ziclix.python.sql import zxJDBC
>>> conn = zxJDBC.connect("jdbc:postgresql:test","postgres","jython25","org.
↳postgresql.Driver")
>>> cursor = conn.cursor()
>>> cursor.execute("select name from country")
>>> while cursor.next():
...     print cursor.fetchone()
...
(u'Netherlands Antilles',)
(u'Algeria',)
(u'Andorra',)
...
```

Often times, queries are not hard-coded and we need the ability to substitute values in the query to select the data that our application requires. Developers also need a way to create dynamic SQL statements at times. Of course, there are multiple ways to perform these feats. The easiest way to substitute variables or create a dynamic query is to simply use string concatenation. After all, the *execute()* method takes a string-based query. The following example shows how to use string concatenation for dynamically forming a query and also substituting variables.

```
# Assume that the user selected a pull-down menu choice determining
# what results to retrieve from the database, either continent or country name.
# The selected choice is stored in the selectedChoice variable. Let's also assume
# that we are interested in all continents or countries beginning with the letter "A"

>>> qry = "select " + selectedChoice + " from country where " + selectedChoice + "
↳like 'A%'"
>>> cursor.execute(qry)
>>> while cursor.next():
...     print cursor.fetchone()
...
(u'Albania',)
(u'American Samoa',)
...
```

This technique works very well for creating dynamic queries, but it also has its share of issues. For instance, reading through concatenated strings of code can become troublesome on the eyes. Maintaining such code is a tedious task. Above that, string concatenation is not the safest way to construct a query as it opens an application up for a SQL injection attack. SQL injection is a technique that is used to pass undesirable SQL code into an application in such a way that it alters a query to perform unwanted tasks. If the user has the ability to type free text into a textfield and have that text passed into a string concatenated query, it is best to perform some other means of filtering to ensure certain

keywords or commenting symbols are not contained in the value. A better way of getting around these issues is to make use of prepared statements.

Note: Ideally, never construct a query statement directly from user data. SQL injection attacks employ such construction as their attack vector. Even when not malicious, user data will often contain characters, such as quotation marks, that can cause the query to fail if not properly escaped. In all cases, it's important to scrub and then escape the user data before it's used in the query.

One other consideration is that such queries will generally consume more resources unless the database statement cache is able to match it (if at all).

But there are two important exceptions to our recommendation:

- SQL statement requirements. Bind variables cannot be used everywhere. However, specifics will depend on the database.
- Ad hoc or unrepresentative queries. In databases like Oracle, the statement cache will cache the execution plan, without taking in account lopsided distributions of values that are indexed - but are known to the database if presented literally. In those cases, a more efficient execution plan will result if the value is put in the statement directly.

However, even in these exceptional cases, it's imperative that any user data is fully scrubbed. A good solution is to use some sort of mapping table, either an internal dictionary or driven from the database itself. In certain cases, a carefully constructed regular expression may also work. Be careful.

Prepared Statements

To get around using the string concatenation technique for substituting variables, we can use a technique known as *prepared statements*. Prepared Statements allow one to use bind variables for data substitution, and they are generally safer to use because most security considerations are taken care of without developer interaction. However, it is always a good idea to filter input to help reduce the risk. Prepared Statements in zxJDBC work the same as they do in JDBC, just a simpler syntax. In the example below, we will perform a query on the country table using a prepared statement. Note that the question marks are used as place holders for the substituted variables. It is also important to note that the *executemany()* method is invoked when using a prepared statement. Any substitution variables being passed into the prepared statement must be in the form of a tuple or list.

```
# Passing a string value into the query
qry = "select continent from country where name = ?"
>>> cursor.executemany(qry, ['Austria'])
>>> cursor.fetchall()
[(u'Europe',)]

# Passing some variables into the query
>>> continent1 = 'Asia'
>>> continent2 = 'Africa'
>>> qry = "select name from country where continent in (?,?)"
>>> cursor.executemany(qry, [continent1, continent2])
>>> cursor.fetchall()
[(u'Afghanistan',), (u'Algeria',), (u'Angola',), (u'United Arab Emirates',), (u
↪ 'Armenia',), (u'Azerbaijan',),
...
```


Resource Management

You should always close connections and cursors. This is not only good practice but absolutely essential in a managed container so as to avoid exhausting the corresponding connection pool, which needs the connections returned as soon as they are no longer in use. The `with` statement makes it easy:

```
from __future__ import with_statement
from itertools import islice
from com.ziclix.python.sql import zxJDBC

# externalize
jdbc_url = "jdbc:oracle:thin:@host:port:sid"
username = "world"
password = "world"
driver = "oracle.jdbc.driver.OracleDriver"

with zxJDBC.connect(jdbc_url, username, password, driver) as conn:
    with conn.cursor() as c:
        c.execute("select * from emp")
        for row in islice(c, 20):
            print row # let's redo this w/ namedtuple momentarily...
```

The older alternative is available. It's more verbose, and similar to the Java code that would normally have to be written to ensure that the resource is closed.

```
try:
    conn = zxJDBC.connect(jdbc_url, username, password, driver)
    cursor = conn.cursor()
    #do something with the cursor
# Be sure to clean up by closing the connection (and cursor)
finally:
    cursor.close()
    conn.close()
```

Metadata

As mentioned previously in this chapter, it is possible to obtain metadata information via the use of certain attributes that are available to both connection and cursor objects. `zxJDBC` matches these attributes to the properties that are found in the JDBC `java.sql.DatabaseMetaData` object. Therefore, when one of these attributes is called, the JDBC `DatabaseMetaData` object is actually obtaining the information.

The following examples show how to retrieve metadata about a connection, cursor, or even a specific query. Note that whenever obtaining metadata about a cursor, you must fetch the data after setting up the attributes.

```
# Obtain information about the connection using connection attributes
>>> conn.dbname
'PostgreSQL'
>>> conn.dbversion
'8.4.0'
>>> conn.drivename
'PostgreSQL Native Driver'
# Check for existing cursors
>>> conn.__cursors__
[<PyExtendedCursor object instance at 1>]

# Obtain information about the cursor and the query
```

```
>>> cursor = conn.cursor()
# List all tables
>>> cursor.tables(None, None, '%', ('TABLE',))
>>> cursor.fetchall()
[(None, u'public', u'city', u'TABLE', None), (None, u'public', u'country', u'TABLE',
↪None), (None, u'public', u'countrylanguage', u'TABLE', None), (None, u'public', u
↪'test', u'TABLE', None)]
```

Data Manipulation Language and Data Definition Language

Any application that will manipulate data contained in a dbms must be able to issue Data Manipulation Language (DML). Of course, DML consists of issuing statements such as INSERT, UPDATE, and DELETE...the basics of CRUD programming. zxJDBC makes it rather easy to use DML in a standard cursor object. When doing so, the cursor will return a value to provide information about the result. A standard DML transaction in JDBC uses a prepared statement with the cursor object, and assigns the result to a variable that can be read afterwards to determine whether the statement succeeded.

zxJDBC also uses cursors to define new constructs in the database using Data Definition Language (DDL). Examples of doing such are creating tables, altering tables, creating indexes, and the like. Similarly to performing DML with zxJDBC, a resulting DDL statement returns a value to assist in determining whether the statement succeeded or not.

In the next couple of examples, we'll create a table, insert some values, delete values, and finally delete the table.

```
# Create a table named PYTHON_IMPLEMENTATIONS
>>> stmt = "create table python_implementations (id integer, python_implementation_
↪varchar, current_version varchar)"
>>> result = cursor.execute(stmt)
>>> print result
None
>>> cursor.tables(None, None, '%', ('TABLE',))
# Ensure table was created
>>> cursor.fetchall()
[(None, u'public', u'city', u'TABLE', None), (None, u'public', u'country', u'TABLE',
↪None), (None, u'public', u'countrylanguage', u'TABLE', None), (None, u'public', u
↪'python_implementations', u'TABLE', None), (None, u'public', u'test', u'TABLE',
↪None)]

# Insert some values into the table
>>> stmt = "insert into PYTHON_IMPLEMENTATIONS values (?, ?, ?)"
>>> result = cursor.executemany(stmt, [1, 'Jython', '2.5.1'])
>>> result = cursor.executemany(stmt, [2, 'CPython', '3.1.1'])
>>> result = cursor.executemany(stmt, [3, 'IronPython', '2.0.2'])
>>> result = cursor.executemany(stmt, [4, 'PyPy', '1.1'])
>>> conn.commit()

# Query the database
>>> cursor.execute("select python_implementation, current_version from python_
↪implementations")
>>> cursor.rowcount
4
>>> cursor.fetchall()
[(u'Jython', u'2.5.1'), (u'CPython', u'3.1.1'), (u'IronPython', u'2.0.2'), (u'PyPy', u
↪'1.1')]

# Update values and re-query
>>> stmt = "update python_implementations set python_implementation = 'CPython -
↪Standard Implementation' where id = 2"
```

```
>>> result = cursor.execute(stmt)
>>> print result
None
>>> conn.commit()
>>> cursor.execute("select python_implementation, current_version from python_
↳implementations")
>>> cursor.fetchall()
[(u'Jython', u'2.5.1'), (u'IronPython', u'2.0.2'), (u'PyPy', u'1.1'), (u'CPython -
↳Standard Implementation', u'3.1.1')]
```

It is a good practice to make use of bulk inserts and updates. Each time a commit is issued it incurs a performance penalty. If DML statements are grouped together and then followed by a commit, the resulting transaction will perform much better. Another good reason to use bulk DML statements is to ensure transactional safety. It is likely that if one statement in a transaction fails, all others should be rolled back. As mentioned previously in the chapter, using a try/except clause will maintain transactional dependencies. If one statement fails then all others will be rolled back. Likewise, if they all succeed then they will be committed to the database with one final commit.

Calling Procedures

Database applications often times make use of procedures and functions that live inside the database. Most often these procedures are written in a SQL procedural language such as Oracle's PL/SQL or PostgreSQL's PL/pgSQL. Writing database procedures and using them with external applications such as written in Python, Java, or the like makes lots of sense because procedures are often the easiest way to work with data. Not only are they running close to the metal since they are in the database, but they also perform much faster than say a Jython application that needs to connect and close connections on the database. Since a procedure lives within the database, there is no performance penalty due to connections being made.

zxJDBC can easily invoke a database procedure just as JDBC can do. This helps developers to create applications that have some of the more database-centric code residing within the database as procedures, and other application-specific code running on the application server and interacting seamlessly with the database. In order to make a call to a database procedure, zxJDBC offers the *callproc()* method which takes the name of the procedure to be invoked. In the following example, we create a relatively useless procedure and then call it using Jython.

PostgreSQL Procedure

```
CREATE OR REPLACE FUNCTION proc_test(
    OUT out_parameter CHAR VARYING(25) )
AS $$
DECLARE
BEGIN
    SELECT python_implementation
        INTO out_parameter
        FROM python_implementations
        WHERE id = 1;

    RETURN;
END;
$$ LANGUAGE plpgsql;
```

Jython Calling Code

```
>>> result = cursor.callproc('proc_test')
>>> cursor.fetchall()
[(u'Jython',)]
```

Although this example was relatively trivial, it is easy to see how the use of database procedures from zxJDBC could

easily become important. Combining database procedures and functions with application code is a powerful technique, but it does tie an application to a specific database so it should be used wisely.

Customizing zxJDBC Calls

At times, it is convenient to have the ability to alter or manipulate a SQL statement automatically. This can be done before the statement is sent to the database, after it is sent to the database, or even just to obtain information about the statement that has been sent. To manipulate or customize data calls, it is possible to make use of the *DataHandler* interface that is available via zxJDBC. There are basically three different methods for handling type mappings when using DataHandler. They are called at different times in the process, one when fetching and the other when binding objects for use in a prepared statement. These datatype mapping callbacks are categorized into four different groups: life cycle, developer support, binding prepared statements, and building results.

At first mention, customizing and manipulating statements can seem overwhelming and perhaps even a bit daunting. However, the zxJDBC DataHandler makes this task fairly trivial. Simply create a handler class and implement the functionality that is required by overriding a given handler method. Below is a listing of the various methods that can be overridden, and we'll look at a simple example afterwards.

Life Cycle

public void preExecute(Statement stmt) throws SQLException; A callback prior to each execution of the statement. If the statement is a PreparedStatement (created when parameters are sent to the execute method), all the parameters will have been set.

public void postExecute(Statement stmt) throws SQLException; A callback after successfully executing the statement. This is particularly useful for cases such as auto-incrementing columns where the statement knows the inserted value.

Developer Support

public String getMetaDataName(String name); A callback for determining the proper case of a name used in a DatabaseMetaData method, such as getTables(). This is particularly useful for Oracle which expects all names to be upper case.

public PyObject getRowId(Statement stmt) throws SQLException; A callback for returning the row id of the last insert statement.

Binding Prepared Statements

public Object getJDBCObject(PyObject object, int type); This method is called when a PreparedStatement is created through use of the execute method. When the parameters are being bound to the statement, the DataHandler gets a callback to map the type. *This is only called if type bindings are present.*

public Object getJDBCObject(PyObject object); This method is called when no type bindings are present during the execution of a PreparedStatement.

Building Results

public PyObject getPyObject(ResultSet set, int col, int type); This method is called upon fetching data from the database. Given the JDBC type, return the appropriate PyObject subclass from the Java object at column col in the ResultSet set.

Now we'll examine a simple example of utilizing this technique. The recipe basically follows these steps:

1. Create a handler class to implement a particular functionality (must implement the DataHandler interface)
2. Assign the created handler class to a given cursor object
3. Use the cursor object to make database calls

In the following example, we override the *preExecute* method to print a message stating that the functionality has been altered. As you can see, it is quite easy to do and opens up numerous possibilities.

PyHandler.py

```
from com.ziclix.python.sql import DataHandler

class PyHandler(DataHandler):
    def __init__(self, handler):
        self.handler = handler
        print 'Inside DataHandler'
    def getPyObject(self, set, col, datatype):
        return self.handler.getPyObject(set, col, datatype)
    def getJDBCObject(self, object, datatype):
        print "handling prepared statement"
        return self.handler.getJDBCObject(object, datatype)
    def preExecute(self, stmt):
        print "calling pre-execute to alter behavior"
        return self.handler.preExecute(stmt)
```

Jython Interpreter Code

```
>>> cursor.datahandler = PyHandler(cursor.datahandler)
Inside DataHandler
>>> cursor.execute("insert into test values (?,?)", [1,2])
calling pre-execute
```

History

zxJDBC was contributed by Brian Zimmer, one-time lead committer for Jython. This API was written to enable Jython developers to have the capability of working with databases using techniques that more closely resembled the Python DB API. The package eventually became part of the Jython distribution and today it is one of the most important underlying APIs for working with higher level frameworks such as Django. The zxJDBC API is evolving at the time of this publication, and it is likely to become more useful in future releases.

Object Relational Mapping

While zxJDBC certainly offers a viable option for database access via Jython, there are many other solutions available. Many developers today are choosing to use ORM (Object Relational Mapping) solutions to work with the database. This section is not an introduction to ORM, we assume that you are at least a bit familiar with the topic. Furthermore, the ORM solutions that are about to be discussed have an enormous amount of very good documentation already available either on the web or in book format. Therefore, this section will give insight on how to use these technologies with Jython, but it will not go into great detail on how each ORM solution works. With that said, there is no doubt in stating that these solutions are all very powerful and capable for standalone and enterprise applications alike.

In the next couple of sections, we'll cover how to use some of the most popular ORM solutions available today with Jython. You'll learn how to set up your environment and how to code Jython to work with each ORM. By the end of this chapter, you should have enough knowledge to begin working with these ORMs using Jython, and even start building Jython ORM applications.

SqlAlchemy

No doubt about it, SqlAlchemy is one of the most widely known and used ORM solutions for the Python programming language. It has been around long enough that its maturity and stability make it a great contender for use in your

applications. It is simple to setup, and easy-to-use for both new databases and legacy databases alike. You can seriously download and install SQLAlchemy and begin using it in a very short amount of time. The syntax for using this solution is very straight forward, and as with other ORM technologies, working with database entities occurs via the use of a mapper that links a special Jython class to a particular table in the database. The overall result is that the application is persisted through the use of entity classes as opposed to database SQL transactions.

In this section we will cover the installation and configuration of SQLAlchemy with Jython. It will then show you how to get started using it through a few short examples, we will not get into great detail as there are plenty of excellent references on SQLAlchemy already. However, this section should fill in the gaps for making use of this great solution on Jython.

Installation

We'll begin by downloading SQLAlchemy from the website (<http://www.sqlalchemy.org>), at the time of this writing the version that should be used is 0.6. This version has been installed and tested with the Jython 2.5.0 release. Once you've downloaded the package, unzip it to a directory on your workstation and then traverse to that directory in your terminal or command prompt. Once you are inside of your SQLAlchemy directory, issue the following command to install:

```
jython setup.py install
```

Once you've completed this process, SQLAlchemy should be successfully installed into your jython Libsite-packages directory. You can now access the SQLAlchemy modules from jython, and you can open up your terminal and check to ensure that the install was a success by importing sqlalchemy and checking the version.

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
'0.6beta1'
>>>
```

After we've ensured that the installation was a success, it is time to begin working with SQLAlchemy via the terminal. However, we have one step left before we can begin. Jython uses zxJDBC to implement the Python database API in Java. The end result is that most of the dialects that are available for use with SQLAlchemy will not work with Jython out of the box. This is because the dialects need to be re-written to implement zxJDBC. At the time of this writing, I could only find one completed dialect, zxoracle, that was rewritten to use zxJDBC, and I'll be showing you some examples based upon zxoracle in the next sections. However, other dialects are in the works including SQLServer and MySQL. The bad news is that SQLAlchemy will not yet work with every database available, on the other hand, Oracle is a very good start and implementing a new dialect is not very difficult. You can find the zxoracle.py dialect included in the source for this book. Browse through it and you will find that it may not be too difficult to implement a similar dialect for the database of your choice. You can either place zxoracle somewhere on your Jython path, or place it into the Lib directory in your Jython installation.

Lastly, we will need to ensure that our database JDBC driver is somewhere on our path so that Jython can access it. Once you've performed the procedures included in this section, start up jython and practice some basic SQLAlchemy using the information from the next couple of sections.

Using SQLAlchemy

We can work directly with SQLAlchemy via the terminal or command line. There is a relatively basic set of steps you'll need to follow in order to work with it. First, import the necessary modules for the tasks you plan to perform. Second, create an engine to use while accessing your database. Third, create your database tables if you have not yet done so, and map them to Python classes using a SQLAlchemy mapper. Lastly, begin to work with the database.

Now there are a couple of different ways to do things in this technology, just like any other. For instance, you can either follow a very granular process for table creation, class creation, and mapping that involves separate steps for each, or you can use what is known as a declarative procedure and perform all of these tasks at the same time. I will show you how to do each of these in this chapter, along with performing basic database activities using SQLAlchemy. If you are new to SQLAlchemy, I suggest reading through this section and then going to sqlalchemy.org and reading through some of the large library of documentation available there. However, if you're already familiar with SQLAlchemy, you can move on if you wish because the rest of this section is a basic tutorial of the ORM solution itself.

Our first step is to create an engine that can be used with our database. Once we've got an engine created then we can begin to perform database tasks making use of it. Type the following lines of code in your terminal, replacing database specific information with the details of your development database.

```
>>> import zxoracle
>>> from sqlalchemy import create_engine
>>> db = create_engine('zxoracle://schema:password@hostname:port/database')
```

Next, we'll create the metadata that is necessary to create our database table using SQLAlchemy. You can create one or more tables via metadata, and they are not actually created until after the metadata is applied to your database engine using a `create_all()` call on the metadata. In this example, I am going to walk you through the creation of a table named `Player` that will be used in an application example in the next section.

```
>>> player = Table('player', metadata,
...     Column('id', Integer, primary_key=True),
...     Column('first', String(50)),
...     Column('last', String(50)),
...     Column('position', String(30)))
>>> metadata.create_all(engine)
```

Our table should now exist in the database and the next step is to create a Python class to use for accessing this table.

```
class Player(object):
    def __init__(self, first, last, position):
        self.first = first
        self.last = last
        self.position = position

    def __repr__(self):
        return "<Player('%s', '%s', '%s')>" % (self.first, self.last, self.position)
```

The next step is to create a mapper to correlate the `Player` python object and the `player` database table. To do this, we use the `mapper()` function to create a new `Mapper` object binding the class and table together. The mapper function then stores the object away for future reference.

```
>>> from sqlalchemy.orm import mapper
>>> mapper(Player, player)
<Mapper at 0x4; Player>
```

Creating the mapper is the last step in the process of setting up the environment to work with our table. Now, let's go back and take a quick look at performing all of these steps in an easier way. If we want to create a table, class, and mapper all at once then we can do this declaratively. Please note that with the Oracle dialect, we need to use a sequence to generate the auto-incremented id column for the table. To do so, import the `sqlalchemy.schema.Sequence` object and pass it to the id column when creating. You must ensure that you've manually created this sequence in your Oracle database or this will not work.

```
SQL> create sequence id_seq
2 start with 1
3 increment by 1;
```

```
Sequence created.
```

```
# Declarative creation of the table, class, and mapper
>>> from sqlalchemy.ext.declarative import declarative_base
>>> from sqlalchemy.schema import Sequence
>>> Base = declarative_base()
>>> class Player(object):
...     __tablename__ = 'player'
...     id = Column(Integer, Sequence('id_seq'), primary_key=True)
...     first = Column(String(50))
...     last = Column(String(50))
...     position = Column(String(30))
...     def __init__(self, first, last, position):
...         self.first = first
...         self.last = last
...         self.position = position
...     def __repr__(self):
...         return "<Player('%s','%s','%s')>" % (self.first, self.last, self.position)
...
```

It is time to create a session and begin working with our database. We must create a session class and bind it to our database engine that was defined with `create_engine` earlier. Once created, the Session class will create new session object for our database. The Session class can also do other things that are out of scope for this section, but you can read more about them at sqlalchemy.org or other great references available on the web.

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=db)
```

We can start to create Player objects now and save them to our session. The objects will be persisted in the database once they are needed, this is also known as a `flush()`. If we create the object in the session and then query for it, sqlalchemy will first persist the object to the database and then perform the query.

```
# Import sqlalchemy module and zxoracle
>>> import zxoracle
>>> from sqlalchemy import create_engine
>>> from sqlalchemy import Table, Column, String, Integer, MetaData, ForeignKey
>>> from sqlalchemy.schema import Sequence

# Create engine
>>> db = create_engine('zxoracle://schema:password@hostname:port/database')

# Create metadata and table
>>> metadata = MetaData()
>>> player = Table('player', metadata,
...     Column('id', Integer, Sequence('id_seq'), primary_key=True),
...     Column('first', String(50)),
...     Column('last', String(50)),
...     Column('position', String(30)))
>>> metadata.create_all(db)

# Create class to hold table object
>>> class Player(object):
...     def __init__(self, first, last, position):
...         self.first = first
...         self.last = last
...         self.position = position
```



```

...     def __repr__(self):
...         return "<Player('%s','%s','%s')>" % (self.first, self.last, self.position)

# Create mapper to map the table to the class
>>> from sqlalchemy.orm import mapper
>>> mapper(Player, player)
<Mapper at 0x4; Player>

# Create Session class and bind it to the database
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=db)
>>> session = Session()

# Create player objects, add them to the session
>>> player1 = Player('Josh', 'Juneau', 'forward')
>>> player2 = Player('Jim', 'Baker', 'forward')
>>> player3 = Player('Frank', 'Wierzbicki', 'defense')
>>> player4 = Player('Leo', 'Soto', 'defense')
>>> player5 = Player('Vic', 'Ng', 'center')
>>> session.add(player1)
>>> session.add(player2)
>>> session.add(player3)
>>> session.add(player4)
>>> session.add(player5)

# Query the objects
>>> forwards = session.query(Player).filter_by(position='forward').all()
>>> forwards
[<Player('Josh','Juneau','forward')>, <Player('Jim','Baker','forward')>]
>>> defensemen = session.query(Player).filter_by(position='defense').all()
>>> defensemen
[<Player('Frank','Wierzbicki','defense')>, <Player('Leo','Soto','defense')>]
>>> center = session.query(Player).filter_by(position='center').all()
>>> center
[<Player('Vic','Ng','center')>]

```

Well, hopefully from this example you can see the benefits of using SQLAlchemy. Of course, you can perform all of the necessary SQL actions such as insert, update, select, and delete against the objects. However, as said before there are many very good tutorials where you can learn how to do these things. We've barely scratched the surface of what you can do with SQLAlchemy, it is a very powerful tool to add to any Jython or Python developer's arsenal.

Hibernate

Hibernate is a very popular object relational mapping solution used in the Java world. As a matter of fact, it is so popular that many other ORM solutions are either making use of hibernate or extending it in various ways. As Jython developers, we can make use of Hibernate to create powerful hybrid applications. Since Hibernate works by mapping POJO (plain old Java object) classes to database tables, we cannot map our Jython objects to it directly. While we could always try to make use of an object factory to coerce our Jython objects into a format that hibernate could use, this approach leaves a bit to be desired. Therefore, if you wish to create an application coded entirely using Jython, this would probably not be the best ORM solution. However, most Jython developers are used to doing a bit of work in Java and as such, they can harness the maturity and power of the hibernate API to create first-class hybrid applications. This section will show you how to create database persistence objects using Hibernate and Java, and then use them directly from a Jython application. The end result, code the entity POJOs in Java, place them into a JAR file along with hibernate and all required mapping documents, and then import the JAR into your Jython application and use.

I have found that the easiest way to create such an application is to make use of an IDE like Eclipse or Netbeans. Then

create two separate projects, one of the projects would be a pure Java application that will include the entity beans. The other project would be a pure Jython application that would include everything else. In this situation, you could simply add resulting JAR from your Java project into the `sys.path` of your Jython project and you'll be ready to go. However, this works just as well if you do not wish to use an IDE.

It is important to note that this section will provide you with one use case for using Jython, Java, and Hibernate together. There may be many other scenarios in which this combination of technologies would work out just as well, if not better. It is also good to note that this section will not cover hibernate in any great depth; we'll just scratch the surface of what it is capable of doing. There are a plethora of great hibernate tutorials available on the web if you find this solution to be useful.

Entity Classes and Hibernate Configuration

Since our hibernate entity beans must be coded in Java, most of the hibernate configuration will reside in your Java project. Hibernate works in a straightforward manner. You basically map a table to a POJO and use a configuration file to map the two together. It is also possible to use annotations as opposed to XML configuration files, but for the purposes of this use case I will show you how to use the configuration files.

The first configuration file we need to assemble is the `hibernate.cfg.xml`, which goes in the root of your Java project. The purpose of this file is to define your database connection information as well as declare which entity configuration files will be used in your project. For the purposes of this example, we will be using the PostgreSQL database, and we'll be using one of my classic examples of the hockey roster application. This makes for a very simple use-case as we only deal with one table here, the `Player` table. Hibernate makes it very possible to work with multiple tables and even associate them in various ways.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.
0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">org.postgresql.Driver</property>
    <property name="connection.url">jdbc:postgresql://localhost/database-name</
    <property>
    <property name="connection.username">username</property>
    <property name="connection.password">password</property>
    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">1</property>
    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>
    <mapping resource="org/jythonbook/entity/Player.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Our next step is to code the plain old Java object for our database table. In this case, we'll code an object named `Player` that contains only four database columns: `id`, `first`, `last`, and `position`. As you'll see, we use standard public accessor methods with private variables in this class.

```
package org.jythonbook.entity;

public class Player {

    public Player() {}

    private long id;
    private String first;
```

```

private String last;
private String position;

public long getId(){
    return this.id;
}

private void setId(long id){
    this.id = id;
}

public String getFirst(){
    return this.first;
}

public void setFirst(String first){
    this.first = first;
}

public String getLast(){
    return this.last;
}

public void setLast(String last){
    this.last = last;
}

public String getPosition(){
    return this.position;
}

public void setPosition(String position){
    this.position = position;
}
}

```

Lastly, we will create a configuration file that will be used by hibernate to map our POJO to the database table itself. We'll ensure that the primary key value is always populated by using a generator class type of increment. Hibernate also allows for the use of other generators, including sequences if desired. The player.hbm.xml file should go into the same package as our POJO, in this case, the org.jythonbook.entity package.

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping
package="org.jythonbook.entity">

    <class name="Player" table="player" lazy="true">
        <comment>Player for Hockey Team</comment>

        <id name="id" column="id">
            <generator class="increment"/>
        </id>

        <property name="first" column="first"/>
        <property name="last" column="last"/>
    </class>
</hibernate-mapping>

```

```
<property name="position" column="position"/>

</class>

</hibernate-mapping>
```

That is all we have to do inside of the Java project for our simple example. Of course, you can add as many entity classes as you'd like to your own project. The main point to remember is that all of the entity classes are coded in Java, and we will code the rest of the application in Jython.

Jython Implementation Using the Java Entity Classes

The remainder of our use-case will be coded in Jython. Although all of the hibernate configuration files and entity classes are coded and place within the Java project, we'll need to import that project into the Jython project, and also import the hibernate JAR file so that we can make use of it's database session and transactional utilities to work with the entities. In the case of Netbeans, you'd create a Python application then set the Python platform to Jython 2.5.0. After that, you should add all of the required hibernate JAR files as well as the Java project JAR file to the Python path from within the project properties. Once you've set up the project and taken care of the dependencies, you're ready to code the implementation.

As said previously, for this example we are coding a hockey roster implementation. The application runs on the command line and basically allows one to add players to a roster, remove players, and check the current roster. All of the database transactions will make use of the Player entity we coded in our Java application, and we'll make use of hibernate's transaction management from within our Jython code.

```
# HockeyRoster.py
#
# Implemenatation logic for the HockeyRoster application

# Import Player class from the Player module
from org.hibernate.cfg import Environment
from org.hibernate.cfg import Configuration
from org.hibernate import Query
from org.hibernate import Session
from org.hibernate import SessionFactory
from org.hibernate import Transaction
from org.jythonbook.entity import Player
import sys

# Define a list to hold each of te Player objects
playerList = []
factory = None

# makeSelection()
#
# Creates a selector for our application. The function prints output to the
# command line. It then takes a parameter as keyboard input at the command line
# in order to choose our application option.

def makeSelection():
    validOptions = ['1','2','3','4','5']
    print "Please chose an option\n"

    selection = raw_input("Press 1 to add a player, 2 to print the roster, 3 to ↵
↳search for a player on the team, 4 to remove player, 5 to quit: ")
    if selection not in validOptions:
```

```

        print "Not a valid option, please try again\n"
    else:
        if selection == '1':
            addPlayer()
        elif selection == '2':
            printRoster()
        elif selection == '3':
            searchRoster()
        elif selection == '4':
            removePlayer()
        else:
            global runApp
            runApp = False
            print "Thanks for using the HockeyRoster application."

# addPlayer()
#
# Accepts keyboard input to add a player object to the roster list. This function
# creates a new player object each time it is invoked and appends it to the list.
def addPlayer():
    addNew = 'Y'
    print "Add a player to the roster by providing the following information\n"
    while addNew.upper() == 'Y':
        first = raw_input("First Name: ")
        last = raw_input("Last Name: ")
        position = raw_input("Position: ")
        id = len(playerList)
        session = factory.openSession()
        try:
            tx = session.beginTransaction()
            player = Player()
            player.first = first
            player.last = last
            player.position = position
            session.save(player)
            tx.commit()
        except Exception, e:
            if tx!=None:
                tx.rollback()
            print e
        finally:
            session.close()

        # playerList.append(player)
        print "Player successfully added to the roster\n"
        addNew = raw_input("Add another? (Y or N)")
    makeSelection()

# printRoster()
#
# Prints the contents of the list to the command line as a report
def printRoster():
    print "=====\n"
    print "Complete Team Roster\n"
    print "=====\n\n"
    playerList = returnPlayerList()
    for player in playerList:
        print "%s %s - %s" % (player.first, player.last, player.position)

```

```

print "\n"
print "=== End of Roster ===\n"
makeSelection()

# searchRoster()
#
# Takes input from the command line for a player's name to search within the
# roster list. If the player is found in the list then an affirmative message
# is printed. If not found, then a negative message is printed.
def searchRoster():
    index = 0
    found = False
    print "Enter a player name below to search the team\n"
    first = raw_input("First Name: ")
    last = raw_input("Last Name: ")
    position = None
    playerList = returnPlayerList()
    while index < len(playerList):
        player = playerList[index]
        if player.first.upper() == first.upper() and player.last.upper() == last.
↪upper():
            found = True
            position = player.position
            index = index + 1
    if found:
        print '%s %s is in the roster as %s' % (first, last, position)
    else:
        print '%s %s is not in the roster.' % (first, last)
    makeSelection()

def removePlayer():
    index = 0
    found = False
    print "Enter a player name below to remove them from the team roster\n"
    first = raw_input("First Name: ")
    last = raw_input("Last Name: ")
    position = None
    playerList = returnPlayerList()
    foundPlayer = Player()
    while index < len(playerList):
        player = playerList[index]
        if player.first.upper() == first.upper() and player.last.upper() == last.
↪upper():
            found = True
            foundPlayer = player
            index = index + 1
    if found:
        print '%s %s is in the roster as %s, are you sure you wish to remove?' %_
↪(foundPlayer.first, foundPlayer.last, foundPlayer.position)
        yesno = raw_input("Y or N")
        if yesno.upper() == 'Y':
            session = factory.openSession()
            try:
                delQuery = "delete from Player player where id = %s" % (foundPlayer.
↪id)

                tx = session.beginTransaction()
                q = session.createQuery(delQuery)

```

```

        q.executeUpdate()
        tx.commit()
        print 'The player has been removed from the roster', foundPlayer.id
    except Exception,e:
        if tx!=None:
            tx.rollback()
        print e
    finally:
        session.close
    else:
        print 'The player will not be removed'
else:
    print '%s %s is not in the roster.' % (first, last)
makeSelection()

def returnPlayerList():
    session = factory.openSession()
    try:
        tx = session.beginTransaction()
        playerList = session.createQuery("from Player").list()
        tx.commit()
    except Exception,e:
        if tx!=None:
            tx.rollback()
        print e
    finally:
        session.close
    return playerList

# main
#
# This is the application entry point. It simply prints the applicaion title
# to the command line and then invokes the makeSelection() function.
if __name__ == "__main__":
    print "Hockey Roster Application\n\n"
    cfg = Configuration().configure()

    factory = cfg.buildSessionFactory()
    global runApp
    runApp = True
    while runApp:
        makeSelection()

```

We begin our implementation in the main block, where the hibernate configuration is loaded. All of the hibernate configuration resides within the Java project, so we are not working with XML here, just making use of it. The code then begins to branch so that various tasks can be performed. In the case of adding a player to the roster, a user could enter the number 1 at the command prompt. You can see that the `addPlayer()` function simply creates a new `Player` object, populates it, and saves it into the database. Likewise, the `searchRoster()` function calls another function named `returnPlayerList()` which queries the player table using hibernate query language and returns a list of `Player` objects.

In the end, we have a completely scalable solution. We can code our entities using a mature and widely used Java ORM solution, and then implement the rest of the application in Jython. This allows us to make use of the best features of the Python language, but at the same time, persist our data using Java.

Conclusion

You would be hard-pressed to find too many enterprise-level applications today that do not make use of a relational database in one form or another. The majority of applications in use today use databases to store information as they help to provide robust solutions. That being said, the topics covered in this chapter are very important to any developer. In this chapter we learned that there are many different ways to implement database applications in Jython, specifically through the Java database connectivity API or an object relational mapping solution.

Part III: Developing Applications with Jython

Chapter 13: Simple Web Applications

One of the major benefits of using Jython is the ability to make use of Java platform capabilities programming in the Python programming language instead of Java. In the Java world today, the most widely used web development technique is the Java servlet. Now, in JavaEE there are techniques and frameworks used so that we can essentially code HTML or other markup languages as opposed to writing Pure Java servlets. However, sometimes writing a pure Java servlet still has its advantages. We can use Jython to write servlets and this adds many more advantages above and beyond what Java has to offer because now we can make use of the Python language features as well. Similarly, we can code web start applications using Jython instead of pure Java to make our lives easier. Coding these applications in pure Java has proven sometimes to be a difficult and sometimes grueling task. We can use some of the techniques available in Jython to make our lives easier. We can even code WSGI applications with Jython making use of the *modjy* integration in the Jython project.

In this chapter we will cover these techniques for coding simple web applications using Jython servlets, web start, and WSGI. We'll get into details on using each of these different techniques, but we will not discuss deployment of such solutions as that will be covered in Chapter 18: Deployment Targets.

Servlets

Writing servlets in Jython is a very productive and easy way to make use of Jython within a web application. No doubt, Java servlets are rarely written using straight Java anymore. Most Java developers make use of Java Server Pages (JSP), Java Server Faces (JSF), or some other framework so that they can use a markup language to work with web content as opposed to Java code. However, in some cases it is still quite useful to use a pure Java servlet. For these cases we can make our lives easier by using Jython instead. There are also great use-cases for JSP, similarly, we can use Jython for implementing the logic in our JSP code. The latter technique allows us to apply a model-view-controller (MVC) paradigm to our programming model where we separate our front-end markup from any implementation logic. Either technique that you choose to make use of is rather easy to implement, and you can even add this functionality to any existing Java web application without any trouble.

Another feature offered to us by Jython servlet usage is dynamic testing. Since Jython compiles at runtime, we can make code changes on-the-fly without recompiling and redeploying our web application. This can make it very easy

to test web applications as usually the most painful part of web application development is the wait time between deployment to the servlet container and testing.

Configuring Your Web Application for Jython Servlets

Very little needs to be done in any web application to make it compatible for use with Jython servlets. Jython contains a built-in class named *PyServlet* that facilitates the creation of Java servlets using Jython source files. We can make use of *PyServlet* quite easily in our application by adding the necessary XML configuration into the application's *web.xml* descriptor such that the *PyServlet* class gets loaded at runtime and any file that contains the *.py* suffix will be passed to it. Once this configuration has been added to a web application, and *jython.jar* has been added to the CLASSPATH then the web application is ready to use Jython servlets.

```
<servlet>
  <servlet-name>PyServlet</servlet-name>
  <servlet-class>org.python.util.PyServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>PyServlet</servlet-name>
  <url-pattern>*.py</url-pattern>
</servlet-mapping>
```

Any servlet that is going to be used by a Java servlet container also needs to be added to the *web.xml* file as well, this allows for the correct mapping of the servlet via the URL. For the purposes of this book, we will code a servlet named *NewJythonServlet* in the next section, so the following XML configuration will need to be added to the *web.xml* file.

```
<servlet>
  <servlet-name>NewJythonServlet</servlet-name>
  <servlet-class>NewJythonServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>NewJythonServlet</servlet-name>
  <url-pattern>/NewJythonServlet</url-pattern>
</servlet-mapping>
```

Writing a Simple Servlet

In order to write a servlet, we must have the *javax.servlet.http.HttpServlet* abstract Java class within our CLASSPATH so that it can be extended by our Jython servlet to help facilitate the code. This abstract class, along with the other servlet implementation classes, is part of the *servlet-api.jar* file. According to the abstract class, there are two methods that we should override in any Java servlet, those being *doGet* and *doPost*. The former performs the HTTP GET operation while the latter performs the HTTP POST operation for a servlet. Other commonly overridden methods include *doPut*, *doDelete*, and *getServletInfo*. The first performs the HTTP PUT operation, the second performs the HTTP DELETE operation, and the last provides a description for a servlet. In the following example, and in most use-cases, only the *doGet* and *doPost* are used.

Let's first show the code for an extremely simple Java servlet. This servlet contains no functionality other than printing it's name along with it's location in the web application to the screen. Following that code we will take a look at the same servlet coded in Jython for comparison.

NewJavaServlet.java

```
import java.io.IOException;
import java.io.PrintWriter;
```

```

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class NewJavaServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse
↵response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {

            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet NewJavaServlet Test</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet NewJavaServlet at " + request.getContextPath ()
↵+ "</h1>");
            out.println("</body>");
            out.println("</html>");

        } finally {
            out.close();
        }
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    public String getServletInfo() {
        return "Short description";
    }

}

```

All commenting has been removed from the code in an attempt to make the code a bit shorter. Now, the equivalent servlet code written in Jython.

```

from javax.servlet.http import HttpServlet

class NewJythonServlet (HttpServlet):
    def doGet(self, request, response):
        self.doPost (request, response)

    def doPost(self, request, response):

```

```
toClient = response.getWriter()
response.setContentType ("text/html")
toClient.println ("<html><head><title>Jython Servlet Test</title>" +
                  "<body><h1>Servlet Jython Servlet at" +
                  request.getContextPath() + "</h1></body>
↩</html>")

def getServletInfo(self):
    return "Short Description"
```

Not only is the concise code an attractive feature, but also the easy development lifecycle for working with dynamic servlets. As stated previously, there is no need to redeploy each time you make a change because of the compile at runtime that Jython offers. Simply change the Jython servlet, save, and reload the webpage to see the update. If you begin to think about the possibilities you'll realize that the code above is just a basic example, you can do anything in a Jython servlet that you can with Java and even most of what can be done using the Python language as well.

To summarize the use of Jython servlets, you simply include *jython.jar* and *servlet-api.jar* in your CLASSPATH. Add necessary XML to the web.xml, and then finally code the servlet by extending the `javax.servlet.http.HttpServlet` abstract class.

Using JSP with Jython

Harnessing Jython servlets allows for a more productive development lifecycle, but in certain situations Jython code may not be the most convenient way to deal with front-facing web code. Sometimes using a markup language such as HTML works better for developing sophisticated front-ends. For instance, it is easy enough to include javascript code within a Jython servlet. However, all of the javascript code would be written within the context of a String. Not only does this eliminate the usefulness of an IDE for situations such as semantic code coloring and auto completion, but it also makes code harder to read and understand. Cleanly separating such code from Jython or Java makes code more clear to read, and easier to maintain in the long run. Using a JSP allows one to integrate Java code into HTML markup in order to generate dynamic page content. I am not a fan of JSP. There, I have said it, JSP can make code a living nightmare if the technology is not used correctly. Although JSP can make it very easy to mix javascript, HTML, and Java into one file, it can make maintenance very difficult. Mixing Java code with HTML or Javascript is a bad idea. The same would also be true for mixing Jython and HTML or Javascript. JSP is a very smart and productive technology if used correctly, but it can be a coders worst enemy if not done right.

The Model-View-Controller (MVC) paradigm allows for clean separation between logic code such as Java or Jython, and markup code such as HTML. Javascript is a toss-up here, but it always gets grouped into the same arena as HTML because it is a client-side scripting language. In other words, Javascript code should also be separated from the logic code. In thinking about MVC, the controller code would be the markup and javascript code used to capture data from the end-user. Model code would be the business logic that manipulates the data. Model code is contained within our Jython or Java. The view would be the markup and Javascript displaying the result.

Clean separation can using MVC can be achieved successfully by combining JSP with Jython servlets. In this section we will take a look at a simple example of how to do so. As with many of the other examples in this text it will only brush upon the surface of great features that are available. Once you learn how to make use of JSP and Jython servlets you can explore further into the technology.

Configuring for JSP

There is no real configuration above and beyond that of configuring a web application to make use of Jython servlets. Add the necessary XML to the web.xml deployment descriptor, include the correct JARs in your application, and begin coding. What is important to note is that the *.py* files that will be used for the Jython servlets must reside within your CLASSPATH. It is common for the Jython servlets to reside in the same directory as the JSP web pages themselves. This can make things easier, but it can also be frowned upon this concept does not make use of packages

for organizing code. For simplicity sake, we will place the servlet code into the same directory as the JSP, but you can do differently if you prefer.

Coding the Controller/View

The controller and the view portion of the application will be coded using markup and javascript code. Obviously this technique utilizes JSP to contain the markup, and the javascript can either be embedded directly into the JSP or reside in separate .js files as needed. The latter is the preferred method in order to make things clean, but some web applications embed small amounts of Javascript within the pages themselves.

The JSP in this example is rather simple, there is no Javascript in the example and it only contains a couple of input text areas. This JSP will include two forms because we will have two separate submit buttons on the page. Each of these forms will redirect to a different Jython servlet, which will do something with the data that has been supplied within the input text. In our example, the first form contains a small textbox in which the user can type any text that will be re-displayed on the page once the corresponding submit button has been pressed. Very cool, eh? Not really, but it is of good value for learning the correlation between JSP and the servlet implementation. The second form contains two text boxes in which the user will place numbers, hitting the submit button in this form will cause the numbers to be passed to another servlet that will calculate and return the sum of the two numbers. Below is the code for this simple JSP.

```
*testJSP.jsp*

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Jython JSP Test</title>
  </head>
  <body>
    <form method="GET" action="add_to_page.py">
      <input type="text" name="p">
      <input type="submit">
    </form>
    <br/>

    <p>${page_text}</p>

    <br/>

    <form method="GET" action="add_numbers.py">
      <input type="text" name="x">
      +
      <input type="text" name="y">
      =
      ${sum}
    <br/>
    <input type="submit" title="Add Numbers">

    </form>

  </body>
</html>
```

In the JSP above, you can see that the first form redirects to a Jython servlet named *add_to_page.py*. In this case, the text that is contained within the input textbox named *p* will be passed into the servlet, and redisplayed in on the page. The text to be redisplayed will be stored in an attribute named *page_text*, and you can see that it is referenced within the JSP page using the `{ }` notation. Below is the code for *add_to_page.py*.

```
#####
#  add_to_page.py
#
#  Simple servlet that takes some text from a web page and redisplay
#  it.
#####

import java, javax, sys

class add_to_page(javax.servlet.http.HttpServlet):

    def doGet(self, request, response):
        self.doPost(request, response)

    def doPost(self, request, response):
        addtext = request.getParameter("p")
        if not addtext:
            addtext = ""

        request.setAttribute("page_text", addtext)

        dispatcher = request.getRequestDispatcher("testJython.jsp")
        dispatcher.forward(request, response)
```

Quick and simple, the servlet takes the request and obtains value contained within the parameter *p*. It then assigns that value to a variable named *addtext*. This variable is then assigned to an attribute in the request named *page_text* and forwarded back to the *testJython.jsp* page. The code could just as easily have forwarded to a different JSP, which is how we'd go about creating a more in-depth application.

The second form in our JSP takes two values and returns the resulting sum to the page. If someone were to enter text instead of numerical values into the text boxes then an error message would be displayed in place of the sum. While very simplistic, this servlet demonstrates that any business logic can be coded in the servlet, including database calls, etc.

```
#####
#  add_numbers.py
#
#  Calculates the sum for two numbers and returns it.
#####

import javax

class add_numbers(javax.servlet.http.HttpServlet):

    def doGet(self, request, response):
        self.doPost(request, response)

    def doPost(self, request, response):
        x = request.getParameter("x")
        y = request.getParameter("y")
```

```

    if not x or not y:
        sum = "<font color='red'>You must place numbers in each value box</font>"
    else:
        try:
            sum = int(x) + int(y)
        except ValueError, e:
            sum = "<font color='red'>You must place numbers only in each value box
↵</font>"

    request.setAttribute("sum", sum)

    dispatcher = request.getRequestDispatcher("testJython.jsp")
    dispatcher.forward(request, response)

```

If you add the JSP and the servlets to the web application you created in the previous Jython Servlet section then this example should work out-of-the-box.

Applets and Java Web Start

At the time of this writing, applets in Jython 2.5.0 are not yet an available option. This is due to the fact that applets must be statically compiled and available for embedding within a webpage using the `<applet>` or `<object>` tag. The static compiler known as *jythonc* has been removed in Jython 2.5.0 in order to make way for better techniques. Jythonc was good for performing certain tasks, such as static compilation of Jython applets, but it created a disconnect in the development lifecycle as it was a separate compilation step that should not be necessary in order to perform simple tasks such as Jython and Java integration. In a future release of Jython, namely 2.5.1 or another release in the near future, a better way to perform static compilation for applets will be included.

For now, in order to develop Jython applets you will need to use a previous distribution including *jythonc* and then associate them to the webpage with the `<applet>` or `<object>` tag. In Jython, applets are coded in much the same fashion as a standard Java applet. However, the resulting lines of code are significantly smaller in Jython because of its sophisticated syntax. GUI development in general with Jython is a big productivity boost compared to developing a Java Swing application for much the same reason. This is why coding applets in Jython is a viable solution and one that should not be overlooked.

Another option for distributing GUI-based applications on the web is to make use of the Java Web Start technology. The only disadvantage of creating a web start application is that it cannot be embedded directly into any web page. A web start application downloads content to the client's desktop and then runs on the client's local JVM. Development of a Java Web Start application is no different than development of a standalone desktop application. The user interface can be coded using Jython and the Java Swing API, much like the coding for an applet user interface. Once you're ready to deploy a web start application then you need to create a Java Network Launching Protocol (JNLP) file that is used for deployment and bundle it with the application. After that has been done, you need to copy the bundle to a web server and create a web page that can be used to launch the application.

In this section we will develop a small web start application to demonstrate how it can be done using the object factory design pattern and also using pure Jython along with the standalone Jython JAR file for distribution. Note that there are probably other ways to achieve the same result and that these are just a couple of possible implementations for such an application.

Coding a Simple GUI-Based Web Application

The web start application that we will develop in this demonstration is very simple, but they can be as advanced as you'd like in the end. The purpose of this section is not to show you how to develop a web-based GUI application, but rather, the process of developing such an application. You can actually take any of the Swing-based applications that were discussed in the GUI chapter and deploy them using web start technology quite easily. As stated in the previous

section, there are many different ways to deploy a Jython web start application. Personally, I prefer to make use of the object factory design pattern to create simple Jython swing applications. However, it can also be done using all .py files and then distributed using the Jython stand-alone JAR file. We will discuss each of those techniques in this section. Quite often I find that if you are mixing Java and Jython code then the object factory pattern works best. The JAR method may work best for you if developing a strictly Jython application.

Object Factory Application Design

The applicaiton we'll be developing in this section is a simple GUI that takes a line of text and redisplay it in JTextArea. I used Netbeans 6.7 to develop the application, so some of this section may reference particular features that are available in that IDE. To get started with creating an object factory web start application, we first need to create a project. I created a new Java application in Netbeans named *JythonSwingApp* and then added *jython.jar* and *plyjy.jar* to the classpath.

First, create the *Main.java* class which will really be the driver for the application. The goal for *Main.java* is to use the Jython object factory pattern to coerce a Jython-based Swing application into Java. This class will be the starting point for the application and then the Jython code will perform all of the work under the covers. Using this pattern, we also need a Java interface that can be implemented via the Jython code, so this example also uses a very simple interface that defines a *start()* method which will be used to make our GUI visible. Lastly, the Jython class named Below is the code for our *Main.java* driver and the Java interface. In our example, we named this class *MainOF.java* to differentiate it from the next example that uses *PythonInterpreter*. The directory structure of this application is as follows.

JythonSwingApp

JythonSimpleSwing.py

jythonswingapp MainOF.java

jythonswingapp.interfaces JySwingType.java

```
*MainOF.java*

package jythonswingapp;

import jythonswingapp.interfaces.JySwingType;
import org.plyjy.factory.JythonObjectFactory;

public class Main {

    JythonObjectFactory factory;

    public static void invokeJython(){

        JySwingType jySwing = (JySwingType) JythonObjectFactory
            .createObject(JySwingType.class, "JythonSimpleSwing");
        jySwing.start();
    }

    public static void main(String[] args) {
        invokeJython();
    }

}
```

As you can see, *MainOF.java* doesn't do much other than coercing the Jython module and invoking the *start()* method. Next, you will see the *JySwingType.java* interface along with the implementation class that is obviously coded in Jython.


```

*JySwingType.java*
package jythonswingapp.interfaces;

public interface JySwingType {
    public void start();
}

*JythonSimpleSwing.py*
import javax.swing as swing
import java.awt as awt
from jythonswingapp.interfaces import JySwingType
import add_player as add_player
import Player as Player

class JythonSimpleSwing(JySwingType, object):
    def __init__(self):
        self.frame=swing.JFrame(title="My Frame", size=(300,300))
        self.frame.defaultCloseOperation=swing.JFrame.EXIT_ON_CLOSE;
        self.frame.layout=awt.BorderLayout()
        self.panel1=swing.JPanel(awt.BorderLayout())
        self.panel2=swing.JPanel(awt.GridLayout(4,1))
        self.panel2.preferredSize = awt.Dimension(10,100)
        self.panel3=swing.JPanel(awt.BorderLayout())

        self.title=swing.JLabel("Text Rendering")
        self.button1=swing.JButton("Print Text", actionPerformed=self.printMessage)
        self.button2=swing.JButton("Clear Text", actionPerformed=self.clearMessage)
        self.textField=swing.JTextField(30)
        self.outputText=swing.JTextArea(4,15)

        self.panel1.add(self.title)
        self.panel2.add(self.textField)
        self.panel2.add(self.button1)
        self.panel2.add(self.button2)
        self.panel3.add(self.outputText)

        self.frame.contentPane.add(self.panel1, awt.BorderLayout.PAGE_START)
        self.frame.contentPane.add(self.panel2, awt.BorderLayout.CENTER)
        self.frame.contentPane.add(self.panel3, awt.BorderLayout.PAGE_END)

    def start(self):
        self.frame.visible=1

    def printMessage(self, event):
        print "Print Text!"
        self.text = self.textField.getText()
        self.outputText.append(self.text)

    def clearMessage(self, event):
        self.outputText.text = ""

```

If you are using Netbeans then when you clean and build your project a JAR file is automatically generated for you. However, you can easily create a JAR file at the command-line or terminal by ensuring that the *JythonSimpleSwing.py* module resides within your classpath and using the *java -jar* option. Another nice feature of using an IDE such as Netbeans is that you can make this into a web-start application by going into the project properties and checking a couple of boxes. Specifically, if you go into the project properties and select *Application - Web Start* from the left-hand

menu, then check the *Enable Web Start* option then the IDE will take care of generating the necessary files to make this happen. Netbeans also has the option to self sign the JAR file which is required to run most applications on another machine via web start. Go ahead and try it out, just ensure that you clean and build your project again after making the changes.

To manually create the necessary files for a web start application, you'll need to generate two additional files that will be placed outside of the application JAR. Create the JAR for your project as you would normally do, and then create a corresponding JNLP file which is used to launch the application, and an HTML page that will reference the JNLP. The HTML page obviously is where you'd open the application if running it from the web. Below is some example code for generating a JNLP as well as embedding in HTML.

launch.jnlp

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<jnlp codebase="file:/path-to-jar/" href="launch.jnlp" spec="1.0+">
```

```
  <information> <title>JythonSwingApp</title>  <vendor>YourName</vendor>  <home-
    page href=""/>  <description>JythonSwingApp</description>  <description
    kind="short">JythonSwingApp</description>
```

```
  </information>
```

```
<security> <all-permissions/> </security>
```

```
  <resources>
```

```
    <j2se version="1.5+"/> <jar eager="true" href="JythonSwingApp.jar" main="true"/>
```

```
    <jar href="lib/PlyJy.jar"/>
```

```
  <jar href="lib/jython.jar"/> </resources>
```

```
  <application-desc main-class="jythonswingapp.Main"> </application-desc>
```

```
</jnlp>
```

launch.html

```
<html>
```

```
  <head> <title>Test page for launching the application via JNLP</title>
```

```
  </head> <body>
```

```
    <h3>Test page for launching the application via JNLP</h3>  <a
    href="launch.jnlp">Launch the application</a>  <!-- Or use the following
    script element to launch with the Deployment Toolkit ->  <!-- Open the
    deployJava.js script to view its documentation ->  <!--  <script src="http:
    //java.com/js/deployJava.js"></script> <script>
```

```
      var url="http://{fill in your URL}/launch.jnlp"  deploy-
      Java.createWebStartLaunchButton(url, "1.6")
```

```
    </script> ->
```

```
  </body>
```

```
</html>
```

In the end, Java web start is a very good way to distribute Jython applications via the web.

PythonInterpreter Application Design

It is also possible to invoke a Jython script using a Jython specific class known as the `PythonInterpreter`. This class essentially allows embedding of Python code within Java. Once instantiated, the `PythonInterpreter` takes a line of Python code as a string and executes it. This technique was first introduced in chapter 10 of this text, which covered Jython and Java integration. Using the `PythonInterpreter`, one can invoke Jython scripts directly within Java application code. This technique can also be used to invoke a Jython GUI application using a Java *Main* class. This technique is especially useful when packaging code into a JAR file for distribution.

Let's take a look at the example from above, this time using `PythonInterpreter` as opposed to object factories for invoking our Jython code. The only piece of code shown here will be *Main.java* as it is the only piece of code that needs to change when using this technique. Of course, the Java interface is no longer needed so it may be removed from the application as well.

Main.java

```
package jythonswingapp;

import org.python.core.PyException;
import org.python.util.PythonInterpreter;

public class Main {
    public static void main(String[] args) throws PyException{
        PythonInterpreter intrp = new PythonInterpreter();
        intrp.exec("import JythonSimpleSwing as jy");
        intrp.exec("jy.JythonSimpleSwing().start()");
    }
}
```

The advantage of using this technique is that we can do away with the object factory design. However, the disadvantages are that embedding script within Java code is not the most clean technique. This leads to code that is difficult to maintain and troubleshoot if issues are found. The other disadvantage is that this technique forces the Java developer to know how to instantiate the Jython module and use it in any way. The object factory design helps to enforce abstraction and simplification as the Java developer only needs to be aware of those methods defined within the Java interface. In most scenarios however, there is only one developer for the application so this such nuances to not matter.

Distributing via Standalone JAR

It is possible to distribute a web start application using the Jython standalone JAR option. To do so, you must have a copy of the Jython standalone JAR file, explode it and add your code into the file, then JAR it back up to deploy. The only setback to using this method is that you may need to play around a bit with your file placement in order to make it work correctly.

In order to distribute your Jython applications via a JAR, first download the Jython standalone distribution. Once you have this, you can extract the files from the *jython.jar* using a tool to expand the JAR such as “Stuffit” or “7zip”. Once the JAR has been exploded, you will need to add any of your *.py* scripts into the *Lib* directory, and any Java classes into the root. For instance, if you have a Java class named *org.jythonbook.Book*, you would place it into the appropriate directory according to the package structure. If you have any additional JAR files to include with your application then you will need to make sure that they are in your classpath. Once you've completed this setup, JAR your manipulated standalone Jython JAR back up into a ZIP format using a tool such as those noted before. You can then rename the ZIP to a JAR. The application can now be run using the java “-jar” option from the command line using an optional external *.py* file to invoke your application.

```
java -jar newStandaloneJar.jar {optional .py file}
```

This is only one such technique used to make a JAR file for containing your applications. There are other ways to perform such techniques, but this seems to be the most straight forward and easiest to do.

WSGI and modjy

WSGI, also known as the *Web Server Gateway Interface*, is a low-level API that provides communication between a web server and a web application. Actually, WSGI is a lot more than that and you can actually write complete web applications using WSGI. However, WSGI is more of a standard from which to write web frameworks. Python PEP 333 specifies the proposed standard interface between web servers and Python web applications or frameworks, to promote web application portability across a variety of web servers.

The basis behind WSGI is beyond the scope of this section. As a matter of fact, complete books could be written on the subject. This section will show you how to utilize WSGI to create a very simple “Hello Jython” application by utilizing *modjy*. Modjy is an implementation of a WSGI compliant gateway/server for jython, built on Java/J2EE servlets. Taken from the modjy website (<http://opensource.xhaus.com/projects/modjy/wiki>), modjy is characterized as follows:

Jython WSGI applications run inside a Java/J2EE container and incoming requests are handled by the servlet container. The container is configured to route requests to the modjy servlet. The modjy servlet then creates an embedded jython interpreter inside the servlet container, and loads a configured jython web application. For instance, a Django application can be loaded via modjy. The modjy servlet then delegates the requests to the configured WSGI application or framework. Lastly, the WSGI response is routed back to the client through the servlet container.

Running a modjy Application in Glassfish

To run a modjy application in any Java servlet container, the first step is to create a Java web application that will be packaged up as a WAR file. You can create an application from scratch or use an IDE such as Netbeans 6.7 to assist. Once you’ve created your web application, ensure that *jython.jar* resides in the CLASSPATH as modjy is now part of Jython as of 2.5.0. Lastly, you will need to configure the modjy servlet within the application deployment descriptor (*web.xml*). In this example, I took the modjy sample application for Google App Engine and deployed it in my local Glassfish environment.

To configure the application deployment descriptor with modjy, we simply configure the modjy servlet, provide the necessary parameters, and then provide a servlet mapping. In the configuration file shown below, note that the modjy servlet class is *com.xhaus.modjy.ModjyServlet*. The first parameter you will need to use with the servlet is named *python.home*. Set the value of this parameter equal to your jython home. Next, set the parameter *python.cachedir.skip* equal to true. The *app_filename* parameter provides the name of the applicaiton callable. Other parameters will be set up the same for each modjy application you configure. The last piece of the *web.xml* that needs to be set up is the servlet mapping. In the example, we set up all URLs to map to the modjy servlet.

```
*web.xml*
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

    <display-name>modjy demo application</display-name>
    <description>
        modjy WSGI demo application
    </description>

    <servlet>
        <servlet-name>modjy</servlet-name>
        <servlet-class>com.xhaus.modjy.ModjyJServlet</servlet-class>
```

```

<init-param>
  <param-name>python.home</param-name>
  <param-value>/Applications/jython/jython2.5.0</param-value>
</init-param>
<init-param>
  <param-name>python.cachedir.skip</param-name>
  <param-value>true</param-value>
</init-param>
<!--
  There are two different ways you can specify an application to modjy
  1. Using the app_import_name mechanism
  2. Using a combination of app_directory/app_filename/app_callable_name
  Examples of both are given below
  See the documenation for more details.
  http://modjy.xhaus.com/locating.html#locating_callables
-->
<!--
  This is the app_import_name mechanism. If you specify a value
  for this variable, then it will take precedence over the other mechanism
<init-param>
  <param-name>app_import_name</param-name>
  <param-value>my_wsgi_module.my_handler_class().handler_method</param-value>
</init-param>
-->
<!--
  And this is the app_directory/app_filename/app_callable_name combo
  The defaults for these three variables are "/application.py/handler
  So if you specify no values at all for any of app_* variables, then modjy
  will by default look for "handler" in "application.py" in the servlet
  context root.
<init-param>
  <param-name>app_directory</param-name>
  <param-value>some_sub_directory</param-value>
</init-param>
-->
<init-param>
  <param-name>app_filename</param-name>
  <param-value>demo_app.py</param-value>
</init-param>
<!--
  Supply a value for this parameter if you want your application
  callable to have a different name than the default.
<init-param>
  <param-name>app_callable_name</param-name>
  <param-value>my_handler_func</param-value>
</init-param>
-->
  <!-- Do you want application callables to be cached? -->
<init-param>
  <param-name>cache_callables</param-name>
  <param-value>1</param-value>
</init-param>
<!-- Should the application be reloaded if it's .py file changes? -->
<!-- Does not work with the app_import_name mechanism -->
<init-param>
  <param-name>reload_on_mod</param-name>
  <param-value>1</param-value>
</init-param>

```

```

    <init-param>
      <param-name>log_level</param-name>
      <param-value>debug</param-value>
    <!-- <param-value>info</param-value> -->
    <!-- <param-value>warn</param-value> -->
    <!-- <param-value>error</param-value> -->
    <!-- <param-value>fatal</param-value> -->
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>modjy</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>

```

The `demo_app` should be coded as follows. As part of the WSGI standard, the application provides a function that the server calls for each request. In this case, that function is named *handler*. The function must take two parameters, the first being a dictionary of CGI-defined environment variables. The second is a callback that returns the HTTP headers. The callback function should also be coded as follows *start_response(status, response_headers, exc_info=None)*, where *status* is an HTTP status, *response_headers* is a list of HTTP headers, and *exc_info* is for exception handling. Let's take a look at the *demo_app.py* application and identify the features we've just discussed.

```

import sys

def escape_html(s): return s.replace('&', '&amp;').replace('<', '&lt;').replace('>',
↳ '&gt;')

def cutoff(s, n=100):
    if len(s) > n: return s[:n]+ '.. cut ..'
    return s

def handler(environ, start_response):
    writer = start_response("200 OK", [ ('content-type', 'text/html') ])
    response_parts = []
    response_parts.append("<html>")
    response_parts.append("<head>")
    response_parts.append("<title>Modjy demo WSGI application running on Local Server!
↳ </title>")
    response_parts.append("</head>")
    response_parts.append("<body>")
    response_parts.append("<p>Modjy servlet running correctly: jython %s on %s:</p>"
↳ % (sys.version, sys.platform))
    response_parts.append("<h3>Hello jython WSGI on your local server!</h3>")
    response_parts.append("<h4>Here are the contents of the WSGI environment</h4>")
    environ_str = "<table border='1'"
    keys = environ.keys()
    keys.sort()
    for ix, name in enumerate(keys):
        if ix % 2:
            background='#ffffff'
        else:
            background='#eeeeee'
        style = " style='background-color:%s;" % background
        value = escape_html(cutoff(str(environ[name]))) or '&#160;'

```

```

        environ_str = "%s\n<tr><td%s>%s</td><td%s>%s</td></tr>" % \
            (environ_str, style, name, style, value)
    environ_str = "%s\n</table>" % environ_str
    response_parts.append(environ_str)
    response_parts.append("</body>")
    response_parts.append("</html>")
    response_text = "\n".join(response_parts)
    return [response_text]

```

This application returns the environment configuration for the server on which you run the application. As you can see, the page is quite simple to code and really resembles a servlet.

Once the application has been set up and configured, simply compile the code into a WAR file and deploy it to the Java servlet container of your choice. In this case, I used Glassfish V2 and it worked nicely. However, this same application should be deployable to Tomcat, JBoss, or the like.

Conclusion

There are various ways that we can use Jython for creating simple web-based applications. Jython servlets are a good way to make content available on the web, and you can also utilize them along with a JSP page which allows for a Model-View-Controller situation. This is a good technique to use for developing sophisticated web applications, especially those mixing some Javascript into the action because it really helps to organize things. Most Java web applications use frameworks or other techniques in order to help organize applications in such a way as to apply the MVC concept. It is great to have a way to do such work with Jython as well.

This chapter also discussed creation of WSGI applications in Jython making use of modjy. This is a good low-level way to generate web applications as well, although modjy and WSGI are usually used for implementing web frameworks and the like. Solutions such as Django use WSGI in order to follow the standard put forth for all Python web frameworks with PEP 333. You can see from the section in this chapter that WSGI is also a nice quick way to write web applications, much like writing a servlet in Jython.

In the upcoming chapter you will learn more about using web frameworks available to Jython, specifically Django and Pylons. These two frameworks can make any web developers life much easier, and now that they are available on the Java platform via Jython they are even more powerful. Using a templating technique such as Django can be really productive and it is a good way to design a full-blown web application. Techniques discussed in this chapter can also be used for developing large web applications, but using a standard framework such as those discussed in the following chapter should not be overlooked. There are many great ways to code Jython web applications today, and the options continue to grow!

Chapter 14: Web Applications with Django

Django is one of the modern Python web frameworks which redefined the web niche in the Python world. A full stack approach, pragmatic design and superb documentation are some of the reason for its success.

If fast web development using the Python language sounds good to you, then fast web development using the Python language and with integration with the whole Java world (which has a strong presence on the enterprise web space) sounds even better. Running Django on Jython allows you to do just that!

And for the Java world, having Django as an option to quickly build web applications while still having the chance to use the existing Java APIs and technologies is very attractive.

In this chapter we will start with a quick introduction to have Django running with your Jython installation in a few steps and then we will build a simple web application to get a feeling of the framework. Later on, in the second half of the chapter we will take a look at the many opportunities of integration between Django web applications and the JavaEE world.

Getting Django

Strictly, to use Django with Jython you only need to get Django itself, and nothing more. But, without third-party libraries, you won't be able to connect to any database, since the built-in Django database backends depend on libraries written in C, which aren't available on Jython.

In practice, you will need at least two packages: Django itself and “django-jython”, which, as you can imagine, is a collection of Django addons that can be quite useful if you happen to be running Django on top of Jython. In particular it includes database backends, which is something you definitely need to fully appreciate the power of Django.

Since the process of getting these two libraries slightly varies depending on your platform, and it's a manual, boring task, we will use an utility to automatically grab and install these libraries. The utility is called “setuptools”. Refer to the appendix A for instructions on how to install setuptools on Jython.

After installing setuptools the `easy_install` command will be available. Armed with this we proceed to install Django:

```
$ easy_install Django==1.0.3
```

Note: I'm assuming that the `bin` directory of the Jython installation is on your `PATH`. If it's not, you will have to explicitly type that path preceding each command like `jython` or `easy_install` with that path (i.e., you will need to type something like `/path/to/jython/bin/easy_install` instead of just `easy_install`)

By reading the output of `easy_install` you can see how it is doing all the tedious work of locating the right package, downloading and installing it:

```
Searching for Django==1.0.3
Reading http://pypi.python.org/simple/Django/
Reading http://www.djangoproject.com/
Reading http://www.djangoproject.com/download/1.0.1-beta-1/tarball/
Best match: Django 1.0.3
Downloading http://media.djangoproject.com/releases/1.0.3/Django-1.0.3.tar.gz
Processing Django-1.0.3.tar.gz
Running Django-1.0.3/setup.py -q bdist_egg --dist-dir
/tmp/easy_install-nTnmlU/Dj  ango-1.0.3/egg-dist-tmp-L-pq4s
zip_safe flag not set; analyzing archive contents...
Unable to analyze compiled code on this platform.
Please ask the author to include a 'zip_safe' setting (either True or False)
in the package's setup.py
Adding Django 1.0.3 to easy-install.pth file
Installing django-admin.py script to /home/lsoto/jython2.5.0/bin

Installed /home/lsoto/jython2.5.0/Lib/site-packages/Django-1.0.3-py2.5.egg
Processing dependencies for Django==1.0.3
Finished processing dependencies for Django==1.0.3
```

Then we install `django-jython`:

```
$ easy_install django-jython
```

Again, you will get an output similar to what you've seen in the previous cases. Once this is finished, you are ready.

If you want to look behind the scenes, take a look at the `Lib/site-packages` subdirectory inside your Jython installation and you will find entries for the libraries we just installed. Those entries are also listed on the `easy-install.pth` file, making them part of `sys.path` by default.

Just to make sure that everything went fine, start jython and try the following statements, which import the top-level packages of Django and django-jython:

```
>>> import django
>>> import doj
```

If you don't get any error printed out on the screen, then everything is OK. Let's start our first application.

A Quick Tour of Django

Note: If you are already familiar with Django, you won't find anything specially new in the rest of this section. Feel free to jump to *J2EE deployment and integration* to look at what's really special if you run Django on Jython.

Django is a full-stack framework. That means that it features cover from communication to the database, to URL processing and web page templating. As you may know, there are complete books which cover Django in detail. We aren't going to go into much detail, but we *are* going to touch many of the features included in the framework, so you can get a good feeling of its strengths in case you haven't had the chance to know or try Django in the past. That way you will know when Django is the right tool for a job.

The only way to take a broad view of such a resourceful framework like Django is to build something really simple with it, and then gradually augment it as we look into what the framework offers. So, we will start following roughly what the official Django tutorial uses (a simple site for polls) to extend it later to touch most of the framework features. In other words: most of the code you will see in this section comes directly from the great Django tutorial you can find on <http://docs.djangoproject.com/en/1.0/intro/tutorial01/>.

Now, as I said on the previous paragraph, Django handles the communication with the database. Right now, the more solid backend in existence for Django/Jython is the one for PostgreSQL. So I encourage you to install PostgreSQL on your machine and setup an user and an empty database to use it in the course of this tour.

Starting a Project (and an “App”)

Django projects, which are usually meant to be complete web sites (or “sub-sites”) are composed of a settings file, a URL mappings file and a set of “apps” which provide the actual features of the web site. As you surely have realized, many web sites share a lot of features: administration interfaces, user authentication/registration, commenting systems, news feeds, contact forms, etc. That's why Django decouples the actual site features in the “app” concept: apps are meant to be *reusable* between different projects (sites).

As we will start small, our project will consist of only one app at first. We will call our project “pollsite”. So, let's create a clean new directory for what we will build on this sections, move to that directory and run:

```
$ django-admin.py startproject pollsite
```

And a python package named “pollsite” will be created under the directory you created previously. At this point, the most important change we *need* to make to the default settings of our shiny new project is to fill the information so Django can talk to the database we created for this tour. So, open the file `pollsite/settings.py` with your text editor of choice and change lines starting with `DATABASE` with something like this:

```
DATABASE_ENGINE = 'doj.backends.zxjdbc.postgresql'
DATABASE_NAME = '<the name of the empty database you created>'
DATABASE_USER = '<the name of the user with R/W access to that database>'
DATABASE_PASSWORD = '<the password of such user>'
```

With this, you are telling Django to use the PostgreSQL driver provided by the `doj` package (which, if you remember from the *Getting Django* section, was the package name of the `django-jython` project) and to connect

with the given credentials. Now, this backend requires the PostgreSQL JDBC driver, which you can download at <http://jdbc.postgresql.org/download.html>.

Once you download the JDBC driver, you need to add it to the Java CLASSPATH. An way to do it in Linux/Unix/MacOSX for the current session is:

```
$ export CLASSPATH=$CLASSPATH:/path/to/postgresql-jdbc.jar
```

If you are on Windows, the command is different:

```
$ set CLASSPATH=%CLASSPATH%;\path\to\postgresql-jdbc.jar
```

Done that, we will create the single app which will be the core of our project. Make sure you are into the `pollsite` directory and run:

```
$ jython manage.py startapp polls
```

This will create the basic structure of a Django app. Note that the app was created inside the project package, so we have the `pollsite` project and the `pollsite.polls` app.

Now we will see what's inside a Django app.

Models

In Django, you define your data schema in Python code, using Python classes. This central schema is used to generate the needed SQL statements to create the database schema, and to dynamically generate SQL queries when you manipulate objects of these special Python classes.

Now, in Django you don't define the schema of the whole project in a single central place. After all, since apps are the real providers of features, it follows that the schema of the whole project isn't more that the combination of the schemas of each app. By the way, we will switch to Django terminology now, and instead of talking about data schemas, we will talk about models (which are actually a bit more than just schemas, but the distinction is not important at this point).

If you look into the `pollsite/polls` directory, you will see that there is a `models.py` file, which is where the app's models must be defined. The following code contains the model for simple polls, each poll containing many choices:

```
from django.db import models

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    def __unicode__(self):
        return self.question

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice = models.CharField(max_length=200)
    votes = models.IntegerField()

    def __unicode__(self):
        return self.choice
```

As you can see, the map between a class inheriting from `models.Model` and a database table is clear, and its more or less obvious how each Django field would be translated to a SQL field. Actually, Django fields can carry more information than SQL fields can, as you can see on the `pub_date` field which includes a description more suited

for human consumption: “date published”. Django also provides more specialized field for rather common types seen on today web applications, like `EmailField`, `URLField` or `FileField`. They free you from having to write the same code again and again to deal with concerns such as validation or storage management for the data these fields will contain.

Once the models are defined, we want to create the tables which will hold the data on the database. First, you will need to add app to the project settings file (yes, the fact that the app “lives” under the project package isn’t enough). Edit the file `pollsite/settings.py` and add `'pollsite.polls'` to the `INSTALLED_APPS` list. It will look like this:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'pollsite.polls',
)
```

Note: As you see, there were a couple of apps already included in your project. These apps are included on every Django project by default, providing some of the basic features of the framework, like sessions.

After that, we make sure we are located on the project directory and run:

```
$ jython manage.py syncdb
```

If the database connection information was correctly specified, Django will create tables and indexes for our models *and* for the models of the other apps which were also included by default on `INSTALLED_APPS`. One of these extra apps is `django.contrib.auth`, which handle user authentication. That’s why you will also be asked for the username and password for the initial admin user for your site:

```
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table polls_poll
Creating table polls_choice

You just installed Django's auth system, which means you don't have any
superusers defined.
Would you like to create one now? (yes/no):
```

Answer yes to that question, and provide the requested information:

```
Username (Leave blank to use u'lsoto'): admin
E-mail address: admin@mailinator.com
Warning: Problem with getpass. Passwords may be echoed.
Password: admin
Warning: Problem with getpass. Passwords may be echoed.
Password (again): admin
Superuser created successfully.
```

After this, Django will continue mapping your models to RDBMS artifacts, creating some indexes for your tables:

```
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for polls.Choice model
```

If we want to know what's doing Django behind the scenes, we can ask that, using the `sqlall` management command (which are how the commands recognized by `manage.py` are called, like the recently used `syncdb`). This command requires an app *label* as argument and prints the SQL statements corresponding to the models contained in the app. By the way, the emphasis on *label* was intentional, as it corresponding to the last part of the “full name” of an app and not to the full name itself. In our case, the label of “pollsite.polls” is simply “polls”. So, we can run:

```
$ jython manage.py sqlall polls
```

And we get the following output:

```
BEGIN;
CREATE TABLE "polls_poll" (
  "id" serial NOT NULL PRIMARY KEY,
  "question" varchar(200) NOT NULL,
  "pub_date" timestamp with time zone NOT NULL
)
;
CREATE TABLE "polls_choice" (
  "id" serial NOT NULL PRIMARY KEY,
  "poll_id" integer NOT NULL
    REFERENCES "polls_poll" ("id") DEFERRABLE INITIALLY DEFERRED,
  "choice" varchar(200) NOT NULL,
  "votes" integer NOT NULL
)
;
CREATE INDEX "polls_choice_poll_id" ON "polls_choice" ("poll_id");
COMMIT;
```

Two things to note here. First, each table got an `id` field which wasn't explicitly specified on our model definition. That's automatic, and is a sensible default (which can be overridden if you really need a different type of primary key, but that's outside the scope of this quick tour). Second, see how the sql is tailored to the particular RDBMS we are using (PostgreSQL in this case), so naturally it may change if you use a different database backend.

OK, Let's move on. We have our model defined, and ready to store polls. The typical next step here would be to make a CRUD administrative interface so polls can be created, edited, removed, etc. Oh, and of course we may envision some searching and filtering capabilities for this administrative, knowing in advance that once the amount of polls grow too much it will become really hard to manage.

Well, no. We won't write this administrative interface from scratch. We will use one of the most useful features of Django: The admin app.

Bonus: The Admin

This is an intermission on our tour through the main architectural points of a Django project (namely: models, views and templates) but is a very nice intermission. The code for the administrative interface we talked about a couple of paragraph back will consist on less than 20 lines of code!

First, let's enable the admin app. To do this, edit `pollsite/settings.py` and add `'django.contrib.admin'` to the `INSTALLED_APPS`. Then edit `pollsite/urls.py` which looks like this:

```
from django.conf.urls.defaults import *

# Uncomment the next two lines to enable the admin:
```

```
# from django.contrib import admin
# admin.autodiscover()

urlpatterns = patterns('',
    # Example:
    # (r'^pollsite/', include('pollsite.foo.urls')),

    # Uncomment the admin/doc line below and add 'django.contrib.admindocs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # (r'^admin/(.*)', admin.site.root),
)
```

And uncomment the lines which enable the admin (but not the admin/doc!), so the file will look this way:

```
from django.conf.urls.defaults import *

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Example:
    # (r'^pollsite/', include('pollsite.foo.urls')),

    # Uncomment the admin/doc line below and add 'django.contrib.admindocs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    (r'^admin/(.*)', admin.site.root),
)
```

Now you can remove all the remaining commented lines, so `urls.py` ends up with the following contents:

```
from django.conf.urls.defaults import *

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^admin/(.*)', admin.site.root),
)
```

I know I haven't explained this `urls.py` file yet, but trust me, we will see it in the next section.

Finally, let's create the database artifacts needed by the admin app, running:

```
$ jython manage.py syncdb
```

Now we will see how this admin looks like. Let's run our site in development mode by executing:

```
$ jython manage.py runserver
```

Note: The development web server is an easy way to test your web project. It will run indefinitely until you abort it

(for example, hitting `Ctrl + C`) and will reload itself when you change a source file already loaded by the server, thus giving almost instant feedback. But, be advised that using this development server in production is a really, really bad idea.

Using a web browser, navigate to <http://localhost:8000/admin/>. You will be presented with a login screen. Use the user credential you made when we first ran `syncdb` in the previous section. Once you log in, you will see a page like the one shown in the figure *The Django Admin*.

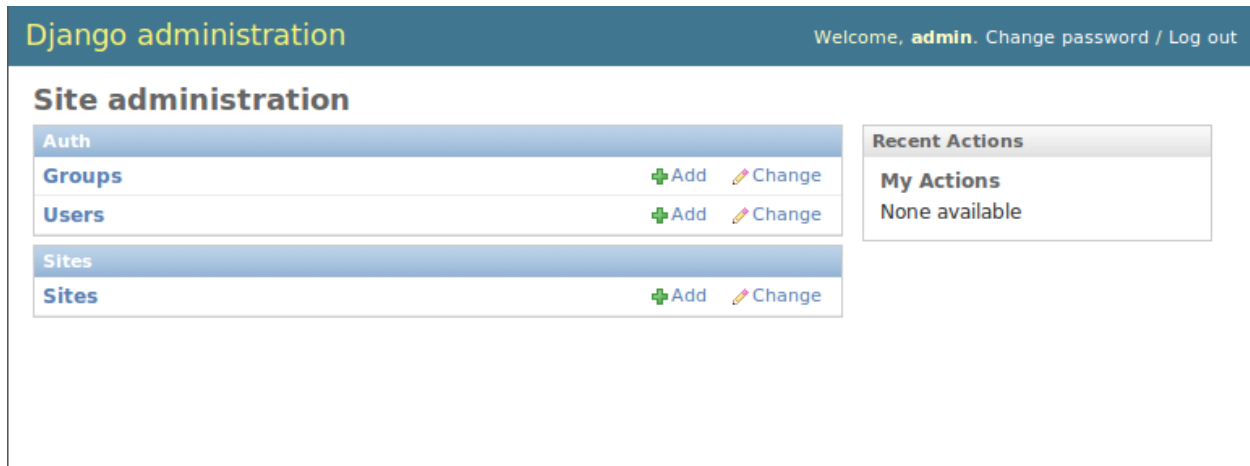


Fig. 4.1: The Django Admin

As you can see, the central area of the admin shows two boxes, titled “Auth” and “Sites”. Those boxes correspond to the “auth” and “sites” apps that are built in on Django. The “Auth” box contain two entries: “Groups” and “Users”, each one corresponding to a model contained in the auth app. If you click the “Users” link you will be presented with the typical options to add, modify and remove users. This is the kind of interfaces that the admin can provide to any other Django app, so we will add our polls app to it.

Doing so is a matter of creating an `admin.py` file under your app (that is, `pollsite/polls/admin.py`) and declaratively telling the admin how you want to present your models in the admin. To administer polls, the following will make the trick:

```
# polls admin.py
from pollsite.polls.models import Poll, Choice
from django.contrib import admin

class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'],
                                     'classes': ['collapse']}),
    ]
    inlines = [ChoiceInline]

admin.site.register(Poll, PollAdmin)
```

This may read like magic to you, but remember that I’m moving quick, as I want you to take a look at what’s possible to do with Django. Let’s look first at what we get after writing this code. Start the development server, go to <http://localhost:8000/>

`//localhost:8000/admin/` and see how a new “Polls” box appears now. If you click the “Add” link in the “Polls” entry, you will see a page like the one on the figure *Adding a Poll using the Admin*.

Django administration Welcome, **admin**. [Change password](#) / [Log out](#)

[Home](#) > [Polls](#) > [Polls](#) > [Add poll](#)

Add poll

Question:

Date information (Hide)

Date published: **Date:** [Today](#) | [Calendar](#)
Time: [Now](#) | [Clock](#)

Choices

Choice: #1

Choice:

Votes:

Choice: #2

Choice:

Votes:

Choice: #3

Choice:

Votes:

[Save and add another](#) [Save and continue editing](#) [Save](#)

Fig. 4.2: Adding a Poll using the Admin

Play a bit with the interface: create a couple of polls, remove one, modify them. Note that the user interface is divided in three parts, one for the question, another for the date (initially hidden) and other for the choices. The first two were defined by the `fieldsets` of the `PollAdmin` class, which let you define the titles of each section (where `None` means no title), the fields contained (they can be more than one, of course) and additional CSS classes providing behaviors like `'collapse'`

It’s fairly obvious that we have “merged” the administration of our two models (`Poll` and `Choice`) into the same user interface, since choices ought to be edited “inline” with their corresponding poll. That was done via the `ChoiceInline` class which declares what model will be inlined and how many empty slots will be shown. The inline is hooked up into the `PollAdmin` later (since you can include many inlines on any `ModelAdmin` class).

Finally, `PollAdmin` is registered as the administrative interface for the `Poll` model using `admin.site.register()`. As you can see, everything is absolutely declarative and works like a charm.

The attentive reader is probably wondering what about the search/filter features I talked about a few paragraphs back. Well, we will implement that, in the poll list interface which you can access when clicking the “Change” link for Polls in the main interface (or also by clicking the link “Polls”, or after adding a Poll).

So, add the following lines to the `PollAdmin` class:

```
search_fields = ['question']
list_filter = ['pub_date']
```

And play with the admin again (that’s why it was a good idea to create a few polls in the last step). The figure *Searching on the Django Admin* shows the search working, using “django” as the search string.

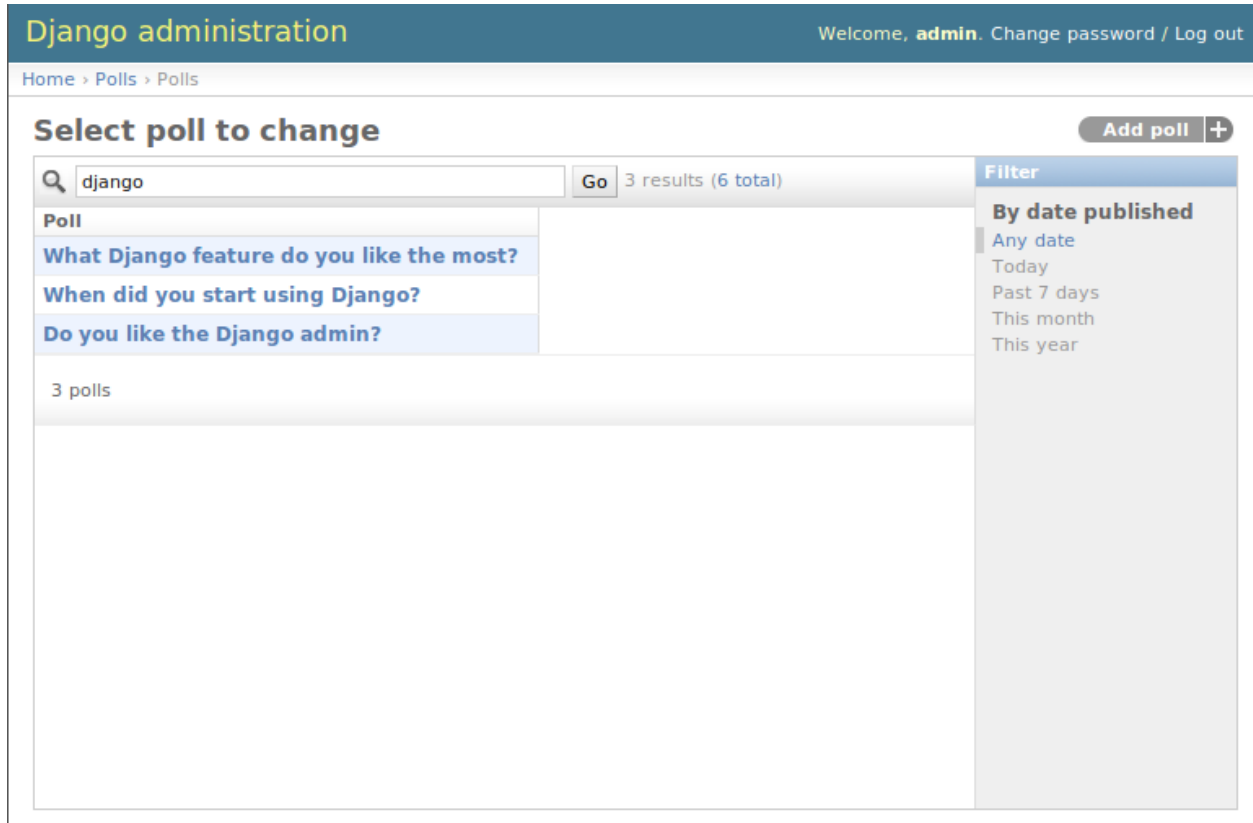


Fig. 4.3: Searching on the Django Admin

Now, if you try the filter by publishing date, it feels a bit awkward because the list of polls only shows the name of the poll, so you can’t see what’s the publishing date of the polls being filtered, to check if the filter worked as advertised. That’s easy to fix, by adding the following line to the `PollAdmin` class:

```
list_display = ['question', 'pub_date']
```

The figure *Filtering and listing more fields on the Django Admin* shows how the interface looks after all these additions.

Once again you can see how admin offers you all these commons features almost for free, and you only have to say what you want in a purely declarative way. However, in case you have more special needs, the admin has hooks which you can use to customize its behavior. It is so powerful that sometimes it happens that a whole web application can be built based purely on the admin. See the official docs <http://docs.djangoproject.com/en/1.0/ref/contrib/admin/> for more information.



Fig. 4.4: Filtering and listing more fields on the Django Admin

Views and Templates

Well, now that you know the admin I won't be able to use a CRUD to showcase the rest of the main architecture of the web framework. That's OK: CRUDs are part of almost all data driven web applications, but they aren't what make your site different. So, now that we have delegated the tedium to the admin app, we will concentrate on polls, which is our business.

We already have our models in place, so it's time to write our views, which are the HTTP-oriented functions that will make our app talk with the outside (which is, after all, the point of creating a *web* application).

Note: Django developers half-jokingly say that Django follows the “MTV” pattern: Model, Template and View. These 3 components map directly to what other modern frameworks call Model, View and Controller. Django takes this apparently unorthodox naming schema because, strictly, the controller is the framework itself. What is called “controller” code in other frameworks is really tied to HTTP and output templates, so it's really part of the view layer. If you don't like this viewpoint, just remember to mentally map Django templates to “views” and Django views to “controllers”.

By convention, code for views go into the app `views.py` file. Views are simple functions which take an HTTP request, do some processing and return an HTTP response. Since an HTTP response typically involves the construction of an HTML page, templates aid views with the job of creating HTML output (and other text-based outputs) in a more maintainable way than manually pasting strings together.

The polls app will have a very simple navigation. First, the user will be presented with an “index” with access to the list of the latest polls. He will select one and we will show the poll “details”, that is, a form with the available choices and a button so he can submit his choice. Once a choice is made, the user will be directed to a page showing the current results of the poll he just voted on.

Before writing the code for the views: a good way to start designing a Django app is to design its URLs. In Django you map URLs to views, using regular expressions. Modern web development takes URLs seriously, and nice URLs (i.e. without cruft like “DoSomething.do” or “ThisIsNotNice.aspx”) are the norm. Instead of patching ugly names with URL rewriting, Django offers a layer of indirection between the URL which triggers a view and the internal name you happen to give to such view. Also, as Django has an emphasis on apps that can be reused across multiple projects, there is a modular way to define URLs so an app can define the relative URLs for its views, and they can be later included on different projects.

Let's start by modifying the `pollsite/urls.py` file to the following:

```
from django.conf.urls.defaults import *

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^admin/(.*)', admin.site.root),
    (r'^polls/', include('pollsite.polls.urls')),
)
```

Note how we added the pattern which says: if the URL starts with `polls/` continue mat matching it following the patters defined on module `pollsite.polls.urls`. So let's create the file `pollsite/polls/urls.py` (note that it will live inside the app) and put the following code in it:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('pollsite.polls.views',
    (r'^$', 'index'),
    (r'^(\d+)/$', 'detail'),
```

```
(r'^(\d+)/vote/$', 'vote'),
(r'^(\d+)/results/$', 'results'),
)
```

The first pattern says: If there is nothing else to match (remember that `polls/` was already matched by the previous pattern), use the `index` view. The others patterns include a placeholder for numbers, written in the regular expression as `\d+`, and it is captured (using the parenthesis) so it will be passed as argument to their respective views. The end result is that an URL like `polls/5/results/` will call the `results` view passing the string `'5'` as the second argument (the first view argument is always the request object). If you want to know more about Django URL dispatching, see <http://docs.djangoproject.com/en/1.0/topics/http/urls/>.

So, from the URL patterns we just created, it can be seen that we need to write the view functions named `index`, `detail`, `vote` and `results`. Here is code for `pollsite/polls/views.py`:

```
from django.shortcuts import get_object_or_404, render_to_response
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from pollsite.polls.models import Choice, Poll

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    return render_to_response('polls/index.html',
                              {'latest_poll_list': latest_poll_list})

def detail(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/detail.html', {'poll': poll})

def vote(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    try:
        selected_choice = poll.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the poll voting form.
        return render_to_response('polls/detail.html', {
            'poll': poll,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.
        return HttpResponseRedirect(
            reverse('pollsite.polls.views.results', args=(poll.id,)))

def results(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/results.html', {'poll': poll})
```

I know this was a bit fast, but remember that we are taking a *quick* tour. The important thing here is to grasp the high level concepts. Each function defined in this file is a view. You can identify them because, well, they are defined on the `views.py` file. But perhaps more importantly, because they receive a request as a first argument.

So, we defined the views named `index`, `details`, `vote` and `results` which are going to be called when an URL match the patterns defined previously. With the exception of `vote`, they are straightforward, and follow the same pattern: They search some data (using the Django ORM and helper functions like `get_object_or_404`

which, even if you aren't familiar with them it's easy to intuitively imagine what they do), and then end up calling `render_to_response`, passing the path of a template and a dictionary with the data passed to the template.

Note: The three trivial views described above represent cases so common in web development that Django provides an abstraction to implement them with even less code. The abstraction is called “Generic Views” and you can learn about them on <http://docs.djangoproject.com/en/1.0/ref/generic-views/>, as well as in the Django tutorial at <http://docs.djangoproject.com/en/1.0/intro/tutorial04/#use-generic-views-less-code-is-better>

The `vote` view is a bit more involved, and it ought to be, since it is the one which do interesting things, namely, registering a vote. It has two paths: one for the exceptional case in which the user has not selected any choice and one in which the user did select one. See how in the first case the view ends up rendering the same template which is rendered by the `detail` view: `polls/detail.html`, but we pass an extra variable to the template to display the error message so the user can know why he is still viewing the same page. In the successful case in which the user selected a choice, we increment the votes and *redirect* the user to the `results` view.

We could have archived the redirection by just calling the view (something like `return results(request, poll.id)`) but, as the comments say, is a good practice to do an *actual* HTTP redirect after POST submissions to avoid problems with the browser back button (or the refresh button). Since the view code don't know to what URLs they are mapped (as that is expected to change from site to site when you reuse the app) the `reverse` function gives you the URL for a given view and parameters.

Before taking a look at templates, a note about them. The Django template language is pretty simple and intentionally *not* as powerful as a programming language. You can't execute arbitrary python code nor call any function. It is designed this way to keep templates simple and webdesigner-friendly. The main features of the template language are expressions, delimited by double braces (`{{` and `}}`), and directives (called “template tags”), delimited by braces and the percent character (`{%` and `%}`). Expression can contain dots which do both attribute access and item access (so you write `{{ foo.bar }}` even if in Python you would write `foo['bar']`) and also pipes to apply filters to the expressions (like, for example, cut a string expression at some given maximum length). And that's pretty much it. You see how obvious they are on the following templates, but I'll give a bit of explanation when introducing some non obvious template tags.

Now, it's time to see the templates for our views. As you can infer by reading the views code we just wrote we need three templates: `polls/index.html`, `polls/detail.html` and `polls/results.html`. We will create the templates subdirectory inside the `polls` app, and then create the templates under it. So here is the content of `pollsite/polls/templates/polls/index.html`:

```
{% if latest_poll_list %}
<ul>
  {% for poll in latest_poll_list %}
    <li><a href="{{ poll.id }}">{{ poll.question }}</a></li>
  {% endfor %}
</ul>
{% else %}
<p>No polls are available.</p>
{% endif %}
```

Pretty simple, as you can see. Let's move to `pollsite/polls/templates/polls/detail.html`:

```
<h1>{{ poll.question }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="./vote/" method="post">
  {% for choice in poll.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}"
    value="{ { choice.id }}" />
```

```
<label for="choice{{ forloop.counter }}">{{ choice.choice }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

One perhaps surprising construct on this template is the `{{ forloop.counter }}` expression, which simply exposes the internal counter the surrounding `{% for %}` loop.

Also note that the `{% if %}` template tag will evaluate to false a expression that is not defined, as will be the case with `error_message` when this template is called from the detail view.

Finally, here is `pollsite/polls/templates/polls/results.html`:

```
<h1>{{ poll.question }}</h1>

<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice }} -- {{ choice.votes }}
        vote{{ choice.votes|pluralize }}</li>
{% endfor %}
</ul>
```

In this template you can see the use of a filter, in the expression `{{ choice.votes|pluralize }}`. It will output an “s” if the number of votes is greater than 1, and nothing otherwise. To learn more about the template tags and filters available by default in Django, see <http://docs.djangoproject.com/en/1.0/ref/templates/builtins/>. And to know more on how it works and how to create new filters and template tags, see <http://docs.djangoproject.com/en/1.0/ref/templates/api/>.

At this point we have a fully working poll site. It’s not pretty, and can use a lot of polishing. But it works! Try it navigating to <http://localhost:8000/polls/>.

Reusing Templates without “include”: Template Inheritance

Like many other template languages, Django also has a “include” directive. But it’s use is very rare, because there is a better solution for reusing templates: inheritance.

It works just like class inheritance. You define a base template, with many “blocks”. Each block has a name. Then other templates can inherit from the base template and override or extend the blocks. You are free to build inheritance chains of any length you want, just like with class hierarchies.

You may have noted that our templates weren’t producing valid HTML, but only fragments. It was convenient, to focus on the important parts of the templates, of course. But it also happens that with a very minor modification they will generate a complete, pretty HTML pages. As you probably guessed by now, they will extend from a site-wide base template.

Since I’m not exactly good with web design, we will take a ready-made template from <http://www.freecsstemplates.org/>. In particular, we will modify this template: <http://www.freecsstemplates.org/preview/exposure/>.

Note that the base template is going to be site-wide, so it belongs to the project, not to an app. We will create a `templates` subdirectory under the *project* directory. Here is the content for `pollsite/templates/base.html`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>Polls</title>
```

```
<link rel="alternate" type="application/rss+xml"
      title="RSS Feed" href="/feeds/polls/" />
<style>
  /* Irrelevant CSS code, see book sources if you are interested */
</style>
</head>
<body>
  <!-- start header -->
  <div id="header">
    <div id="logo">
      <h1><a href="/polls/">Polls</a></h1>
      <p>an example for the Jython book</a></p>
    </div>
    <div id="menu">
      <ul>
        <li><a href="/polls/">Home</a></li>
        <li><a href="/contact/">Contact Us</a></li>
        <li><a href="/admin/">Admin</a></li>
      </ul>
    </div>
  </div>
<!-- end header -->
<!-- start page -->
  <div id="page">
    <!-- start content -->
    <div id="content">
      {% block content %} {% endblock %}
    </div>
  <!-- end content -->
  <br style="clear: both;" />
</div>
<!-- end page -->
<!-- start footer -->
  <div id="footer">
    <p> <a href="/feeds/polls/">Subscribe to RSS Feed</a> </p>
    <p class="legal">
      &copy;2009 Apress. All Rights Reserved.
      &nbsp;&nbsp;&nbsp;&bull;&nbsp;&nbsp;&nbsp;
      Design by
      <a href="http://www.freecsstemplates.org/">Free CSS Templates</a>
      &nbsp;&nbsp;&nbsp;&bull;&nbsp;&nbsp;&nbsp;
      Icons by <a href="http://famfamfam.com/">FAMFAMFAM</a>. </p>
  </div>
<!-- end footer -->
</body>
</html>
```

As you can see, the template declares only one block, named “content” (near the end of the template before the footer). You can define as many blocks as you want, but to keep things simple we will do only one.

Now, to let Django find this template we need to tweak the settings. Edit `pollsite/settings.py` and locate the `TEMPLATE_DIRS` section. replace it with the following:

```
import os
TEMPLATE_DIRS = (
    os.path.dirname(__file__) + '/templates',
    # Put strings here, like "/home/html/django_templates" or
    # "C:/www/django/templates".
```

```

# Always use forward slashes, even on Windows.
# Don't forget to use absolute paths, not relative paths.
)

```

That's a trick to avoid hardcoding the project root directory. The trick may not work on all situations, but it will work for us. Now, that we have this `base.html` template in place, we will inherit from it in `pollsite/polls/templates/polls/index.html`:

```

{% extends 'base.html' %}
{% block content %}
{% if latest_poll_list %}
<ul>
    {% for poll in latest_poll_list %}
    <li><a href="{{ poll.id }}">{{ poll.question }}</a></li>
    {% endfor %}
</ul>
{% else %}
<p>No polls are available.</p>
{% endif %}
{% endblock %}

```

As you can see, the changes are limited to the addition of the two first lines and the last one. The practical implication is that the template is overriding the “content” block and inheriting all the rest. Do the same with the other two templates of the poll app and test the application again, visiting <http://localhost:8000/polls/>. It will look as shown on the figure *The Poll Site After Applying a Template*.

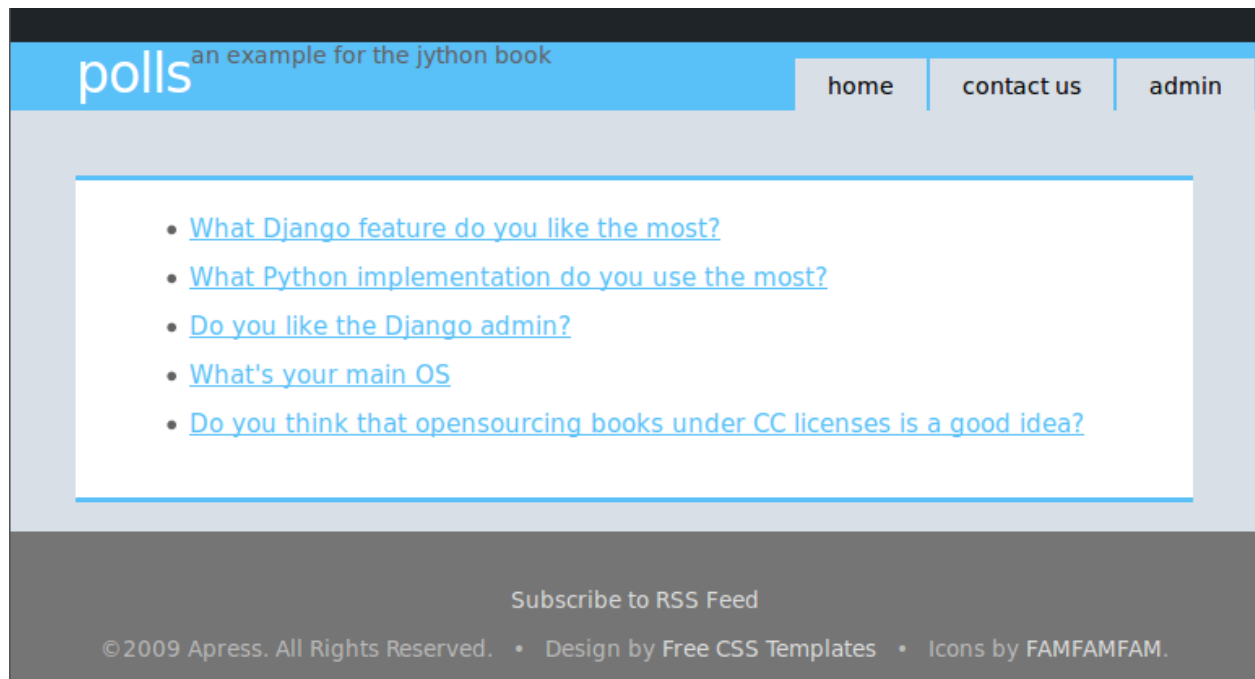


Fig. 4.5: The Poll Site After Applying a Template

At this point we could consider our sample web application to be complete. But I want to highlight some other features included in Django that can help you to develop your web apps (just like the admin). To showcase them we will add the following features to our site:

1. A contact form (note that the link is already included in our common base template)

2. A RSS feed for the latest polls (also note the link was already added on the footer)
3. User Comments on polls.

Forms

Django features some help to deal with HTML forms, which is always a bit tiresome. We will use this help to implement the “contact us” feature. Since it sounds like a common feature that could be reused on in the future, we will create a new app for it. Move to the project directory and run:

```
$ jython manage.py startapp contactus
```

Remember to add an entry for this app on `pollsite/settings.py` under the `INSTALLED_APPS` list as `'pollsite.contactus'`.

Then we will delegate URL matching the `/contact/` pattern to the app, by modifying `pollsite/urls.py` and adding one line for it:

```
from django.conf.urls.defaults import *

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^admin/(.*)', admin.site.root),
    (r'^polls/', include('pollsite.polls.urls')),
    (r'^contact/', include('pollsite.contactus.urls')),
)
```

Later, we create `pollsite/contactus/urls.py`. For simplicity’s sake we will use only one view to display and process the form. So the file `pollsite/contactus/urls.py` will simply consist of:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('pollsite.contactus.views',
    (r'^$', 'index'),
)
```

And the contents of `pollsite/contactus/views.py` is:

```
from django.shortcuts import render_to_response
from django.core.mail import mail_admins
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=200)
    email = forms.EmailField()
    title = forms.CharField(max_length=200)
    text = forms.CharField(widget=forms.Textarea)

def index(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            mail_admins(
                "Contact Form: %s" % form.title,
                "%s <%s> Said: %s" % (form.name, form.email, form.text))
```



```

        return render_to_response("contactus/success.html")
    else:
        form = ContactForm()
    return render_to_response("contactus/form.html", {'form': form})

```

The important bit here is the `ContactForm` class in which the form is declaratively defined and which encapsulates the validation logic. We just call the `is_valid()` method on our view to invoke that logic and act accordingly. See <http://docs.djangoproject.com/en/1.0/topics/email/#mail-admins> to learn about the `main_admins` function included on Django and how to adjust the project settings to make it work.

Forms also provide quick ways to render them in templates. We will try that now. This is the code for `pollsite/contactus/templates/contactus/form.html` which is the template used the the view we just wrote:

```

{% extends "base.html" %}
{% block content %}
<form action="." method="POST">
<table>
{{ form.as_p }}
</table>
<input type="submit" value="Send Message" >
</form>
{% endblock %}

```

Here we take advantage of the `as_table()` method of Django forms, which also takes care of rendering validation errors. Django forms also provide other convenience functions to render forms, but if none of them suits your need, you can always render the form in custom ways. See <http://docs.djangoproject.com/en/1.0/topics/forms/> for details on form handling.

Before testing this contact form, we need to write the template `pollsite/contactus/templates/contactus/success.html` which is also used from `pollsite.contactus.views.index::`. This template is quite simple:

```

{% extends "base.html" %}
{% block content %}
<h1> Send us a message </h1>
<p><b>Message received, thanks for your feedback!</b></p>
{% endblock %}

```

And we are done. Test it by navigation to <http://localhost:8000/contact/>. Try submitting the form without data, or with erroneous data (for example with an invalid email address). You will get something like what's shown on the figure *Django Form Validation in Action*. Without needing to write much code you get all that validation, almost for free. Of course the forms framework is extensible, so you can create custom form field types with their own validation or rendering code. Again, I'll refer you to <http://docs.djangoproject.com/en/1.0/topics/forms/> for detailed information.

Feeds

It's time to implement the feed we are offering on the link right before the footer. It surely won't surprise you to know that Django include ways to state declaratively your feeds and write them very quickly. Let's start by modifying `pollsite/urls.py` to leave it as follows:

```

from django.conf.urls.defaults import *
from pollsite.polls.feeds import PollFeed

from django.contrib import admin
admin.autodiscover()

```

The screenshot shows a web application titled "polls" with the subtitle "an example for the jython book". The navigation bar includes links for "home", "contact us", and "admin". The main content area is titled "Send us a message" and contains a form with the following fields:

- Name:** A text input field containing "Leo".
- Email:** A text input field containing "Not a valid mail". To the right of this field is a red error message: "• Enter a valid e-mail address."
- Title:** A text input field containing "Test".
- Text:** A large text area containing "Some text".

At the bottom of the form is a "Send Message" button. The footer of the page includes a link to "Subscribe to RSS Feed" and copyright information: "©2009 Apress. All Rights Reserved. • Design by Free CSS Templates • Icons by FAMFAMFAM."

Fig. 4.6: Django Form Validation in Action

```
urlpatterns = patterns('',
    (r'^admin/(.*)', admin.site.root),
    (r'^polls/', include('pollsite.polls.urls')),
    (r'^contact/', include('pollsite.contactus.urls')),
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
     {'feed_dict': {'polls': PollFeed}}),
)
```

The changes are the import of the `PollFeed` class (which we haven't wrote yet) and the last pattern for URLs starting with `/feeds/`, which will map to a built-in view which takes a dictionary with feeds as argument (in our case, `PollFeed` is the only one). Writing the this class, which will describe the feed, is very easy. Let's create the file `pollsite/polls/feeds.py` and put the following code on it:

```
from django.contrib.syndication.feeds import Feed
from django.core.urlresolvers import reverse
from pollsite.polls.models import Poll

class PollFeed(Feed):
    title = "Polls"
    link = "/polls"
    description = "Latest Polls"

    def items(self):
        return Poll.objects.all().order_by('-pub_date')

    def item_link(self, poll):
        return reverse('pollsite.polls.views.detail', args=(poll.id,))

    def item_pubdate(self, poll):
        return poll.pub_date
```

And we are almost ready. When a request for the URL `/feeds/polls/` is received by Django, it will use this feed description to build all the XML data. The missing part is how will be the content of polls displayed on the feeds. To do this, we need to create a template. By convention, it has to be named `feeds/<feed_name>_description.html`, where `<feed_name>` is what we specified as the key on the `feed_dict` in `pollsite/urls.py`. Thus we create the file `pollsite/polls/templates/feeds/polls_description.html` with this very simple content:

```
<ul>
    {% for choice in obj.choice_set.all %}
    <li>{{ choice.choice }}</li>
    {% endfor %}
</ul>
```

The idea is simple: Django passes each object returned by `PollFeed.items()` to this template, in which it takes the name `obj`. You then generate an HTML fragment which will be embedded on the feed result.

And that's all. Test it by pointing your browser to <http://localhost:8000/feeds/polls/>, or by subscribing to that URL with your preferred feed reader. Opera, for example, displays the feed as shown by figure *Poll feed displayed on the Opera browser*

Comments

Since comments are a common feature of current web sites, Django includes a mini-framework to make it easy the incorporation of comments to any project or app. I will show you how to use it in our project. First, add a new URL pattern for the Django comments app, so the `pollsite/urls.py` file will look like this:

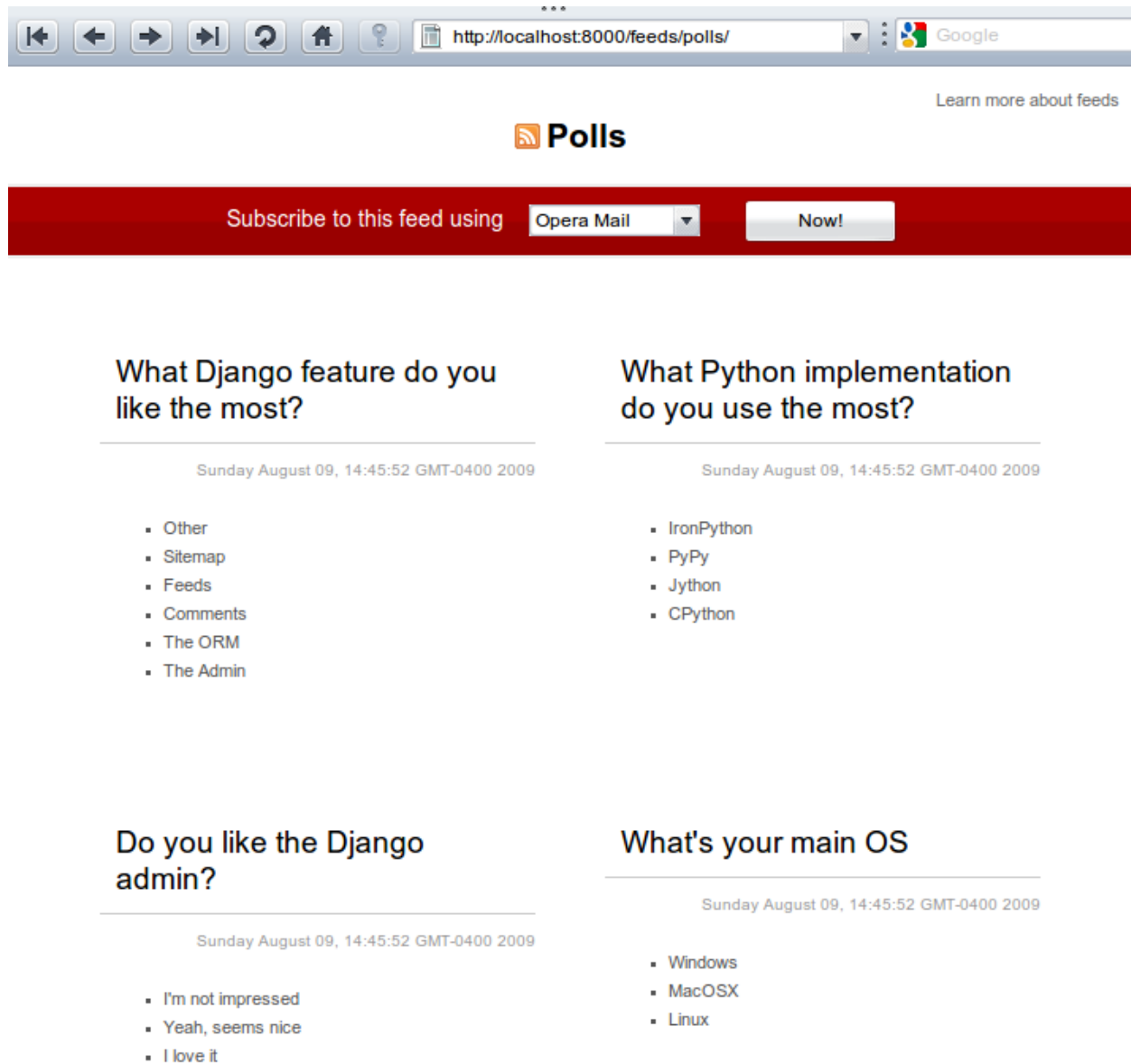


Fig. 4.7: Poll feed displayed on the Opera browser

```

from django.conf.urls.defaults import *
from pollsite.polls.feeds import PollFeed

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^admin/(.*)', admin.site.root),
    (r'^polls/', include('pollsite.polls.urls')),
    (r'^contact/', include('pollsite.contactus.urls')),
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
     {'feed_dict': {'polls': PollFeed}}),
    (r'^comments/', include('django.contrib.comments.urls')),
)

```

Then, add 'django.contrib.comments' to the INSTALLED_APPS on pollsite/settings.py. After that, we will let Django create the necessary tables by running:

```
$ jython manage.py syncdb
```

The comments will be added to the poll page, so we must edit pollsite/polls/templates/polls/detail.html. We will add the following code just before the {% endblock %} line which currently is the last line of the file:

```

{% load comments %}
{% get_comment_list for poll as comments %}
{% get_comment_count for poll as comments_count %}

{% if comments %}
<p>{{ comments_count }} comments:</p>
{% for comment in comments %}
<div class="comment">
    <div class="title">
        <p><small>
            Posted by <a href="{{ comment.user_url }}">{{ comment.user_name }}</a>,
            {{ comment.submit_date|timesince }} ago:
        </small></p>
    </div>
    <div class="entry">
        <p>
            {{ comment.comment }}
        </p>
    </div>
</div>

{% endfor %}

{% else %}
<p>No comments yet.</p>
{% endif %}

<h2>Left your comment:</h2>
{% render_comment_form for poll %}

```

Basically, we are importing the “comments” template tag library (by doing {% load comments %}) and then we just use it. It supports binding comments to *any database object*, so we don’t need to do anything special to make it work. The figure *Comments Powered Poll* shows what we get in exchange for that short snippet of code.

Do you think that opensourcing books under CC licenses is a good idea?

☐ No
☐ Yes

Vote

1 comments:

Posted by [Leo Soto](#), 19 minutes ago:

I also think we should highlight publishers that are friendly to the idea, like Apress

Left your comment:

Name

Email address

URL

Comment

Post

Preview

Fig. 4.8: Comments Powered Poll

If you try the application by yourself you will note that after submitting a comment you get a ugly page showing the success message. Or if you don't enter all the data, an ugly error form. That's because we are using the comments templates. A quick and effective fix for that is creating the file `pollsite/templates/comments/base.html` with the following content:

```
{% extends 'base.html' %}
```

Yeah, it's only one line! It shows the power of template inheritance: all what we need was to change the base template of the comments framework to inherit from our global base template.

And more...

At this point I hope you have learned to appreciate Django strenghts: It's a very good web framework in itself, but it also takes the "batteries included" philosophy and comes with solutions for many common problems in web development. This usually speed up a lot the process of creating a new website. And we didn't touched other common topics that Django provides out of the box like user authentication (from the login dialog to easily declaring which views require an authenticated user, or an user with some special characteristic like being an site administrator), or generic views.

But this book is about *Jython* and we will use the rest of this chapter to show the interesting possibilities that appear when you run Django on *Jython*. If you want to learn more about Django itself, I recommend (again) the excellent official documentation available on <http://docs.djangoproject.com/>.

J2EE deployment and integration

Although you *could* deploy your application using Django's built in development server, it's a terrible idea. The development server isn't designed to operate under heavy load and this is really a job that is more suited to a proper application server. We're going to install Glassfish v2.1 - an opensource highly performant JavaEE 5 application server from Sun Microsystems and show deployment onto it.

Let's install Glassfish now - obtain the release from

```
https://glassfish.dev.java.net/public/downloadsindex.html
```

At the time of this writing, Glassfish v3.0 is being prepared for release and it will support Django and Jython out of the box, but we'll stick to the stable release as the documentation and stability has been well established. Download the v2.1 release (currently v2.1-b60e). I strongly suggest you use JDK6 to do your deployment.

Once you have the installation JAR file, you can install it by issuing

```
% java -Xmx256m -jar glassfish-installer-v2.1-b60e-windows.jar
```

If your glassfish installer file has a different name, just use that instead of the filename listed in the above example. Be careful where you invoke this command though - Glassfish will unpack the application server into a subdirectory 'glassfish' in the directory that you start the installer.

One step that tripped me up during my impatient installation of Glassfish is that you actually need to invoke ant to complete the installation. On UNIX you need to invoke

```
% chmod -R +x lib/ant/bin
% lib/ant/bin/ant -f setup.xml
```

or for Windows

```
% lib\ant\bin\ant -f setup.xml
```

This will complete the setup - you'll find a bin directory with "asadmin" or "asadmin.bat" which will indicate that the application server has been installed.. You can start the server up by invoking

```
% bin/asadmin start_domain -v
```

On Windows, this will start the server in the foreground - the process will not daemonize and run in the background. On UNIX operating systems, the process will automatically daemonize and run in the background. In either case, once the server is up and running, you will be able to reach the web administration screen through a browser by going to <http://localhost:5000/>. The default login is 'admin' and the password is 'adminadmin'.

Currently, Django on Jython only supports the PostgreSQL database officially, but there is a preliminary release of a SQL Server backend as well as a SQLite3 backend. Let's get the PostgreSQL backend working - you will need to obtain the PostgreSQL JDBC driver from <http://jdbc.PostgreSQL.org>.

At the time of this writing, the latest version was in postgresql-8.4-701.jdbc4.jar, copy that jar file into your GLASSFISH_HOME/domains/domain1/lib directory. This will enable all your applications hosted in your appserver to use the same JDBC driver.

You should now have a GLASSFISH_HOME/domains/domain1/lib directory with the following contents

```
applibs/  
classes/  
databases/  
ext/  
postgresql-8.3-604.jdbc4.jar
```

You will need to stop and start the application server to let those libraries load up.

```
% bin/asadmin stop_domain  
% bin/asadmin start_domain -v
```

Deploying your first application

Django on Jython includes a built in command to support the creation of WAR files, but first, you will need to do a little bit of configuration you will need to make everything run smoothly. First we'll setup a simple Django application that has the administration application enabled so that we have some models that we play with. Create a project called 'hello' and make sure you add 'django.contrib.admin' and 'doj' applications to the INSTALLED_APPS.

Now enable the user admin by editing urls.py and uncomment the admin lines. Your urls.py should now look something like this

```
from django.conf.urls.defaults import *  
from django.contrib import admin  
admin.autodiscover()  
urlpatterns = patterns('',  
    (r'^admin/(.*)', admin.site.root),  
)
```

Disabling PostgreSQL logins

The first thing I inevitably do on a development machine with PostgreSQL is disable authentication checks to the database. The fastest way to do this is to enable only local connections to the database by editing the pg_hba.conf file. For PostgreSQL 8.3, this file is typically located in c:\PostgreSQL8.3\data/pg_hba.conf and on UNIXes - it is typically located in /etc/PostgreSQL/8.3/data/pg_hba.conf

At the bottom of the file, you'll find connection configuration information. Comment out all the lines and enable trusted connections from localhost. Your edited configuration should look something like this

#	TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
host	all	all	127.0.0.1/32	trust	

This will let any username password to connect to the database. You do not want to do this for a public facing production server. You should consult the PostgreSQL documentation for instructions for more suitable settings. After you've edited the connection configuration, you will need to restart the PostgreSQL server.

Create your PostgreSQL database using the createdb command now

```
> createdb demodb
```

Setting up the database is straightforward - just enable the pgsql backend from Django on Jython. Note that backend will expect a username and password pair even though we've disabled them in PostgreSQL. You can populate anything you want for the DATABASE_NAME and DATABASE_USER settings. The database section of your settings module should now look something like this

```
DATABASE_ENGINE = 'doj.backends.zxjdbc.postgresql'
DATABASE_NAME = 'demodb'
DATABASE_USER = 'ngvictor'
DATABASE_PASSWORD = 'nosecrets'
```

Initialize your database now

```
> jython manage.py syncdb
Creating table django_admin_log
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
```

You just installed Django's auth system, which means you don't have any superusers defined. Would you like to create one now? (yes/no): yes Username: admin E-mail address: admin@abc.com Warning: Problem with getpass. Passwords may be echoed. Password: admin Warning: Problem with getpass. Passwords may be echoed. Password (again): admin Superuser created successfully. Installing index for admin.LogEntry model Installing index for auth.Permission model Installing index for auth.Message model

All of this should be review so far, now we're going to take the application and deploy it into the running Glassfish server. This is actually the easy part. Django on Jython comes with a custom 'war' command that builds a self contained file which you can use to deploy into any Java servlet container.

A note about WAR files

For JavaEE servers, a common way to deploy your applications is to deploy a 'WAR' file. This is just a fancy name for a zip file that contains your application and any dependencies it requires that the application server has not made available as a shared resource. This is a robust way of making sure that you minimize the impact of versioning changes of libraries if you want to deploy multiple applications in your app server.

Consider your Django applications over time - you will undoubtedly upgrade your version of Django, you may upgrade the version of your database drivers - you may even decide to upgrade the version of the Jython language you wish to deploy on. These choices are ultimately up to you if you bundle all your dependencies in your WAR file. By bundling up all your dependencies into your WAR file, you can ensure that your app will "just work" when you go to deploy it. The server will automatically partition each application into its own space with concurrently running versions of the same code.

To enable the war command, add the 'doj' application to your settings in the INSTALLED_APPS list. Next, you will need to enable your site's media directory and a context relative root for your media. Edit your settings.py module so that that your media files are properly configured to be served. The war command will automatically configure your media files so that they are served using a static file servlet and the URLs will be remapped to be after the context root.

Edit your settings module and configure the MEDIA_ROOT and MEDIA_URL lines.

```
MEDIA_ROOT = 'c:\dev\hello\media_root' MEDIA_URL = '/site_media/'
```

Now you will need to create the media_root subdirectory under your 'hello' project and drop in a sample file so you can verify that static content serving is working. Place a file "sample.html" into your media_root directory. Put whatever contents you want into it - we're just using this to ensure that static files are properly served.

In english - that means when the above configuration is used - 'hello' will be deployed into your servlet container and the container will assign some URL path to be the 'context root' in Glassfish's case - this means your app will live in '<http://localhost:8000/hello/>'. The site_media directory will be visible at "http://localhost:8000/hello/site_media". DOJ will automatically set the static content to be served by Glassfish's fileservlet which is already highly performant. There is no need to setup a separate static file server for most deployments.

Build your war file now using the standard manage.py script, and deploy using the asadmin tool

```
c:\dev\hello>jython manage.py war

Assembling WAR on c:\docume~1\ngvictor\locals~1\temp\tmpl__snn\hello

Copying WAR skeleton...
Copying jython.jar...
Copying Lib...
Copying django...
Copying media...
Copying hello...
Copying site_media...
Copying doj...
Building WAR on C:\dev\hello.war...
Cleaning c:\docume~1\ngvictor\locals~1\temp\tmpl__snn...

Finished.

Now you can copy C:\dev\hello.war to whatever location your application server wants,
→it.

C:\dev\hello>cd \glassfish
C:\glassfish>bin\asadmin.bat deploy hello.war
Command deploy executed successfully.

C:\glassfish>
```

That's it. You should now be able to see your application running on

```
http://localhost:8080/hello/
```

The administration screen should also be visible at :

```
http://localhost:8080/hello/admin/
```

You can verify that your static media is being served correctly by going to:

```
http://localhost:8080/hello/site\_media/sample.html
```

That's it. Your basic deployment to a servlet container is now working.

Extended installation

The `war` command in `doj` provides extra options for you to specify extra JAR files to include with your application and which can bring down the size of your WAR file. By default, the `'war'` command will bundle the following items:

- Jython
- Django and it's administration media files
- your project and media files
- all of your libraries in site-packages

You can specialize your WAR file to include specific JAR files and you can instruct `doj` to assemble a WAR file with just the python packages that you require. The respective options for `"manage.py war"` are `"--include-py-packages"` and `"--include-jar-libs"`. The basic usage is straight forward, simply pass in the location of your custom python packages and the JAR files to these two arguments and `distutils` will automatically decompress the contents of those compressed volumes and recompress them into your WAR file.

To bundle JAR files up, you will need to specify a list of files to `"--include-java-libs"`.

The following example bundles the `jTDS` JAR file and a regular python module called `urllib3` with our WAR file.:

```
$ jython manage.py war --include-java-libs=$HOME/downloads/jtds-1.2.2.jar \
    --include-py-package=$HOME/PYTHON_ENV/lib/python2.5/site-packages/urllib3
```

You can have multiple JAR files or python packages listed, but you must delimit them with your operating system's path separator. For UNIX systems - this means `":"` and for Windows it is `";"`.

Eggs can also be installed using `"--include-py-path-entries"` using the egg filename. For example

```
$ jython manage.py war --include-py-path-entries=$HOME/PYTHON_ENV/lib/python2.5/site-
↳ packages/urllib3
```

Connection pooling with JavaEE

Whenever your web application goes to fetch data from the database, that data has to come back over a database connection. Some databases have 'cheap' database connections like MySQL, but for many databases - creating and releasing connections is quite expensive. Under high load conditions - opening and closing database connections on every request can quickly consume too many file handles - and your application will crash.

The general solution to this is to employ database connection pooling. While your application will continue to create new connections and close them off, a connection pool will manage your database connections from a reusable set. When you go to close your connection - the connection pool will simply reclaim your connection for use at a later time. Using a pool means you can put an enforced upper limit restriction on the number of concurrent connections to the database. Having that upper limit means you can reason about how your application will perform when the upper limit of database connections is hit.

While Django does not natively support database connection pools with CPython, you can enable them in the PostgreSQL driver for Django on Jython. Creating a connection pool that is visible to Django/Jython is a two step process in Glassfish. First, we'll need to create a JDBC connection pool, then we'll need to bind a JNDI name to that pool. In a JavaEE container, JNDI - the Java Naming and Directory Interface - is a registry of names bound to objects. It's really best thought of as a hashtable that typically abstracts a factory that emits objects.

In the case of database connections - JNDI abstracts a `ConnectionFactory` which provides proxy objects that behave like database connections. These proxies automatically manage all the pooling behavior for us. Lets see this in practice now.

First we'll need to create a JDBC ConnectionFactory. Go to the administration screen of Glassfish and go down to Resources/JDBC/JDBC Resources/Connection Pools. From there you can click on the 'New' button and start to configure your pool.

Set the name to "pgpool-demo", the resource type should be "javax.sql.ConnectionPoolDataSource" and the Database Vendor should be PostgreSQL. Click 'Next'.

At the bottom of the next page, you'll see a section with "Additional Properties". You'll need to set four parameters to make sure the connection is working, assuming that the database is configured for a username/password of ngvictor/nosecrets - here's what you need to connect to your database.

Name	Value
databaseName	demodb
serverName	localhost
password	nosecrets
user	ngvictor

You can safely delete all the other properties - they're not needed. Click 'Finish'.

Your pool will now be visible on the left hand tree control in the Connection Pools list. Select it and try pinging it to make sure it's working. If all is well, Glassfish will show you a successful Ping message.

We now need to bind a JNDI name to the connection factory to provide a mechanism for Jython to see the pool. Go to the JDBC Resources and click 'New'. Use the JNDI name: "jdbc/pgpool-demo", select the 'pgpool-demo' as your pool name and hit "OK">

Verify from the command line that the resource is available

```
glassfish\bin $ asadmin list-jndi-entries --context jdbc
Jndi Entries for server within jdbc context:
pgpool-demo__pm: javax.naming.Reference
__TimerPool: javax.naming.Reference
__TimerPool__pm: javax.naming.Reference
pgpool-demo: javax.naming.Reference
Command list-jndi-entries executed successfully.
```

Now, we need to enable the Django application use the JNDI name based lookup if we are running in an application server, and fail back to regular database connection binding if JNDI can't be found. Edit your settings.py module and add an extra configuration to enable JNDI.

```
DATABASE_ENGINE = 'doj.backends.zxjdbc.postgresql'
DATABASE_NAME = 'demodb'
DATABASE_USER = 'ngvictor'
DATABASE_PASSWORD = 'nosecrets'
DATABASE_OPTIONS = {'RAW_CONNECTION_FALLBACK': True, \
                    'JNDI_NAME': 'jdbc/pgpool-demo' }
```

Note that we're duplicating the configuration to connect to the database. This is because we want to be able to fall back to regular connection binding in the event that JNDI lookups fail. This makes our life easier when we're running in a testing or development environment.

That's it.

You're finished configuring database connection pooling. That wasn't that bad now was it?

Dealing with long running tasks

When you're building a complex web application, you will inevitably end up having to deal with processes which need to be processed in the background. If you're building on top of CPython and Apache, you're out of luck here - there's

no standard infrastructure available for you to handle these tasks. Luckily these services have had years of engineering work already done for you in the Java world. We'll take a look at two different strategies for dealing with long running tasks.

Thread Pools

The first strategy is to leverage managed thread pools in the JavaEE container. When your webapplication is running within Glassfish, each HTTP request is processed by the HTTP Service which contains a threadpool. You can change the number of threads to affect the performance of the webserver. Glassfish will also let you create your own threadpools to execute arbitrary work units for you.

The basic API for threadpools is simple:

- WorkManager which provides an abstracted interface to the thread pool
- Work is an interface which encapsulates your unit of work
- WorkListener which is an interface that lets you monitor the progress of your Work tasks.

First, we need to tell Glassfish to provision a threadpool for our use. In the Administration screen, go down to Configuration/Thread Pools. Click on 'New' to create a new thread pool. Give your threadpool the name "backend-workers". Leave all the other settings as the default values and click "OK".

You've now got a thread pool that you can use. The threadpool exposes an interface where you can submit jobs to the pool and the pool will either execute the job synchronously within a thread, or you can schedule the job to run asynchronously. As long as your unit of work implements the `javax.resource.spi.work.Work` interface, the threadpool will happily run your code. A unit of class may be as simple as the following snippet of code

```
from javax.resource.spi.work import Work

class WorkUnit(Work):
    """
    This is an implementation of the Work interface.
    """
    def __init__(self, job_id):
        self.job_id = job_id

    def release(self):
        """
        This method is invoked by the threadpool to tell threads
        to abort the execution of a unit of work.
        """
        logger.warn("[%d] Glassfish asked the job to stop quickly" % self.job_id)

    def run(self):
        """
        This method is invoked by the threadpool when work is
        'running'
        """
        for i in range(20):
            logger.info("[%d] just doing some work" % self.job_id)
```

This WorkUnit class above doesn't do anything very interesting, but it does illustrate the basic structure of what unit of work requires. We're just logging message to disk so that we can visually see the thread execute.

WorkManager implements several methods which can run your job and block until the threadpool completes your work, or it can run the job asynchronously. Generally, I prefer to run things asynchronously and simply check the status of the work over time. This lets me submit multiple jobs to the threadpool at once and check the status of each of the jobs.

To monitor the progress of work, we need to implement the `WorkListener` interface. This interface gives us notifications as a task progresses through the 3 phases of execution within the thread pool. Those states are :

1. Accepted
2. Started
3. Completed

All jobs must go to either Completed or Rejected states. The simplest thing to do then is to simply build up lists capturing the events. When the length of the completed and the rejected lists together are the same as the number of jobs we submitted, we know that we are done. By using lists instead of simple counters, we can inspect the work objects in much more detail.

Here's the code for our `SimpleWorkListener`

```
from javax.resource.spi.work import WorkListener
class SimpleWorkListener(WorkListener):
    """
    Just keep track of all work events as they come in
    """
    def __init__(self):
        self.accepted = []
        self.completed = []
        self.rejected = []
        self.started = []

    def workAccepted(self, work_event):
        self.accepted.append(work_event.getWork())
        logger.info("Work accepted %s" % str(work_event.getWork()))

    def workCompleted(self, work_event):
        self.completed.append(work_event.getWork())
        logger.info("Work completed %s" % str(work_event.getWork()))

    def workRejected(self, work_event):
        self.rejected.append(work_event.getWork())
        logger.info("Work rejected %s" % str(work_event.getWork()))

    def workStarted(self, work_event):
        self.started.append(work_event.getWork())
        logger.info("Work started %s" % str(work_event.getWork()))
```

To access the threadpool, you simply need to know the name of the pool we want to access and schedule our jobs. Each time we schedule a unit of work, we need to tell the pool how long to wait until we timeout the job and provide a reference to the `WorkListener` so that we can monitor the status of the jobs.

The code to do this is listed below

```
from com.sun.enterprise.connectors.work import CommonWorkManager
from javax.resource.spi.work import Work, WorkManager, WorkListener
wm = CommonWorkManager('backend-workers')
listener = SimpleWorkListener()
for i in range(5):
    work = WorkUnit(i)
    wm.scheduleWork(work, -1, None, listener)
```

You may notice that the `scheduleWork` method takes in a `None` in the third argument. This is the execution context - for our purposes, it's best to just ignore it and set it to `None`. The `scheduleWork` method will return immediately and

the listener will get callback notifications as our work objects pass through. To verify that all our jobs have completed (or rejected) - we simply need to check the listener's internal lists.

```
while len(listener.completed) + len(listener.rejected) < num_jobs:
    logger.info("Found %d jobs completed" % len(listener.completed))
    time.sleep(0.1)
```

That covers all the code you need to access thread pools and monitor the status of each unit of work. Ignoring the actual WorkUnit class, the actual code to manage the threadpool is about a dozen lines long.

JavaEE standards and thread pools

Unfortunately, this API is not standard in the JavaEE 5 specification yet so the code listed here will only work in Glassfish. The API for parallel processing is being standardized for JavaEE 6, and until then you will need to know a little bit of the internals of your particular application server to get threadpools working. If you're working with Weblogic or Websphere, you will need to use the CommonJ APIs to access the threadpools, but the logic is largely the same.

Passing messages across process boundaries

While threadpools provide access to background job processing, sometimes it may be beneficial to have messages pass across process boundaries. Every week there seems to be a new Python package that tries to solve this problem, for Jython we are lucky enough to leverage Java's JMS. JMS specifies a message brokering technology where you may define publish/subscribe or point to point delivery of messages between different services. Messages are asynchronously sent to provide loose coupling and the broker deals with all manner of boring engineering details like delivery guarantees, security, durability of messages between server crashes and clustering.

While you could use a handrolled RESTful messaging implementation - using OpenMQ and JMS has many advantages.

1. It's mature. Do you really think your messaging implementation handles all the corner cases? Server crashes? Network connectivity errors? Reliability guarantees? Clustering? Security? OpenMQ has almost 10 years of engineering behind it. There's a reason for that.
2. The JMS standard is just that - standard. You gain the ability to send and receive messages between any JavaEE code.
3. Interoperability. JMS isn't the only messaging broker in town. The Streaming Text Orientated Messaging Protocol (STOMP) is another standard that is popular amongst non-Java developers. You can turn a JMS broker into a STOMP broker by using stompconnect. This means you can effectively pass messages between any messaging client and any messaging broker using any of a dozen different languages.

In JMS there are two types of message delivery mechanisms:

- Publish/Subscribe: This is for the times when we want to message one or more subscribers about events that are occurring. This is done through JMS 'topics'.
- Point to point messaging: These are single sender, single receiver message queues. Appropriately, JMS calls these 'queues'

We need to provision a couple objects in Glassfish to get JMS going. In a nutshell, we need to create a connection factory which clients will use to connect to the JMS broker. We'll create a publish/subscribe resource and a point to point messaging queue. In JMS terms, there are called "destinations". They can be thought of as postboxes that you send your mail to.

Go to the Glassfish administration screen and go to Resources/JMS Resources/Connection Factories. Create a new connection factory with the JNDI name: "jms/MyConnectionFactory". Set the resource type to

javax.jms.ConnectionFactory. Delete the username and password properties at the bottom of the screen and add a single property: 'imqDisableSetClientID' with a value of 'false'. Click 'OK'.

TODO screenshot of the property setting

By setting the imqDisableSetClientID to false, we are forcing clients to declare a username and password when they use the ConnectionFactory. OpenMQ uses the login to uniquely identify the clients of the JMS service so that it can properly enforce the delivery guarantees of the destination.

We now need to create the actual destinations - a topic for publish/subscribe and a queue for point to point messaging. Go to Resources/JMS Resources/Destination Resources and click 'New'. Set the JNDI name to 'jms/MyTopic', the destination name to 'MyTopic' and the Resource type to be 'javax.jms.Topic'. Click "OK" to save the topic.

TODO: create the topic image

Now we need to create the JMS queue for point to point messages. Create a new resource, set the JNDI name to 'jms/MyQueue', the destination name to 'MyQueue' and the resource type to "javax.jms.Queue". Click OK to save.

TODO: create the queue image

Like the database connections discussed earlier, the JMS services are also acquired in the JavaEE container through the use of JNDI name lookups. Unlike the database code, we're going to have to do some manual work to acquire the naming context which we do our lookups against. When our application is running inside of Glassfish, acquiring a context is very simple. We just import the class and instantiate it. The context provides a lookup() method which we use to acquire the JMS connection factory and get access to the particular destinations that we are interested in. In the following example, I'll publish a message onto our topic. Lets see some code first and I'll go over the finer details of what's going on

```
from javax.naming import InitialContext, Session
from javax.naming import DeliverMode, Message
context = InitialContext()

tfactory = context.lookup("jms/MyConnectionFactory")

tconnection = tfactory.createTopicConnection('senduser', 'sendpass')
tsession = tconnection.createTopicSession(False, Session.AUTO_ACKNOWLEDGE)
publisher = tsession.createPublisher(context.lookup("jms/MyTopic"))

message = tsession.createTextMessage()
msg = "Hello there : %s" % datetime.datetime.now()
message.setText(msg)
publisher.publish(message, DeliverMode.PERSISTENT,
                  Message.DEFAULT_PRIORITY, 100)
tconnection.close()
context.close()
```

In this code snippet, we acquire a topic connection through the connection factory. To reiterate - topics are for publish/subscribe scenarios. We create a topic session - a context where we can send and receive messages to next. The two arguments passed to creating the topic session specify a transactional flag and how our client will acknowledge receipt of messages. We're going to just disable transactions and get the session to automatically send acknowledgements back to the broker on message receipt.

The last step to getting our publisher is well - creating the publisher. From there we can start publishing messages up to the broker.

At this point - it is important to distinguish between persistent messages and durable messages. JMS calls a message 'persistent' if the messages received by the *broker* are persisted. This guarantees that senders know that the broker has received a message. It makes no guarantee that messages will actually be delivered to a final recipient.

Durable subscribers are guaranteed to receive messages in the case that they temporarily drop their connection to the broker and reconnect at a later time. The JMS broker will uniquely identify subscriber clients with a combination of

the client ID, username and password to uniquely identify clients and manage message queues for each client.

Now we need to create the subscriber client. We're going to write a standalone client to show that your code doesn't have to live in the application server to receive messages. The only trick we're going to apply here is that while we can simply create an InitialContext with an empty constructor for code in the app server, code that exists outside of the application server must know where to find the JNDI naming service. Glassfish exposes the naming service via CORBA - the Common Object Request Broker Architecture. In short - we need to know a factory class name to create the context and we need to know the URL of where the object request broker is located.

The following listener client can be run on the same host as the Glassfish server

```
"""
This is a standalone client that listens messages from JMS
"""
from javax.jms import TopicConnectionFactory, MessageListener, Session
from javax.naming import InitialContext, Context
import time

def get_context():
    props = {}
    props[Context.INITIAL_CONTEXT_FACTORY]="com.sun.appserv.naming.SLASCtxFactory"
    props[Context.PROVIDER_URL]="iiop://127.0.0.1:3700"
    context = InitialContext(props)
    return context

class TopicListener(MessageListener):
    def go(self):
        context = get_context()
        tfactory = context.lookup("jms/MyConnectionFactory")
        tconnection = tfactory.createTopicConnection('recvuser', 'recvpass')
        tsession = tconnection.createTopicSession(False, Session.AUTO_ACKNOWLEDGE)
        subscriber = tsession.createDurableSubscriber(context.lookup("jms/MyTopic"),
        ↪ 'mysub')
        subscriber.setMessageListener(self)
        tconnection.start()
        while True:
            time.sleep(1)
            context.close()
            tconnection.close()

        def onMessage(self, message):
            print message.getText()

if __name__ == '__main__':
    TopicListener().go()
```

There are only a few key differences between the subscriber and publisher side of a JMS topic. First, the subscriber is created with a unique client id - in this case - it's 'mysub'. This is used by JMS to determine what pending messages to send to the client in the case that the client drops the JMS connections and rebinds at a later time. If we don't care to receive missed messages, we could have created a non-durable subscriber with "createSubscriber" instead of "createDurableSubscriber" and we would not have to pass in a client ID.

Second, the listener employs a callback pattern for incoming messages. When a message is received, the onMessage will be called automatically by the subscriber object and the message object will be passed in.

Now we need to create our sending user and receiving user on the broker. Drop to the command line and go to GLASSFISH_HOME/imq/bin. We are going to create two users - one sender and one receiver.

```
GLASSFISH_HOME/imq/bin $ imqusermgr add -u senduser -p sendpass
User repository for broker instance: imqbroker
User senduser successfully added.

GLASSFISH_HOME/imq/bin $ imqusermgr add -u recvuser -p recvpass
User repository for broker instance: imqbroker
User recvuser successfully added.
```

We now have two new users with username/password pairs of senduser/sendpass and recvuser/recvpass.

You have enough code now to enable publish/subscribe messaging patterns in your code to signal applications that live outside of your application server. We can potentially have multiple listeners attached to the JMS broker and JMS will make sure that all subscribers get messages in a reliable way.

Let's take a look now at sending message through a queue - this provides reliable point to point messaging and it adds guarantees that messages are persisted in a safe manner to safeguard against server crashes. This time, we'll build our send and receive clients as individual standalone clients that communicate with the JMS broker.

```
from javax.jms import Session
from javax.naming import InitialContext, Context
import time

def get_context():
    props = {}
    props[Context.INITIAL_CONTEXT_FACTORY]="com.sun.appserv.naming.SlASCtxFactory"
    props[Context.PROVIDER_URL]="iiop://127.0.0.1:3700"
    context = InitialContext(props)
    return context

def send():
    context = get_context()
    qfactory = context.lookup("jms/MyConnectionFactory")
    # This assumes a user has been provisioned on the broker with
    # username/password of 'senduser/sendpass'
    qconnection = qfactory.createQueueConnection('senduser', 'sendpass')
    qsession = qconnection.createQueueSession(False, Session.AUTO_ACKNOWLEDGE)
    qsender = qsession.createSender(context.lookup("jms/MyQueue"))
    msg = qsession.createTextMessage()
    for i in range(20):
        msg.setText('this is msg [%d]' % i)
        qsender.send(msg)

def recv():
    context = get_context()
    qfactory = context.lookup("jms/MyConnectionFactory")
    # This assumes a user has been provisioned on the broker with
    # username/password of 'recvuser/recvpass'
    qconnection = qfactory.createQueueConnection('recvuser', 'recvpass')
    qsession = qconnection.createQueueSession(False, Session.AUTO_ACKNOWLEDGE)
    qreceiver = qsession.createReceiver(context.lookup("jms/MyQueue"))
    qconnection.start() # start the receiver

    print "Starting to receive messages now:"
    while True:
        msg = qreceiver.receive(1)
        if msg <> None and isinstance(msg, TextMessage):
            print msg.getText()
```

The `send()` and `recv()` functions are almost identical to the publish/subscriber code used to manage topics. A minor difference is that the JMS queue APIs don't use a callback object for message receipt. It is assumed that client applications will actively dequeue objects from the JMS queue instead of acting as a passive subscriber.

The beauty of this JMS code is that you can send messages to the broker and be assured that even in case the server goes down, your messages are not lost. When the server comes back up and your endpoint client reconnects - it will still receive all of its pending messages.

We can extend this example even further. Codehaus.org has a messaging project called STOMP - the Streaming Text Orientated Messaging Protocol. STOMP is simpler, but less performant than raw JMS messages, but the tradeoff is that clients exist in a dozen different languages. STOMP also provides an adapter called 'stomp-connect' which allows us to turn a JMS broker into a STOMP messaging broker.

This will enable us to have applications written in just about *any* language communicate with our applications over JMS. There are times when I have existing CPython code that leverages various C libraries like Imagemagick or NumPy to do computations that are simply not supported with Jython or Java.

By using `stompconnect`, I can send work messages over JMS, bridge those messages over STOMP and have CPython clients process my requests. The completed work is then sent back over STOMP, bridged to JMS and received by my Jython code.

First, you'll need to obtain latest version of `stomp-connect` from codehaus.org. Download `stompconnect-1.0.zip` from here :

<http://stomp.codehaus.org/Download>

After you've unpacked the zip file, you'll need to configure a JNDI property file so that STOMP can act as a JMS client. The configuration is identical to our Jython client. Create a file called "jndi.properties" and place it in your `stompconnect` directory. The contents should have the two following lines

```
java.naming.factory.initial=com.sun.appserv.naming.SLASCtxFactory
java.naming.provider.url=iiop://127.0.0.1:3700
```

You now need to pull in some JAR files from Glassfish to gain access to JNDI, JMS and some logging classes that STOMP requires. Copy the following JAR files from `GLASSFISH_HOME/lib` into `STOMPCONNECT_HOME/lib` :

- `appserv-admin.jar`
- `appserv-deployment-client.jar`
- `appserv-ext.jar`
- `appserv-rt.jar`
- `j2ee.jar`
- `javaee.jar`

Copy the `imqjmsra.jar` file from `GLASSFISH_HOME/imq/lib/imqjmsra.jar` to `STOMPCONNECT_HOME/lib`.

You should be able to now start the connector with the following command line

```
java -cp "lib\*;stompconnect-1.0.jar" \
    org.codehaus.stomp.jms.Main tcp://0.0.0.0:6666 \
    "jms/MyConnectionFactory"
```

If it works, you should see a bunch of output that ends with a message that the server is listening for connection on `tcp://0.0.0.0:6666`. Congratulations, you now have a STOMP broker acting as a bidirectional proxy for the OpenMQ JMS broker.

Receiving messages in CPython that originate from Jython+JMS is as simple as the following code.

```
import stomp
serv = stomp.Stomp('localhost', 6666)
serv.connect({'client-id': 'reader_client', \
            'login': 'recvuser', \
            'passcode': 'recvpass'})
serv.subscribe({'destination': '/queue/MyQueue', 'ack': 'client'})
frame = self.serv.receive_frame()
if frame.command == 'MESSAGE':
    # The message content will arrive in the STOMP frame's
    # body section
    print frame.body
    serv.ack(frame)
```

Sending messages is just as straight forward. From our CPython ocde, we just need to import the stomp client and we can send messages back to our Jython code.

```
import stomp
serv = stomp.Stomp('localhost', 6666)
serv.connect({'client-id': 'sending_client', \
            'login': 'senduser', \
            'passcode': 'sendpass'})
serv.send({'destination': '/queue/MyQueue', 'body': 'Hello world!'})
```

Conclusion

We’ve covered a lot of ground here. I’ve shown you how to get Django on Jython to use database connection pooling to enforce limits on the database resources an application can consume. We’ve looked at setting up JMS queues and topic to provide both point to point and publish/subscribe messages between Jython processes. We then took those messaging services and provided interoperability between Jython code and non-Java code.

In my experience, the ability to remix a hand picked collection of technologies is what gives Jython so much power. You can use both the technology in JavaEE, leveraging years of hard won experience and get the benefit of using a lighter weight, more modern web application stack like Django.

The future of Jython and Django support in application server is very promising. Websphere now uses Jython for it’s official scripting language and the version 3 release of Glassfish will offer first class support of Django applications. You’ll be able to deploy your web applications without building WAR files up. Just deploy straight from your source directory and you’re off to the races.

Chapter 15: Introduction to Pylons

While Django is currently the most popular webframework for Python, it is by no means your only choice. Where Django grew out of the needs of newsrooms to implement content management solutions rapidly - Pylons grew out of a need to build web applications in environments that may have existing databases to integrate with and the applications don’t fit neatly into the class of applications that are loosely defined in the “content management” space.

Pylons greatest strength is that it takes a best of breed approach to constructing it’s technology stack. Where everything is ‘built in’ with Django and the entire application stack is specifically designed with a single world view of how applications should be done - Pylons takes precisely the opposite approach. Pylons - the core codebase that lives in the ‘pylons’ namespace - it’s remarkably small. With the 0.9.7 release, it’s hovering around 5,500 lines of code. Django by comparison weighs in at about 125,000 lines of code.

Pylons manages to do this magic by leveraging existing libraries extensively and the Pylons community works with many other Python projects to develop standard APIs to promote interoperability.

Ultimately picking Django or Pylons is about deciding which tradeoffs you're willing to make. While Django is extremely easy to learn because all the documentation is in one place and all the documentation relating to any particular component is always discussed in the context of building a web application - you lose some flexibility when you need to start doing things that are at the margins of what Django was designed for.

For example, in a project I've worked on recently, I needed to interact with a nontrivial database that was implemented in SQL Server 2000. For Django, implementing the SQL Server backend was quite difficult. There aren't that many web developers using Django on Windows, never mind SQL Server. While the Django ORM is a part of Django, it is also not the core focus of Django. Supporting arbitrary databases is simply not a goal for Django and rightly so.

The Django project assumes you'll do a rational thing and simply use Postgresql as your database backend. Web developers have better things to do than build backend drivers for every possible database.

Pylons on the other hand leverages SQLAlchemy. SQLAlchemy is probably the most powerful database toolkit available in Python. It *only* focuses on database access. The SQL Server backend was already built in a robust way for CPython and implementing the extra code for a Jython backend took 2 days - and this was from not seeing any of the code in SQLAlchemy's internals.

That experience alone sold me on Pylons. I don't have to rely the 'webframework' people to experts in databases. Similarly, I don't have to rely on the database experts to know anything about web templating.

In short when you have to deal with the weird stuff - Pylons makes a fabulous choice - and lets be honest - there's almost always weird stuff you're going to have to do.

A guide for the impatient

The best way to install Pylons is inside of a virtualenv. Create a new virtualenv for Jython and run `easy_install`

```
> easy_install "Pylons==0.9.7"
```

Create your application

```
> paster create --template=pylons RosterTool
# TODO: just use the defaults for everything. No sqlalchemy
```

Launch the development server

```
> paster serve --reload development.ini
```

Open a browser and connect to <http://127.0.0.1:5000/>

TODO: include screenshot here

Drop a static file into `rosterool/public/welcome.html`

```
<html>
  <body>Just a static file</body>
</html>
```

You should now be able to load the static content by going to

```
http://localhost:5000/welcome.html
```

Add a controller

```
RosterTool/roster > paster controller roster
```

Paste will install a directory named “controllers” and install some files in there including a module named “roster.py”. You can open it up and you’ll see a class named “RosterController” and it will have a single method “index”. Pylons is smart enough to automatically map a URL to a controller classname and invoke a method. To invoke the RosterController’s index method, you just need to invoke

```
http://localhost:5000/roster/index
```

Congratulations, you’ve got your most basic possible web application running now. It handles basic HTTP GET requests and calls a method on a controller and a response comes out. Lets cover each of these pieces in detail now.

A note about Paste

While you setup your toy Pylons application, you probably wondered why Pylons seems to use a command line tool called “paster” instead of something obvious like “pylons”. Paster is actually a part of the Paste set of tools that Pylons uses.

Paste is used to build web application frameworks - not web applications - but web application frameworks like Pylons. Everytime you use “paster”, that’s Paste being called. Everytime you access the HTTP request and response objects - that’s WebOb - a descendant of Paste’s HTTP wrapper code. Pylons uses Paste extensively for configuration management, testing, basic HTTP handling with WebOb. You would do well to at least skim over the Paste documentation to see what is available in paste.

Pylons MVC

Pylons, like Django any any reasonably sane webframework (or GUI toolkit for that matter) uses the model-view-controller design pattern.

In Pylons this maps to:

Component	Implementation
Model	SQLAlchemy (or any other database toolkit you prefer)
View	Mako (or any templating language you prefer)
Controller	Plain Python code

To reiterate - Pylons is about letting you - the application developer decide on the particular tradeoffs you’re willing to make. If using a templating language more similar to Django is better for your web designers, then switch go Jinja2. If you don’t really want to deal with SQLAlchemy - you can use SQLAlchemy or raw SQL if you prefer.

Pylons provides some tools to help you hook these pieces together in a rational way.

Routes is a library that maps URLs to classes. This is your basic mechanism for dispatching methods whenever your webserver is hit. Routes provides similar functionality to what Django’s URL dispatcher provides.

Webhelpers is the defacto standard library for Pylons. It contains commonly used functions for the web like flashing status messages to users, date conversion functions, HTML tag generation, pagination functions, text processing - the list goes on.

Pylons also provides infrastructure so that you can manipulate things that are particular to web applications including:

- WSGI middleware to add functionality to your application with minimal intrusion into your existing codebase.
- A robust testing framework including a shockingly good debugger you can use through the web.
- Helpers to enable REST-ful API development so you can expose your application as a programmatic interface.

Now let’s wrap up the hockey roster up in a web application. We’ll target a couple features:

- form handling and validation to add new players through the web
- login and authentication to make sure not anybody can edit our lists

- add a JSON/REST api so that we can modify data from other tools

In the process, we'll use the interactive debugger from both command line and through the web to directly observe and interact with the state of the running application.

An interlude into Java's memory model

A note about reloading - sometimes if you're doing development with Pylons on Jython, Java will through an Out-Of-Memory error like this

```
java.lang.OutOfMemoryError: PermGen space
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:620)
```

Java keeps track of class definitions in something called the Permanent Generation heap space. This is a problem for Pylons when the HTTP threads are restarted and your classes are reloaded. The old class definitions don't go away - they never get garbage collected.. Since Jython is dynamically creating Java classes behind the scenes, each time your development server restarts - you're potentially getting hundreds of new classes loaded into the JVM.

Repeat this several times and it doesn't take long until your JVM has run out of permgen space and it keels over and dies.

To modify the permgen heap size, you'll need to instruct Java using some extended command line options. To set the heap to 128M, you'll need to use "-XX:MaxPermSize=128M".

To get this behavior by default for Jython, you'll want to edit your Jython startup script in JYTHON_HOME/bin/jython (or jython.bat) by editing the line that reads

```
set _JAVA_OPTS=
```

to be

```
set _JAVA_OPTS=-XX:MaxPermSize=128M
```

This shouldn't be a problem in production environments where you're not generating new class definitions during runtime, but it can be quite frustrating during development.

Invoking the Pylons shell

Yes, I'm going to start with testing right away because it will provide you with a way to explore the Pylons application in an interactive way.

Pylons gives you an interactive shell much like Django's. You can start it up with the following commands.

```
RosterTool > jython setup.py egg_info
RosterTool > paster shell test.ini
```

This will yield a nice interactive shell you can start playing with right away. Now lets take a look at those request and response objects in our toy application.

```
RosterTool > paster shell test.ini

Pylons Interactive Shell
Jython 2.5.0 (Release_2_5_0:6476, Jun 16 2009, 13:33:26)
[OpenJDK Server VM (Sun Microsystems Inc.)]

All objects from rostertool.lib.base are available
```

```
Additional Objects:
mapper      - Routes mapper object
wsgiapp      - This project's WSGI App instance
app          - paste.fixture wrapped around wsgiapp

>>> resp = app.get('/roster/index')
>>> resp
<Response 200 OK 'Hello World'>
>>> resp.req
<Request at 0x43 GET http://localhost/roster/index>
```

Pylons lets you actually run requests against the application and play with the resulting response. Even for something as 'simple' as the HTTP request and response,, Pylons uses a library to provide convenience methods and attributes to make your development life easier. In this case - it's WebOb - a derivative of Paste's older HTTP wrapper code.

The request and the response objects both have literally dozens of attributes and methods that are provided by the framework. You' almost certainly going to benefit if you take time to browse through WebOb's documentation.

Here's four attributes you really have to know to make sense of the request object. The best thing to do is to try playing with the request object in the shell.

request.GET GET is a special dictionary of the variables that were passed in the URL. Pylons automatically converts URL arguments that appear multiple times into discrete key value pairs.

```
>>> resp = app.get('/roster/index?foo=bar&x=42&x=50')
>>> resp.req.GET
UnicodeMultiDict([('foo', u'bar'), ('x', u'42'), ('x', u'50')])
>>> req.GET['x']
u'50'
>>> req.GET.getall('x')
[u'42', u'50']
```

Note how you can get either the last value or the list of values depending on how you choose to fetch values from the dictionary. This can cause subtle bugs if you're not paying attention.

request.POST POST is similar to GET, but appropriately - it only returns the variables that were sent up during an HTTP POST submission.

request.params Pylons merges all the GET and POST data into a single MultiValueDict. In almost all cases, this is the one attribute that you really want to use to get the data that the user sent to the server.

request.headers This dictionary provides all the HTTP headers that the client sent to the server.

Context Variables and Application Globals

Most webframeworks provide a request scoped variable to act as a bag of values. Pylons is no exception - whenever you create a new controller with paste - it will automatically import an attribute 'c' which is the context variable.

This is one aspect of Pylons which I've found to be frustrating. The 'c' attribute is code generated as an import when you instruct paste to build you a new controller. The 'c' value is *not* an attribute of your controller - Pylons has special global threadsafe variables - this is just one of them. You can store variables that you want to exist for the duration of the request in the context. These values won't persist after the request/response cycle has completed so don't confuse this with the session variable.

The other global variable you'll end up using a lot is pylons.session. This is where you'll store variables that need to persist over the course of several request/response cycles. You can treat this variable as aa special dictionary - just use standard Jython dictionary syntax and Pylons will handle the rest.

Routes

Routes is much like Django’s URL dispatcher. It provides a mechanism for you to map URLs to controllers classes and methods to invoke.

Generally, I find that Routes makes a tradeoff of less URL matching expressiveness in exchange for simpler reasoning about which URLs are directed to a particular controller and method. Routes doesn’t support regular expressions, just simple variable substitution.

A typical route will look something like this

```
map.connect('/{mycontroller}/{someaction}/{var1}/{var2}')
```

The above route would find the controller called “Mycontroller” (note the casing of the class) and invoke the “someaction” method on that object. Variables var1 and var2 would be passed in as arguments.

The connect() method of the map object will also take in optional arguments to fill in default values for URLs that do not have enough URL encoded data in them to properly invoke a method with the minimum required number of arguments. The front page is an example of this - let’s try connecting the frontpage to the Roster.index method.

Edit rostertool/config/routing.py so that there are 3 lines after #CUSTOM_ROUTES_HERE that should read something like this

```
map.connect('/', controller='roster', action='index')
map.connect('/{action}/{id}/', controller='roster')
map.connect('/add_player/', controller='roster', action='add_player')
```

While this *looks* like it should work, you can try running “paster serve”, it won’t.

Pylons always tries to serve static content before searching for controllers and methods to invoke. You’ll need to go to RosterTool/rostertool/public and delete the ‘index.html’ file that paster installed when you first created your application.

Load <http://localhost:5000/> again in your browser - the default index.html should be gone and you should now get your welcome page.

Controllers and Templates

Leveraging off of the Table model we defined in chapter 12, let’s create the hockey roster, but this time using the postgresql database. I’ll assume that you have a postgresql installation running that allows you create new databases.

```
>>> from sqlalchemy import create_engine
>>> from sqlalchemy.schema import Sequence
>>> db = create_engine('postgresql+zxjdbc://myuser:mypass@localhost:5432/mydb')
>>> connection = db.connect()
>>> metadata = MetaData()
>>> player = Table('player', metadata,
...     Column('id', Integer, primary_key=True),
...     Column('first', String(50)),
...     Column('last', String(50)),
...     Column('position', String(30)))
>>> metadata.create_all(engine)
```

Now let’s wire the data up to the controllers, display some data and get basic form handling working. We’re going to create a basic CRUD (create, read, update, delete) interface to the sqlalchemy model. Because of space constraints, this HTML is going to be very basic, but you’ll get a taste of how things fit together.

Paste doesn’t just generate a stub for your controller - it will also code generate an empty functional test case in rostertool/tests/functional/ as test_roster.py. We’ll visit testing shortly.

Controllers are really where the action occurs in Pylons. This is where your application will take data from the database and prepare it for a template to render it as HTML. Let's put the list of all players on the front page of the site. We'll implement a template to render the list of all players. Then, we'll implement a method in the controller to override the `index()` method of `Roster` to use `SQLAlchemy` to load the records from disk and send them to the template.

Along the way, we'll touch on template inheritance so that you can see how you can save keystrokes by subclassing your templates in `Mako`.

First, let's create two templates, `base.html` and `list_players.html` in the `rosterool/templates` directory.

`base.html`

```
<html>
  <body>
    <div class="header">
      ${self.header()}
    </div>

    ${self.body()}
  </body>
</html>

<%def name="header()">
  <h1>${c.page_title}</h1>
  <% messages = h.flash.pop_messages() %>
  % if messages:
  <ul id="flash-messages">
    % for message in messages:
    <li>${message}</li>
    % endfor
  </ul>
  % endif
</%def>
```

`list_players.html`

```
<%inherit file="base.html" />
<table border="1">
  <tr>
    <th>Position</th><th>Last name</th><th>First name</th><th>Edit</th>
  </tr>
  % for player in c.players:
  ${makerow(player)}
  % endfor
</table>

<h2>Add a new player</h2>
${h.form(h.url_for(controller='roster', action='add_player'), method='POST')}
  ${h.text('first', 'First Name')} <br />
  ${h.text('last', 'Last Name')} <br />
  ${h.text('position', 'Position')} <br />
  ${h.submit('add_player', "Add Player")}
${h.end_form()}

<%def name="makerow(row)">
<tr>
  <td>${row.position}</td>\
  <td>${row.last}</td>\
  <td>${row.first}</td>>
```

```
<td><a href="${h.url_for(controller='roster', action='edit_player', id=row.id)}">
↪Edit</a></td>\
</tr>
</%def>
```

There's quite a bit going on here. The base template lets Mako define a boilerplate set of HTML that all pages can reuse. Each section is defined with a `<%def name="block()">` section and the blocks are overloaded in the subclassed templates. In effect - Mako lets your page templates look like objects with methods that can render subsections of your pages.

The `list_players.html` template has content that is immediately substituted into the `self.body()` method of the base template. The first part of our body uses our magic context variable 'c'. Here - we're iterating over each of the players in the database and rendering them into a table as a row. Note here that we can use the Mako method syntax to create a method called 'makerow' and invoke it directly within our template.

#XX: Aside for Mako Mako provides a rich set of functions for templating. I'm only going to use the most basic parts of Mako - inheritance, variable substitution and loop iteration to get the toy application working. I strongly suggest you dive into the Mako documentation to discover features and get a better understanding of how to use the template library. ##

Next, we add in a small form to create new players. The trick here is to see that the form is being generated programmatically by helper functions. Pylons automatically imports `YOURPROJECT/lib/helpers` (in our case - `roster-tool.lib.helpers`) as the 'h' variable in your template. The helpers module typically imports functions from parts of Pylons or a dependant library to allow access to those features from anywhere in the application. Although this seems like a violation of 'separation of concerns' - look at the template and see what it buys us? We get fully decoupled URLs from the particular controller and method that need to be invoked. The template uses a special routes function "url_for" to compute the URL that would have been mapped for a particular controller and method. The last part of our `list_players.html` file contains code to display alert messages.

Let's take a look at our `rosterool.lib.helpers` module now

```
from routes import url_for
from webhelpers.html.tags import *
from webhelpers.pylonslib import Flash as _Flash

# Send alert messages back to the user
flash = _Flash()
```

Here, we're importing the `url_for` function from routes to do our URL reversal computations. We import HTML tag generators from the main `html.tags` helper modules and we import `Flash` to provide alert messages for our pages. I'll show you how flash messages are used when we cover the controller code in more detail in the next couple of pages.

Now, create a controller with paste (you've already done this if you were impatient at the beginning of the chapter)

```
$ cd ROSTERTOOL/rosterool
$ paster controller roster
```

RosterController should get a method very short method that reads

```
def index(self):
    session = Session()
    c.page_title = 'Player List'
    c.players = session.query(Player).all()
    return render('list_players.html')
```

This code is fairly straight forward, we are simply using a SQLAlchemy session to load all the Player objects from disk and assigning to the special context variable 'c'. Pylons is then instructed to render the `list_player.html` file. Let's take a look at that file now:

The context should be your default place to place values you want to pass to other parts of the application. Note that Pylons will automatically bind in URL values to the context so while you can grab the form values from `self.form_result`, you can also grab raw URL values from the context.

You should be able run the debug webserver now and you can get to the front page to load an empty list of players. Start up your debug webserver as you did at the beginning of this chapter and go to <http://localhost:5000/> to see the page load with your list of players (currently an empty list).

Now we need to get to the meaty part where we can start create, edit and delete players. We'll make sure that the inputs are at least minimally validated, errors are displayed to the user and that alert messages are properly populated.

First, we need a page that shows just a single player and provides buttons for edit and delete.

```
<%inherit file="base.html" />

<h2>Edit player</h2>
${h.form(h.url_for(controller='roster', action='save_player', id=c.player.id), method=
    'POST')}
    ${h.hidden('id', c.player.id)} <br />
    ${h.text('first', c.player.first)} <br />
    ${h.text('last', c.player.last)} <br />
    ${h.text('position', c.player.position)} <br />
    ${h.submit('save_player', "Save Player")}
${h.end_form()}

${h.form(h.url_for(controller='roster', action='delete_player', id=c.player.id),
    method='POST')}
    ${h.hidden('id', c.player.id)} <br />
    ${h.hidden('first', c.player.first)} <br />
    ${h.hidden('last', c.player.last)} <br />
    ${h.hidden('position', c.player.position)} <br />
    ${h.submit('delete_player', "Delete Player")}
${h.end_form() }
```

This template assumes that there is a 'player' value assigned to the context and not suprisingly - it's going to be a full blown instance of the Player object that we first saw in chapter 12. The helper functions let us define our HTML form using webhelper tag generation functions. This means you won't have to worry about escaping characters or remember the particular details of the HTML attributes. The helper.tag functions will do sensible things by default.

I've setup the edit and delete forms to point to different URLs. You might want to 'conserve' URLs but having discrete URLs for each action has advantages - especially for debugging. You can trivially view which URLs are being hit on a webserver by reading log files. Seeing the same kind of behavior if the URLs are the same, but the behavior is dictated by some form value - well that's a whole lot harder to debug. It's also a lot harder to setup in your controllers because you need to dispatch the behavior on a per method level. Why not just have separate methods for separate behaviour - everybody will thank you for it when they need to debug your code in the future.

Before we create our controller methods for create, edit and delete - we'll create a formencode schema to provide basic validation. Again - Pylons doesn't provide validation behaviour - it just leverages another library to do so. In `rosterool/controllers/roster.py`

```
class PlayerForm(formencode.Schema):
    # You need the next line to drop the submit button values
    allow_extra_fields=True

    first = formencode.validators.String(not_empty=True)
    last = formencode.validators.String(not_empty=True)
    position = formencode.validators.String(not_empty=True)
```

This simply provides basic string verification on our inputs. Note how this doesn't provide any hint as to what the

HTML form looks like - or that it's HTML at all. FormEncode can validate arbitrary Python dictionaries and return errors about them.

I'm just going to show you the add method, and the edit_player methods. You should try to implement the save_player and delete_player methods to make sure you understand what's going on here.

```
from pylons.decorators import validate
from rostertool.model import Session, Player

@validate(schema=PlayerForm(), form='index', post_only=False, on_get=True)
def add_player(self):
    first = self.form_result['first']
    last = self.form_result['last']
    position = self.form_result['position']
    session = Session()
    if session.query(Player).filter_by(first=first, last=last).count() > 0:
        h.flash("Player already exists!")
        return h.redirect_to(controller='roster')
    player = Player(first, last, position)
    session.add(player)
    session.commit()
    return h.redirect_to(controller='roster', action='index')

def edit_player(self, id):
    session = Session()
    player = session.query(Player).filter_by(id=id).one()
    c.player = player
    return render('edit_player.html')
```

A couple of notes here. edit_player is passed in the 'id' attribute directly by Routes. In the edit_player method - 'player' is assigned to the context, but the context is never explicitly passed into the template renderer. Pylons is going to automatically take the attributes bound to the context and write them into template and render the HTML output.

With the add_player method, I'm using the validate decorator to enforce the inputs against the PlayerForm. In the case of error, the form attribute of the decorator is used to load an action against the current controller. In this case - 'index' - so the front page loads.

The SQLAlchemy code should be familiar to you if you have already gone through chapter 12. The last line of the add_player method is a redirect to prevent problems with hitting reload in the browser. Once all data manipulation has occurred - the server redirects the client to a results page. In the case that a user hits reload on the result page - no data will be mutated.

Here's the signatures of the remaining methods you'll need to implement to make things work:

- save_player(self):
- delete_player(self):

If you get stuck, you can always consult the working sample code on the book website.

Adding in a JSON API

JSON integration into Pylons is very straight forward. The steps are roughly the same as adding controller methods for plain HTML views. You invoke paste, paste then generates your controller stubs and test stubs, you add in some routes to wire controllers to URLs and then you just fill in the controller code.

```
$ cd ROSTERTOOL_HOME/rostertool
$ paster controller api
```

Pylons provides a special `@jsonify` decorator which will automatically convert Python primitive types into JSON objects. It will *not* convert the POST data into an object though - that's your responsibility. Adding a simple read interface into the player list requires only adding a single method to your ApiController

```
@jsonify
def players(self):
    session = Session()
    players = [{'first': p.first,
                'last': p.last,
                'position': p.position,
                'id': p.id} for p in session.query(Player).all()]
    return players
```

adding a hook so that people can POST data to your server in JSON format to create new player is almost as easy

```
import simplejson as json

@jsonify
def add_player(self):
    obj = json.loads(request.body)
    schema = PlayerForm()
    try:
        form_result = schema.to_python(obj)
    except formencode.Invalid, error:
        response.content_type = 'text/plain'
        return 'Invalid: '+unicode(error)
    else:
        session = Session()
        first, last, position = obj['first'], obj['last'], obj['position']
        if session.query(Player).filter_by(last=last, first=first,
                                           position=position).count() == 0:
            session.add(Player(first, last, position))
            session.commit()
            return {'result': 'OK'}
        else:
            return {'result': 'fail', 'msg': 'Player already exists'}
```

Unit testing, Functional Testing and Logging

One of my favourite features in Pylons is its rich set of testing, and debugging. It even manages to take social networking, turn it upside down and make it into a debugger feature. We'll get to that shortly.

The first step to knowing how to test code in pylons is to familiarize yourself with the nose testing framework. nose makes testing simple by getting out of your way. There are no classes to subclass, just start writing functions that start with the word 'test' and nose will run them. Write a class that has "Test" prefixed in the name and nose will treat it as a suite of tests running each method that starts with 'test'. For each test method, nose will execute the `setup()` method just prior to executing your test and nose will execute the `teardown()` method after your test case.

Best of all, nose will automatically hunt down anything that looks like a test and will run it for you. There is no complicated chain of testcases you need to organize in a tree. The computer will do that for you.

Let's take a look at your first testcase - we'll just instrument the model, in this case - SQLAlchemy. Since the model layer has no dependency on Pylons - this effectively - a test of just SQLAlchemy.

In `ROSTERTOOL_HOME/rosterool/tests`, create a module called "test_models.py" with the following content

```
from rosterool.model import Player, Session, engine
```

```

class TestModels(object):

    def setup(self):
        self.cleanup()

    def teardown(self):
        self.cleanup()

    def cleanup(self):
        session = Session()
        for player in session.query(Player):
            session.delete(player)
        session.commit()

    def test_create_player(self):
        session = Session()
        player1 = Player('Josh', 'Juneau', 'forward')
        player2 = Player('Jim', 'Baker', 'forward')
        session.add(player1)
        session.add(player2)

        # But 2 are in the session, but not in the database
        assert 2 == session.query(Player).count()
        assert 0 == engine.execute("select count(id) from player").fetchone()[0]
        session.commit()

        # Check that 2 records are all in the database
        assert 2 == session.query(Player).count()
        assert 2 == engine.execute("select count(id) from player").fetchone()[0]

```

Before we can run the tests, we'll need to edit the model module a little so that the models know to lookup the connection URL from Pylons's configuration file. In your test.ini, add a line setting the sqlalchemy.url setting to point to your database in the [app:main] section.

You should have a line that looks something like this

```

[app:main]
use = config:development.ini
sqlalchemy.url = postgresql+zxjdbc://username:password@localhost:5432/mydb

```

Now edit the model file so that the create_engine call uses that configuration. This is as simple as importing config from pylons and doing a dictionary lookup. The two lines you want are

```

from pylons import config
engine = create_engine(config['sqlalchemy.url'])

```

and that's it. Your model will now lookup your database connection string from Pylons. Even better - nose will know how to use that configuration as well.

From the command line, you can run the tests from ROSTERTOOL_HOME like this now

```

ROSTERTOOL_HOME $ nosetests rostertool/tests/test_models.py
.
-----
Ran 1 test in 0.502s

```

Perfect! To capture stdout and get verbose output, you can choose to use the '-sv' option. Nose has it's own active community of developers. You can get plugins to do coverage analysis and performance profiling with some of the

plugins. Use “nosetests –help” for a list of the options available for a complete list.

Due to the nature of Pylons and it’s pathologically decoupled design, writing small unit tests to test each little piece of code is very easy. Feel free to assemble your tests any which way you want. Just want to have a bunch of test funtoins? Great! If you need to have setup and teardown and writing a test class makes sense - then do so.

Testing with nose is a joy - you aren’t forced to fit into any particular structure with respect to where you tests must go so that they will be executed. You can organize your tests in a way that makes the most sense to *you*.

That covers basic unit testing, but suppose we want to test the JSON interface to our hockey roster. We really want to be able to invoke GET and POST on the URLs to make sure that URL routing is working as we expect. We want to make sure that the content-type is properly set to ‘application/x-json’. In other words - we want to have a proper functional test - a test that’s not as fine grained as a unit test.

The prior exposure to the ‘app’ object when we ran the paste shell should give you a rough idea of what is required. In Pylons, you can instrument your application code by using a TestController. Lucky for you, Pylons has already create one for you in your <app>/tests directory. Just import it, subclass it and you can start using the ‘app’ object just like you did inside of the shell.

Lets take a look at a functional test in detail now. Here’s a sample you cna save into roster-tool/tests/functional/test_api.py

```
from rostertool.tests import *
import simplejson as json
from rostertool.model.models import Session, Player

class TestApiController(TestController):
    # Note that we're using subclasses of unittest.TestCase so we need
    # to be careful with setup/teardown camelcasing unlike nose's
    # default behavior

    def setUp(self):
        session = Session()
        for player in session.query(Player):
            session.delete(player)
        session.commit()

    def test_add_player(self):
        data = json.dumps({'first': 'Victor',
                           'last': 'Ng',
                           'position': 'Goalie'})
        # Note that the content-type is set in the headers to make
        # sure that paste.test doesn't URL encode our data
        response = self.app.post(url(controller='api', action='add_player'),
                                params=data,
                                headers={'content-type': 'application/x-json'})
        obj = json.loads(response.body)
        assert obj['result'] == 'OK'

        # Do it again and fail
        response = self.app.post(url(controller='api', action='add_player'),
                                params=data,
                                headers={'content-type': 'application/x-json'})
        obj = json.loads(response.body)
        assert obj['result'] <> 'OK'
```

There’s a minor detail which you can easily miss when you’re using the TestController as your superclass. First off, TestController is a descendant of unittest.TestCase frmo the standard python unit test library. Nose will not run ‘setup’ and ‘teardown’ methods on TestCase subclasses. Instead, you’ll have to use the camel case names that TestCase uses.

Reading through the testcase should show you how much detail you can be exposed. All your headers are exposed, the response content is exposed - indeed the HTTP response is completely exposed as an object for you to inspect and verify.

So great, now we can run small unit tests, bigger functional tests - lets's take a look at the debugging facilities provided through the web.

Consider what happens with most web application stacks when an error occurs. Maybe you get a stack trace, maybe you don't. If you're lucky, you can see the local variables at each stack frame like Django does. Usually though, you're out of luck if you want to interact with the live application as the error is occurring.

Eventually, you may locate the part of the stack trace that triggered the error, but the only way of sharing that information is through either the mailing lists or by doing a formal patch against source control. Let's take a look at an example of that.

We're going to startup our application in development mode. We're also going to intentionally break some code in the controller to see the stack trace. But first, we'll need to put some data into our app. run

Add a sqlalchemy.url configuration line as you did in the test.ini configuration, and let's startup the application in development mode. We're going to have the server run so that any code changes on the file system are automatically detected and the code is reloaded

```
$ paster serve development.ini --reload
```

We'll add a single player "John Doe" as a center, and save the record

```
# TODO: insert screenshot of the add user interface
```

Now let's intentionally break some code to trigger the debugger. Modify the RosterController's index method and edit the call that loads the list of players. We'll use the web session instead of the database session to try loading the Player objects.

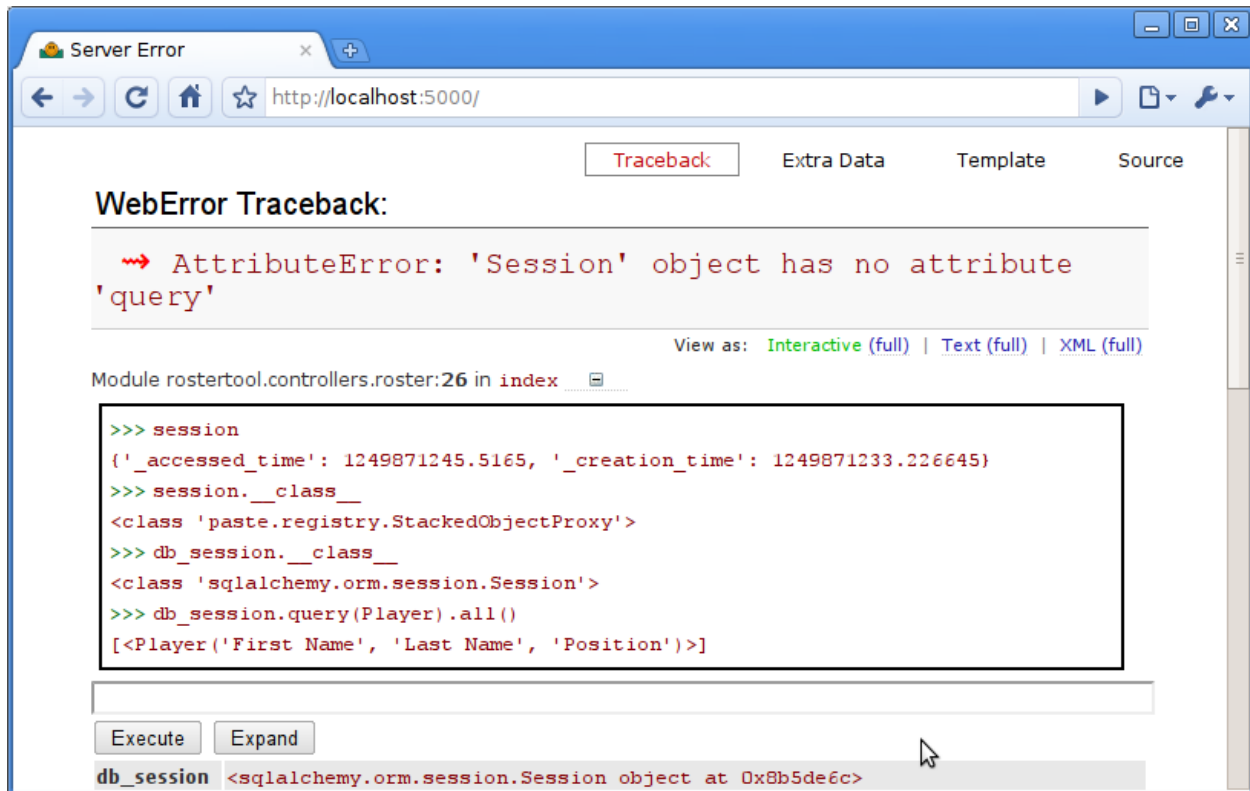
```
def index(self):
    db_session = Session()
    c.page_title = 'Player List'
    c.players = session.query(Player).all()
    return render('list_players.html')
```

Load <http://localhost:5000/> to see the error page. You should see something like this

```
# XXX: insert screen capture of the error page listing
'AttributeError: Session object has no attribute 'not_a_method'
```

There's a lot of information that Pylons throws back at you. Along the top of the screen, you'll see 4 tabs: Traceback, Extra Data, Template and Source - Pylons will have put you in the Traceback tab by default to start with. If you look at the error, you'll see the exact line number in the source file that the error occurred in. What's special about Pylons traceback tab is that this is actually a fully interactive session.

You can select the "+" signs to expand each stackframe and a text input along with some local variables on that frame will be revealed. That text input is an interface into your server process. You can type virtually any python command into it, hit enter and you will get back live results. From here, we can see that we should have used the 'db_session' and not the 'session' variable.



This is pretty fantastic. If you click on the ‘view’ link, you can even jump to the full source listing of the Jython module that caused the error. One bug in Pylons at the time of writing is that sometimes, the hyper link is malformed. So while the traceback will correctly list the line number that the error occurred at, the source listing may go to the wrong line.

The Pylons developers have also embedded an interface into search engines to see if your error has been previously reported. If you scroll down to the bottom of your traceback page, you’ll see another tab control with a ‘Search Mail Lists’ option. Here, Pylons will automatically extract the exception message and provide you an interface so you can literally search all the mailing lists that are relevant to your particular Pylons installation.

If you can’t find your error on the mailing lists, you can go to the next tab “Post traceback” and submit your stacktrace to a webservice on PylonsHQ.com so that you can try to debug your problems online with other collaborators. Combining unit tests, functional tests, and the myriad of debugging options afforded to you in the web debugger - Pylons makes the debugging experience as painless as possible.

Deployment into a servlet container

Deploying your pylons application into a servlet container is very straight forward. Just install snakefight from PyPI using `easy_install` and you can start building WAR files.

```
$ easy_install snakefight
...snakefight will download and install here ...
$ jython setup.py bdist_war --paste-config test.ini
```

By default, snakefight will bundle a complete instance of your Jython installation into the WAR file. What it doesn’t include is any JAR files that your application depends on. For our small example, this is just the postgresql JDBC driver. You can use the `--include-jars` options and provide a comma separated list of JAR files.

```
$ jython setup.py bdist_war \
    --include-jars=postgresql-8.3-604.jdbc4.jar \
    --paste-config=test.ini
```

The final WAR file will be located under the dist directory. It will contain your postgresql JDBC driver, a complete installation of Jython including anything located in site-packages and your application. Your war file should deploy without any issues into any standards compliant servlet container.

Conclusion

We've only scratched the surface of what's possible with Pylons, but I hope you've gotten a taste of what is possible with Pylons. Pylons uses a large number of packages so you will need to spend more time getting over the initial learning curve, but the dividend is the ability to pick and choose the libraries that best solve your particular problems.

Chapter 16: GUI Applications

The C implementation of Python comes with Tcl/Tk for writing Graphical User Interfaces (GUIs). On Jython, the GUI toolkit that you get automatically is Swing, which comes with the Java Platform. Like CPython, there are other toolkits available for writing GUIs in Jython. Since Swing is available on any modern Java installation, we will focus on the use of Swing GUIs in this chapter.

Swing is a large subject, and can't really be covered in a single chapter. In fact there are entire books devoted to the subject. I will provide some introduction to Swing, but only enough to describe the use of Swing from Jython. For in depth coverage of Swing, one of the many books or web tutorials should be used. [FJW: some suggested books/tutorials?].

Using Swing from Jython has a number of advantages over the use of Swing in Java. For example, bean properties are less verbose in Jython, and binding actions in Jython is much less verbose (in Java anonymous classes are typically used, in Jython a function can be passed).

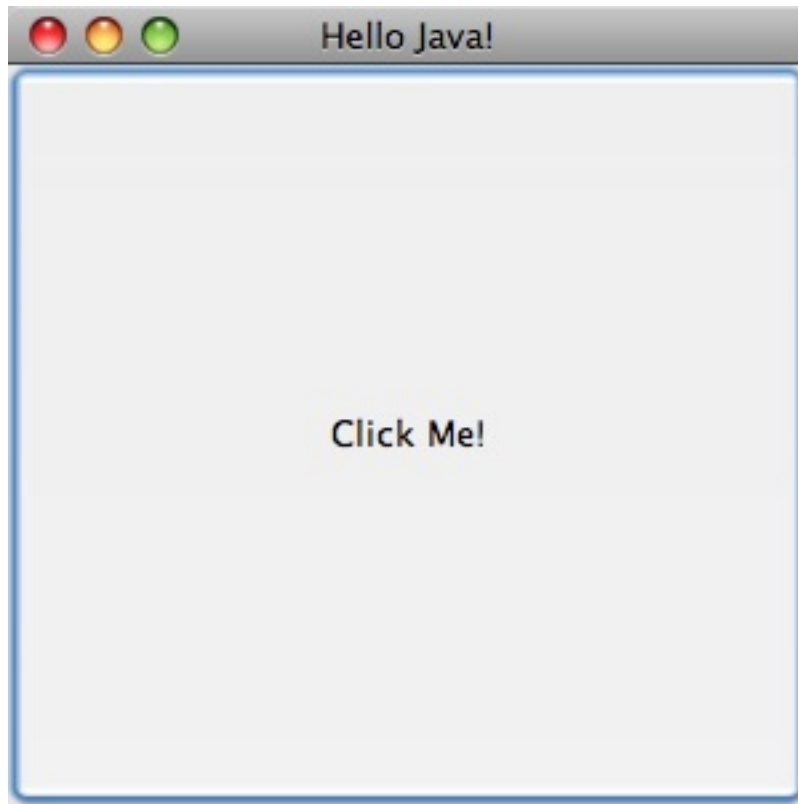
Let's start with an simple Swing application in Java, then we will look at the same application in Jython.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

public class HelloWorld {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello Java!");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        JButton button = new JButton("Click Me!");
        button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    System.out.println("Clicked!");
                }
            }
        );
        frame.add(button);
        frame.setVisible(true);
    }
}
```

This simple application draws a JFrame that is completely filled with a JButton. When the button is pressed, “Clicked!” prints out on the command line.



Now let’s see what this program looks like in Jython

```
from javax.swing import JButton, JFrame

frame = JFrame('Hello, Jython!',
               defaultCloseOperation = JFrame.EXIT_ON_CLOSE,
               size = (300, 300)
               )

def change_text(event):
    print 'Clicked!'

button = JButton('Click Me!', actionPerformed=change_text)
frame.add(button)
frame.visible = True
```

Except for the title, the application produces the same JFrame with JButton, outputting “Clicked!” when the button is clicked.



Let's go through the Java and the Jython examples line by line to get a feel for the differences between writing Swing apps in Jython and Java. First the import statements:

In Java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
```

In Jython

```
from javax.swing import JButton, JFrame
```

In Jython, it is always best to have explicit imports by name, instead of using

```
from javax.swing import *
```

for the reasons covered in Chapter 7. Note that we did not need to import `ActionEvent` or `ActionListener`, since Jython's dynamic typing allowed us to avoid mentioning these classes in our code.

Next, we have some code that creates a `JFrame`, and then sets a couple of bean properties.

In Java :: `JFrame frame = new JFrame("Hello Java!"); frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); frame.setSize(300, 300);`

In Jython

```
frame = JFrame('Hello, Jython!',
               defaultCloseOperation = JFrame.EXIT_ON_CLOSE,
               size = (300, 300)
               )
```

In Java a new JFrame is created, then the bean properties `defaultCloseOperation` and `size` are set. In Jython, we are able to add the bean property setters right in the call to the constructor. This shortcut is covered in detail in chapter [FJW]? already. Still, it will bare some repeating here, since bean properties are so important in the Swing libraries. In short, if you have a getters and setters of the form `getFoo/setFoo`, you can treat them as properties of the object with the name “foo”. So instead of `x.getFoo()` you can use `x.foo`. Instead of `x.setFoo(bar)` you can use `x.foo = bar`. If you take a look at any Swing app above a reasonable size, you are likely to see large blocks of setters like:

```
JTextArea t = JTextArea();
t.setText(message)
t.setEditable(false)
t.setWrapStyleWord(true)
t.setLineWrap(true)
t.setAlignmentX(Component.LEFT_ALIGNMENT)
t.setSize(300, 1)
```

which, in my opinion, look better in the idiomatic Jython property setting style:

```
t = JTextArea()
t.text = message
t.editable = False
t.wrapStyleWord = True
t.lineWrap = True
t.alignmentX = Component.LEFT_ALIGNMENT
t.size = (300, 1)
```

Or rolled into the constructor:

```
t = JTextArea(text = message,
              editable = False,
              wrapStyleWord = True,
              lineWrap = True,
              alignmentX = Component.LEFT_ALIGNMENT,
              size = (300, 1)
            )
```

One thing to watch out for when you use properties rolled into the constructor, is that you don’t know the order in which the setters will be called. Generally this is not a problem, as the bean properties are not usually order dependant. The big exception to this is `setVisible()`, you probably want to set the visible property outside of the constructor to avoid any strangeness while the properties are being set. Going back to our short example, the next block of code creates a JButton, and binds the button to an action that prints out “Clicked!”.

In Java

```
JButton button = new JButton("Click Me!");
button.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Clicked!");
        }
    }
);
frame.add(button);
```

In Jython

```
def change_text(event):
    print 'Clicked!'

button = JButton('Click Me!', actionPerformed=change_text)
frame.add(button)
```

I think Jython’s method is particularly nice here when compared to Java. Here we can pass a first class function “change_text” directly to the JButton in it’s constructor, in place of the more cumbersome Java “addActionListener” method where we need to create an anonymous ActionListener class and define it’s actionPerformed method with all of the ceremony of the static type declarations. This is one case where Jython’s readability really stands out. Finally, in both examples we set the visible property to True. Again, although we could have set this property in the frame constructor, the visible property is one of those rare order-dependant properties that we want to set at the right time (in this case, last).

In Java

```
frame.setVisible(true);
```

In Jython

```
frame.visible = True
```

Now that we have looked at a simple example, it makes sense to see what a medium sized app might look like in Jython. Since Twitter apps have become the “Hello World” of GUI applications these days, we will go with the trend. The following application gives the user a log in prompt. When the user successfully logs in the most recent tweets in their timeline are displayed. Here is the code:

```
import twitter
import re

from javax.swing import (BoxLayout, ImageIcon, JButton, JFrame, JPanel,
    JPasswordField, JLabel, JTextArea, JTextField, JScrollPane,
    SwingConstants, WindowConstants)
from java.awt import Component, GridLayout
from java.net import URL
from java.lang import Runnable

class JyTwitter(object):
    def __init__(self):
        self.frame = JFrame("Jython Twitter")
        self.frame.defaultCloseOperation = WindowConstants.EXIT_ON_CLOSE

        self.loginPanel = JPanel(GridLayout(0,2))
        self.frame.add(self.loginPanel)

        self.usernameField = JTextField('',15)
        self.loginPanel.add(JLabel("username:", SwingConstants.RIGHT))
        self.loginPanel.add(self.usernameField)

        self.passwordField = JPasswordField('', 15)
        self.loginPanel.add(JLabel("password:", SwingConstants.RIGHT))
        self.loginPanel.add(self.passwordField)

        self.loginButton = JButton('Log in',actionPerformed=self.login)
        self.loginPanel.add(self.loginButton)

        self.message = JLabel("Please Log in")
```

```
self.loginPanel.add(self.message)

self.frame.pack()
self.frame.visible = True

def login(self, event):
    self.message.text = "Attempting to Log in..."
    self.frame.show()
    username = self.usernameField.text
    try:
        self.api = twitter.Api(username, self.passwordField.text)
        self.timeline(username)
        self.loginPanel.visible = False
        self.message.text = "Logged in"
    except:
        self.message.text = "Log in failed."
        raise
    self.frame.size = 400, 800
    self.frame.show()

def timeline(self, username):
    timeline = self.api.GetFriendsTimeline(username)
    self.resultPanel = JPanel()
    self.resultPanel.layout = BoxLayout(self.resultPanel, BoxLayout.Y_AXIS)
    for s in timeline:
        self.showTweet(s)

    scrollpane = JScrollPane(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
                             JScrollPane.HORIZONTAL_SCROLLBAR_NEVER)
    scrollpane.preferredSize = 400, 800
    scrollpane.viewport.view = self.resultPanel

    self.frame.add(scrollpane)

def showTweet(self, status):
    user = status.user
    p = JPanel()

    # image grabbing seems very expensive, good place for a callback?
    p.add(JLabel(ImageIcon(URL(user.profile_image_url))))

    p.add(JTextArea(text = status.text,
                    editable = False,
                    wrapStyleWord = True,
                    lineWrap = True,
                    alignmentX = Component.LEFT_ALIGNMENT,
                    size = (300, 1)
                    ))
    self.resultPanel.add(p)

if __name__ == '__main__':
    JyTwitter()
```

This code depends on the python-twitter package. This package can be found on the Python package index (PyPi). If you have easy_install (see chapter ? for instructions on easy_install) then you can install python-twitter like this:

```
jython easy_install python-twitter
```


This will automatically install python-twitter's dependency: simplejson. Now you should be able to run the application. You should see the following login prompt:



If you put in the wrong password, you should see:



And finally, once you have successfully logged in, you should see something like this:



The constructor creates the outer frame, imaginatively called `self.frame`. We set `defaultCloseOperation` so that the app will terminate if the user closes the main window. We then create a `loginPanel` that holds the text fields for the user to enter username and password, and create a login button that will call the `self.login` method when clicked. We then put a “Please log in” label in and make the frame visible.

```
def __init__(self):
    self.frame = JFrame("Jython Twitter")
    self.frame.defaultCloseOperation = WindowConstants.EXIT_ON_CLOSE

    self.loginPanel = JPanel(GridLayout(0,2))
    self.frame.add(self.loginPanel)

    self.usernameField = JTextField('',15)
    self.loginPanel.add(JLabel("username:", SwingConstants.RIGHT))
    self.loginPanel.add(self.usernameField)

    self.passwordField = JPasswordField('', 15)
    self.loginPanel.add(JLabel("password:", SwingConstants.RIGHT))
    self.loginPanel.add(self.passwordField)

    self.loginButton = JButton('Log in',actionPerformed=self.login)
    self.loginPanel.add(self.loginButton)

    self.message = JLabel("Please Log in")
    self.loginPanel.add(self.message)

    self.frame.pack()
    self.frame.visible = True
```

The login method changes the label text and calls into python-twitter to attempt a login. It’s in a try/except block that will display “Log in failed” if something goes wrong. A real application would check different types of exceptions to see what went wrong and change the display message accordingly.

```
def login(self, event):
    self.message.text = "Attempting to Log in..."
    self.frame.show()
    username = self.usernameField.text
    try:
        self.api = twitter.Api(username, self.passwordField.text)
        self.timeline(username)
        self.loginPanel.visible = False
        self.message.text = "Logged in"
    except:
        self.message.text = "Log in failed."
        raise
    self.frame.size = 400,800
    self.frame.show()
```

If the login succeeds, we call the `timeline` method, which populates the frame with the latest tweets that the user is following. In the `timeline` method, we call `GetFriendsTimeline` from the python-twitter API, then we iterate through the status objects and call `showTweet` on each. All of this gets dropped into a `JScrollPane` and set to a reasonable size, then added to the main frame.

```
def timeline(self, username):
    timeline = self.api.GetFriendsTimeline(username)
    self.resultPanel = JPanel()
    self.resultPanel.layout = BoxLayout(self.resultPanel, BoxLayout.Y_AXIS)
    for s in timeline:
```

```
self.showTweet(s)

scrollpane = JScrollPane(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
                          JScrollPane.HORIZONTAL_SCROLLBAR_NEVER)
scrollpane.preferredSize = 400, 800
scrollpane.viewport.view = self.resultPanel

self.frame.add(scrollpane)
```

In the `showTweet` method, we go through the tweets and add a `JLabel` with the user's icon (fetched by url from `user.profile_image_url`) and a `JTextArea` to contain the text of the tweet. Note all of the bean properties that we set to get the `JTextArea` to display correctly.

```
def showTweet(self, status):
    user = status.user
    p = JPanel()

    # image grabbing seems very expensive, good place for a callback?
    p.add(JLabel(ImageIcon(URL(user.profile_image_url))))

    p.add(JTextArea(text = status.text,
                    editable = False,
                    wrapStyleWord = True,
                    lineWrap = True,
                    alignmentX = Component.LEFT_ALIGNMENT,
                    size = (300, 1)
                    ))
    self.resultPanel.add(p)
```

And that concludes our quick tour of Swing from Jython. Again, Swing is a very large subject, so you'll want to look into some more dedicated Swing resources to really get a handle on it. After this chapter, it should be reasonably straightforward to translate the Java examples you find into Jython examples.

Chapter 17: Deployment Targets

Deployment of Jython applications varies from container to container. However, they are all very similar and usually allow deployment of WAR file or exploded directory web applications. Deploying to “the cloud” is a different scenario all together. Some cloud environments have typical Java application servers available for hosting, while others such as the Google App Engine, and mobile run a bit differently. In this chapter, we'll discuss how to deploy web based Jython applications to a few of the more widely used Java application servers. We will also cover deployment of Jython web applications to the Google App Engine and mobile devices. While many of the deployment scenarios are quite similar, this chapter will walk through some of the differences from container to container.

In the end, one of the most important things to remember is that we need to make jython available to our application. There are different ways to do this, either by ensuring that the *jython.jar* file is included with the application server, or by packaging the JAR directly into each web application. This chapter assumes that you are using the latter technique. Placing the *jython.jar* directly into each web application is a good idea because it allows the web application to follow the Java paradigm of “deploy anywhere”. You do not need to worry whether you are deploying to Tomcat or Glassfish because the Jython runtime is embedded in your application.

Another new, yet very attractive solution is to deploy to the new Java Store. The Java Store is still in alpha mode at the time of this writing, but it will eventually afford developers of Java applications a place to deploy their apps and have them distributed via a nicely polished store front application. The distribution center for the Java Store is known as the Java Warehouse. In this chapter, we'll discuss a possible solution for packaging applications to deploy to the Java Warehouse.

Lastly, this section will briefly cover some of the reasons why mobile deployment is not yet a viable option for Jython. While a couple of targets exist in the mobile world, namely Android and JavaFX, both environments are still very new and Jython has not yet been optimized to run on either.

Application Servers

As with any Java web application, the standard web archive (WAR) files are universal throughout the Java application servers available today. This is good because it makes things a bit easier when it comes to the “write once run everywhere” philosophy that has been brought forth with the Java name. The great part of using Jython for deployment to application servers is just that, we can harness the technologies of the JVM to make our lives easier and deploy a Jython web application to any application server in the WAR format with very little tweaking.

If you have not yet used Django or Pylons on Jython, then you may not be aware that the resulting application to be deployed is in the WAR format. This is great because it leaves no assumption as to how the application should be deployed. All WAR files are deployed in the same manner according to each application server. This section will discuss how to deploy a WAR file on each of the three most widely used Java application servers. Now, all application servers are not covered in this section mainly due to the number of servers available today. Such a document would take more than one section of a book no doubt. However, you should be able to follow similar deployment instructions as those discussed here for any of the application servers available today for deploying Jython web applications in the WAR file format.

Tomcat

Arguably the most widely used of all Java application servers, Tomcat offers easy management and a small footprint compared to some of the other options available. Tomcat will plug into most IDEs that are in-use today, so you can manage the web container from within your development environment. This makes it handy to deploy and undeploy applications on-the-fly. For the purposes of this section, I’ve used Netbeans 6.7, so there may be some references to it.

To get started, download the Apache Tomcat server from the site at <http://tomcat.apache.org/>. Tomcat is constantly evolving, so I’ll note that when writing this book the deployment procedures were targeted for the 6.0.20 release. Once you have downloaded the server and placed it into a location on your hard drive, you may have to change permissions. I had to use the `chmod +x` command on the entire `apache-tomcat-6.0.20` directory before I was able to run the server. You will also need to configure an administrative account by going into the `/conf/tomcat-users.xml` file and adding one. Be sure to grant the administrative account the “manager” role. This should look something like the following once completed.

```
*tomcat-users.xml*
<tomcat-users>
  <user username="admin" password="myadminpassword" roles="manager"/>
</tomcat-users>
```

After this has been done you can add the installation to an IDE environment of your choice if you’d like. For instance, if you wish to add to Netbeans 6.7 you will need to go to the “Services” tab in the navigator, right-click on servers, choose “Tomcat 6.x” option, and then fill in the appropriate information pertaining to your environment. Once complete, you will be able to start, stop, and manage the Tomcat installation from the IDE.

Deploying Web Start

Deploying a web-start application is as easy as copying the necessary files to a location on the web server that is accessible via the web. In the case of Tomcat, you will need to copy the contents of your web start application to a single directory contained within the “<tomcat-root>/webapps/ROOT” directory. For instance, if you have a web-start application entitled *JythonWebStart*, then you would package the JAR file along with the JNLP and HTML file for the appli-

cation into a directory entitled *JythonWebStart* and then place that directory into the “<tomcat-root>/webapps/ROOT” directory.

Once the application has been copied the appropriate locations, you should be able to access it via the web if Tomcat is started. The URL should look something like the following: *http://your-server:8080/JythonWebStart/launch.jnlp*. Of course, you will need to use your server name and the port that you are using along with the appropriate JNLP name for your application.

Deploying a WAR or Exploded Directory Application

To deploy a web application to Tomcat, you have two options. You can either use a WAR file including all content for your entire web application, or you can deploy an exploded directory application which is basically copy-and-paste for your entire web application directory structure into the “<tomcat-root>/webapps/ROOT” directory. Either way will work the same, and we will discuss each technique in this section.

For manual deployment of a web application, you can copy either your exploded directory web application or your WAR file into the “<tomcat-root>/webapps” directory. By default, Tomcat is setup to “autodeploy” applications. This means that you can have Tomcat started when you copy your WAR or exploded directory into the “webapps” location. Once you’ve done this then you should see some feedback from the Tomcat server if you have a terminal open (or from within the IDE). After a few seconds the application should be deployed successfully and available via the URL. The bonus to deploying exploded directory applications is that you can take any file within the application and change it at will. Once you are done with the changes, that file will be redeployed when you save it...this really saves on development time!

If you do not wish to have autodeploy enabled (perhaps in a production environment), then you can deploy applications on startup of the server. This process is basically the same as “autodeploy” except any new applications that are copied into the “webapps” directory are not deployed until the server is restarted. Lastly, you can always make use of the Tomcat manager to deploy web applications as well. To do this, open your web browser to the index of Tomcat, usually <http://localhost:8080/index.html>, and then click on the “Manager” link in the left-hand menu. You will need to authenticate at that point using your administrator password, but once you are in the console deployment is quite easy. In an effort to avoid redundancy, I will once again redirect you to the Tomcat documentation for more information on deploying a web application via the Tomcat manager console.

Glassfish

At the time of this writing, the Glassfish V2 application server was mainstream and widely used. The Glassfish V3 server was still in preview mode but showed a lot of potential for Jython application deployment. In this section, we will cover WAR and web-start deployment to Glassfish V2 since it is the most widely used version. We will also discuss deployment for Django on Glassfish V3 since this version has added support for Django (and more Python web frameworks soon). Glassfish is very similar to Tomcat in terms of deployment, but there are a couple of minor differences which will be covered in this section.

To start out, you will need to download a glassfish distribution from the site at <https://glassfish.dev.java.net/>. Again, I recommend downloading V2 since it is the most widely used at the time of this writing. Installation is quite easy, but a little more involved than that of Tomcat. The installation of Glassfish will not be covered in this text as it varies depending upon which version you are using. There are detailed instructions for each version located on the Glassfish website, so I will redirect you there for more information.

Once you have Glassfish installed, you can utilize the server via the command-line or terminal, or you can use an IDE just like Tomcat. To register a Glassfish V2 or V3 installation with Netbeans 6.7, just go to the “Services” tab in the Netbeans navigator and right-click on “Servers” and then add the version you are planning to register. Once the “Add Server Instance” window appears, simply fill in the information depending upon your environment.

There is an administrative user named “admin” that is set up by default with a Glassfish installation. In order to change the default password, it is best to startup Glassfish and log into the administrative console. The default administrative console port is 4848.

Deploying Web Start

Deploying a web start application is basically the same as any other web server, you simply make the web start JAR, JNLP, and HTML file accessible via the web. On Glassfish, you need to traverse into your “domain” directory and you will find a “docroot” inside. The path should be similar to “<glassfish-install-loc>/domains/domain1/docroot”. Anything placed within the docroot area is visible to the web, so of course this is where you will place any web-start application directories. Again, a typical web start application will consist of your application JAR file, a JNLP file, and an HTML page used to open the JNLP. All of these files should typically be placed inside a directory appropriately named per your application, and then you can copy this directory into docroot.

WAR File and Exploded Directory Deployment

Again, there are a variety of ways to deploy an application using Glassfish. Let’s assume that you are using V2, you have the option to “hot deploy” or use the Glassfish Admin Console to deploy your application. Glassfish will work with either an exploded directory or WAR file deployment scenario. By default, the Glassfish “autodeploy” option is turned on, so it is quite easy to either copy your WAR or exploded directory application into the autodeploy location to deploy. If the application server is started, it will automatically start your application (if it runs without issues). The autodeploy directory for Glassfish V2 resides in the location “<glassfish-install-loc>/domains/domain1/autodeploy”.

Glassfish v3 Django Deployment

The Glassfish V3 server has some capabilities built into it to help facilitate the process of deploying a Django application. In the future, there will also be support for other Jython web frameworks such as Pylons.

Other Java Application Servers

If you have read through the information contained in the previous sections, then you have a fairly good idea of what it is like to deploy a Jython web application to a Java application server. There is no difference between deploying Jython web applications and Java web applications for the most part. You must be sure that you include *jython.jar* as mentioned in the introduction, but for the most part deployment is the same. However, I have run into cases with some application servers such as JBoss where it wasn’t so cut-and-dry to run a Jython application. For instance, I have tried to deploy a Jython servlet application on JBoss application server 5.1.0 GA and had lots of issues. For one, I had to manually add *servlet-api.jar* to the application because I was unable to compile the application in Netbeans without doing so...this was not the case with Tomcat or Glassfish. Similarly, I had issues trying to deploy a Jython web application to JBoss as there were several errors that had incurred when the container was scanning *jython.jar* for some reason.

All in all, with a bit of tweaking and perhaps an additional XML configuration file in the application, Jython web applications will deploy to *most* Java application servers. The bonus to deploying your application on a Java application server is that you are in complete control of the environment. For instance, you could embed the *jython.jar* file into the application server lib directory so that it was loaded at startup and available for all applications running in the environment. Likewise, you are in control of other necessary components such as database connection pools and so forth. If you deploy to another service that lives in “the cloud”, you have very little control over the environment. In the next section, we’ll study one such environment by Google which is known as the Google App Engine. While this “cloud” service is an entirely different environment than your basic Java web application server, it contains some nice features that allow one to test applications prior to deployment in the cloud.

Google App Engine

The new kid on the block, at least for the time of this writing, is the Google App Engine. Fresh to the likes of the Java platform, the Google App Engine can be used for deploying applications written in just about any language that

runs on the JVM, Jython included. The App Engine went live in April of 2008, allowing Python developers to begin using its services to host Python applications and libraries. In the spring of 2009, the App Engine added support for the Java platform. Along with support of the Java language, most other languages that run on the JVM will also deploy and run on the Google App Engine, including Jython. It has been mentioned that more programming languages will be supported at some point in the future, but at the time of this writing Python and Java were the only supported languages.

The App Engine actually runs a slightly slimmed-down version of the standard Java library. You must download and develop using the Google App Engine SDK for Java in order to ensure that your application will run in the environment. You can download the SDK by visiting this link: <http://code.google.com/appengine/downloads.html> along with viewing the extensive documentation available on the Google App Engine site. The SDK comes complete with a development web server that can be used for testing your code before deploying, and several demo applications ranging from easy JSP programs to sophisticated demos that use Google authentication. No doubt about it, Google has done a good job at creating an easy learning environment for the App Engine so that developers can get up and running quickly.

In this section you will learn how to get started using the Google App Engine SDK, and how to deploy some Jython web applications. You will learn how to deploy a Jython servlet application as well as a WSGI application utilizing modjy. Once you've learned how to develop and use a Jython Google App Engine program using the development environment, you will learn a few specifics about deploying to the cloud. If you have not done so already, be sure to visit the link mentioned in the previous paragraph and download the SDK so that you can follow along in the sections to come.

Please note that the Google App Engine is a very large topic. Entire books could be written on the subject of developing Jython applications to run on the App Engine. With that said, I will cover the basics to get up and running with developing Jython applications for the App Engine. Once you've read through this section I suggest to go to the Google App Engine documentation for further details.

Starting with an SDK Demo

We will start by running the demo application known as “guestbook” that comes with the Google App Engine SDK. This is a very simple Java application that allows one to sign in using an email address and post messages to the screen. In order to start the SDK web server and run the “guestbook” application, open up a terminal and traverse into the directory where you expanded the Google App Engine .zip file and run the following command:

```
<app-engine-base-directory>/bin/dev_appserver.sh demos/guestbook/war
```

Of course, if you are running on windows there is a corresponding .bat script for you to run that will start the web server. Once you've issued the preceding command it will only take a second or two before the web server starts. You can then open a browser and traverse to <http://localhost:8080> to invoke the “guestbook” application. This is a basic JSP-based Java web application, but we can deploy a Jython application and use it in the same manner as we will see in a few moments. You can stop the web server by pressing “CTRL+C”.

Deploying to the Cloud

Prior to deploying your application to the cloud, you must of course set up an account with the Google App Engine. If you have another account with Google such as GMail, then you can easily activate your App Engine account using that same username. To do so, go to the Google App Engine link: <http://code.google.com/appengine/> and click “Sign Up”. Enter your existing account information or create a new account to get started.

After your account has been activated you will need to create an application by clicking on the “Create Application” button. You have a total of 10 available application slots to use if you are making use of the free App Engine account. Once you've created an application then you are ready to begin deploying to the cloud. In this section of the book, we create an application known as *jythongae*. This is the name of the application that you must create on the App Engine. You must also ensure that this name is supplied within the *appengine-web.xml* file.

Working with a Project

The Google App Engine provides project templates to get you started developing using the correct directory structure. Eclipse has a plugin that makes it easy to generate Google App Engine projects and deploy them to the App Engine. If interested in making use of the plugin, please visit <http://code.google.com/appengine/docs/java/tools/eclipse.html> to read more information and download the plugin. Similarly, Netbeans has an App Engine plugin that is available on the Kenai site appropriately named *nbappengine* (<http://kenai.com/projects/nbappengine>). In this text we will cover the use of Netbeans 6.7 to develop a simple Jython servlet application to deploy on the App Engine. You can either download and use the template available with one of these IDE plugins, or simply create a new Netbeans project and make use of the template provided with the App Engine SDK (<app-engine-base-directory/demos/new_project_template) to create your project directory structure. For the purposes of this tutorial, we will make use of the *nbappengine* plugin. If you are using Eclipse you will find a section following this tutorial that provides some Eclipse plugin specifics.

In order to install the *nbappengine* plugin, you add the ‘App Engine’ update center to the Netbeans plugin center by choosing the *Settings* tab and adding the update center using <http://deadlock.netbeans.org/hudson/job/nbappengine/lastSuccessfulBuild/artifact/build/updates/updates.xml.gz> as the URL. Once you’ve added the new update center you can select the *Available Plugins* tab and add all of the plugins in the “Google App Engine” category then choose *Install*. After doing so, you can add the “App Engine” as a server in your Netbeans environment using the “Services” tab. To add the server, point to the base directory of your Google App Engine SDK. Once you have added the App Engine server to Netbeans then it will become an available deployment option for your web applications.

Create a new Java web project and name it *JythonGAE*. For the deployment server, choose “Google App Engine”, and you will notice that when your web application is created an additional file will be created within the *WEB-INF* directory named *appengine-web.xml*. This is the Google App Engine configuration file for the JythonGAE application. Any of the .py files that we wish to use in our application must be mapped in this file so that they will *not* be treated as static files by the Google App Engine. By default, Google App Engine treats all files outside of the *WEB-INF* directory as static unless they are JSP files. Our application is going to make use of three Jython servlets, namely *NewJythonServlet.py*, *AddNumbers.py* and *AddToPage.py*. In our *appengine-web.xml* file we can exclude all .py files from being treated as static by adding the suffix to the exclusion list as follows.

appengine-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <application>jythongae</application>
  <version>1</version>
  <static-files>
    <exclude path="/**/*.py"/>
  </static-files>
  <resource-files/>
  <ssl-enabled>>false</ssl-enabled>
  <sessions-enabled>>true</sessions-enabled>
</appengine-web-app>
```

At this point we will need to create a couple of additional directories within our *WEB-INF* project directory. We should create a *lib* directory and place *jython.jar* and *appengine-api-1.0-sdk-1.2.2.jar* into the directory. Note that the App Engine JAR may be named differently according to the version that you are using. We should now have a directory structure that resembles the following:

```
JythonGAE
  WEB-INF
    lib
      jython.jar
      appengine-api-1.0-sdk-1.2.2.jar
    appengine-web.xml
    web.xml
  src
```

web

Now that we have the application structure set up, it is time to begin building the actual logic. In a traditional Jython servlet application we need to ensure that the *PyServlet* class is initialized at startup and that all files ending in *.py* are passed to it. As we've seen in chapter 13, this is done in the *web.xml* deployment descriptor. However, I have found that this alone does not work when deploying to the cloud. I found some inconsistencies while deploying against the Google App Engine development server and deploying to the cloud. For this reason, I will show you the way that I was able to get the application to function as expected in both the production and development Google App Engine environments. In chapter 12, the object factory pattern for coercing Jython classes into Java was discussed. If this same pattern is applied to Jython servlet applications then we can use the factories to coerce our Jython servlet into Java bytecode at runtime. We then map the resulting coerced class to a servlet mapping in the application's *web.xml* deployment descriptor. We can also deploy our Jython applets and make use of *PyServlet* mapping to the *.py* extension in the *web.xml*. I will comment in the source where the code for the two implementations differs.

Object Factories with App Engine

In order to use object factories to coerce our code, we must use an object factory along with a Java interface, and once again we will use the PlyJy project to make this happen. Please note that if you choose to not use the object factory pattern and instead use *PyServlet* you can safely skip forward to the next subsection. The first step is to add *PlyJy.jar* to the *lib* directory that we created previously to ensure it is bundled with our application. There is a Java servlet contained within the PlyJy project named *JythonServletFacade*, and what this Java servlet does is essentially use the *JythonObjectFactory* class to coerce a named Jython servlet and then invoke its resulting *doGet* and *doPost* methods. There is also a simple Java interface named *JythonServletInterface* in the project, and it must be implemented by our Jython servlet in order for the coercion to work as expected. Below you will see these two pieces of code that are contained in the PlyJy project.

JythonServletFacade.java

```
public class JythonServletFacade extends HttpServlet {

    private JythonObjectFactory factory = null;

    String pyServletName = null;

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        factory = factory.getInstance();
        pyServletName = getInitParameter("PyServletName");
        JythonServletInterface jythonServlet = (JythonServletInterface) factory.
        ↪createObject(JythonServletInterface.class, pyServletName);
        jythonServlet.doGet(request, response);
    }
    ...
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        factory = factory.getInstance();
        pyServletName = getInitParameter("PyServletName");
        JythonServletInterface jythonServlet = (JythonServletInterface) factory.
        ↪createObject(JythonServletInterface.class, pyServletName);
        jythonServlet.doPost(request, response);
    }
    ...
}
```

JythonServletInterface.java

```
public interface JythonServletInterface {
    public void doGet(HttpServletRequest request, HttpServletResponse response);
    public void doPost(HttpServletRequest request, HttpServletResponse response);
}
```

Using PyServlet Mapping

When we use the PyServlet mapping implementation, there is no need to coerce objects using factories. You simply set up a servlet mapping within *web.xml* and use your Jython servlets directly with the .py extension in the URL. However, I've seen issues while using PyServlet on the App Engine in that this implementation will deploy to the development App Engine server environment, but when deployed to the cloud you will receive an error when trying to invoke the servlet. It is because of these inconsistencies that I chose to implement the object factory solution for Jython servlet to App Engine deployment.

Example Jython Servlet Application for App Engine

The next piece of the puzzle is the code for our application. In this example, we'll make use of a simple servlet that displays some text as well as the same example that was used in chapter 13 with JSP and Jython. The code below sets up three Jython servlets. The first servlet simply displays some output, the next two perform some mathematical logic, and then there is a JSP to display the results for the mathematical servlets.

NewJythonServlet.py

```
from javax.servlet.http import HttpServlet
from org.plyjy.interfaces import JythonServletInterface

class NewJythonServlet (JythonServletInterface, HttpServlet):
    def doGet(self, request, response):
        self.doPost (request, response)

    def doPost(self, request, response):
        toClient = response.getWriter()
        response.setContentType ("text/html")
        toClient.println ("<html><head><title>Jython Servlet Test Using_
↳Object Factory</title>" +
                                                                    "<body><h1>Jython Servlet Test for_
↳GAE</h1></body></html>")

    def getServletInfo(self):
        return "Short Description"
```

AddNumbers.py

```
import javax
class add_numbers (javax.servlet.http.HttpServlet):
    def doGet(self, request, response):
        self.doPost(request, response)
    def doPost(self, request, response):
        x = request.getParameter("x")
        y = request.getParameter("y")
        if not x or not y:
            sum = "<font color='red'>You must place numbers in each value box</font>"
        else:
```

```
        try:
            sum = int(x) + int(y)
        except ValueError, e:
            sum = "<font color='red'>You must place numbers only in each value box
↪</font>"
    request.setAttribute("sum", sum)
    dispatcher = request.getRequestDispatcher("testJython.jsp")
    dispatcher.forward(request, response)
```

AddToPage.py

```
import java, javax, sys

class add_to_page(javax.servlet.http.HttpServlet):
    def doGet(self, request, response):
        self.doPost(request, response)

    def doPost(self, request, response):
        addtext = request.getParameter("p")
        if not addtext:
            addtext = ""

        request.setAttribute("page_text", addtext)
        dispatcher = request.getRequestDispatcher("testJython.jsp")
        dispatcher.forward(request, response)
```

testjython.jsp - Note that this implementation differs if you plan to make use of the object factory technique. Instead of using *add_to_page.py* and *add_numbers.py* as your actions, you would utilize the servlet instead, namely */add_to_page* and */add_numbers*

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Jython JSP Test</title>
  </head>
  <body>
    <form method="GET" action="add_to_page.py">
      <input type="text" name="p">
      <input type="submit">
    </form>
    <%
      Object page_text = request.getAttribute("page_text");
      Object sum = request.getAttribute("sum");
      if(page_text == null){
        page_text = "";
      }
      if(sum == null){
        sum = "";
      }
    %>
    <br/>
    <p><%= page_text %></p>
    <br/>
    <form method="GET" action="add_numbers.py">
      <input type="text" name="x">
      +
      <input type="text" name="y">
      =
      <%= sum %>
    </form>
  </body>
</html>
```

```

        <br/>
        <input type="submit" title="Add Numbers">
    </form>

</body>
</html>

```

As mentioned previously, it is important that all of the Jython servlets reside within your classpath somewhere. If using Netbeans, you can either place the servlets into the source root of your project (not inside a package), or you can place them in the web folder that contains your JSP files. If doing the latter, I have found that you may have to tweak your CLASSPATH a bit by adding the web folder to your list of libraries from within the project properties. Next, we need to ensure that the deployment descriptor includes the necessary servlet definitions and mappings for the application. Now, if you are using the object factory implementation and the *JythonServletFacade* servlet, you would have noticed that there is a variable named *PyServletName* which the *JythonObjectFactory* is using as the name of our Jython servlet. Well, within the *web.xml* we must pass an *<init-param>* using *PyServletName* as the *<param-name>* and the name of our Jython servlet as the *<param-value>*. This will basically pass the name of the Jython servlet to the *JythonServletFacade* servlet so that it can be used by the object factory.

web.xml

```

<web-app>
  <display-name>Jython Google App Engine</display-name>

  <!-- Used for the PyServlet Implementation -->
  <servlet>
    <servlet-name>PyServlet</servlet-name>
    <servlet-class>org.python.util.PyServlet</servlet-class>
  </servlet>

  <!-- The next three servlets are used for the object factory implementation only.
       They can be excluded in the PyServlet implementation -->
  <servlet>
    <servlet-name>NewJythonServlet</servlet-name>
    <servlet-class>org.plyjy.servlets.JythonServletFacade</servlet-class>
    <init-param>
      <param-name>PyServletName</param-name>
      <param-value>NewJythonServlet</param-value>
    </init-param>
  </servlet>
  <servlet>
    <servlet-name>AddNumbers</servlet-name>
    <servlet-class>org.plyjy.servlets.JythonServletFacade</servlet-class>
    <init-param>
      <param-name>PyServletName</param-name>
      <param-value>AddNumbers</param-value>
    </init-param>
  </servlet>
  <servlet>
    <servlet-name>AddToPage</servlet-name>
    <servlet-class>org.plyjy.servlets.JythonServletFacade</servlet-class>
    <init-param>
      <param-name>PyServletName</param-name>
      <param-value>AddToPage</param-value>
    </init-param>
  </servlet>

  <!-- The following mapping should be used for the PyServlet implementation -->

```

```
<servlet-mapping>
  <servlet-name>PyServlet</servlet-name>
  <url-pattern>*.py</url-pattern>
</servlet-mapping>

<!-- The following three mappings are used in the object factory implementation --
→>

<servlet-mapping>
  <servlet-name>NewJythonServlet</servlet-name>
  <url-pattern>/NewJythonServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>AddNumbers</servlet-name>
  <url-pattern>/AddNumbers</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>AddToPage</servlet-name>
  <url-pattern>/AddToPage</url-pattern>
</servlet-mapping>
</web-app>
```

Note that when using the PyServlet implementation you should exclude those portions in the *web.xml* above that are used for the object factory implementation. The PyServlet mapping can be contained within the *web.xml* in both implementations without issue. That's it, now you can deploy the application to your Google App Engine development environment and it should run without any issues. You can also choose to deploy to another web server to test for compatibility if you wish. You can deploy directly to the cloud by right-clicking the application and choosing the "Deploy to App Engine" option.

Using Eclipse

If you wish to use the Eclipse IDE for development, you should definitely download the Google App Engine plugin using the link provided earlier in the chapter. You should also use the PyDev plugin which is available at <http://pydev.sourceforge.net/>. For the purposes of this section, I used Eclipse Galileo and started a new project named "JythonGAE" as a Google Web Application. When creating the project, make sure you check the box for using Google App Engine and uncheck the Google Web Toolkit option. You will find that Eclipse creates a directory structure for your application that is much the same as the project template that is included with the Google App Engine SDK.

If you follow through the code example from the previous section, you can create the same code and set up the *web.xml* and *appengine-web.xml* the same way. The key is to ensure that you create a *lib* directory within the *WEB-INF* and you place the files in the appropriate location. You will need to ensure that your Jython servlets are contained in your *CLASSPATH* by either adding them to the source root for your project, or by going into the project properties and adding the *war* directory to your *Java Build Path*. When doing so, make sure you do *not* include the *WEB-INF* directory or you will receive errors.

When you are ready to deploy the application, you can choose to use the Google App Engine development environment or deploy to the cloud. You can run the application by right-clicking on the project and choosing *Run As* option and then choose the Google Web Application option. The first time you run the application you may need to set up the runtime. If you are ready to deploy to the cloud, you can right-click on the project and choose the *Google -> Deploy to App Engine* option. After entering your Google username and password then your application will be deployed.

Deploy Modjy to GAE

We can easily deploy WSGI applications using Jython's modjy API as well. To do so, you need to add an archive of the Jython *Lib* directory to your *WEB-INF* project directory. According to the modjy website, you need to obtain

the source for Jython, then zip the *Lib* directory and place it into another directory along with a file that will act as a pointer to the zip archive. The modjy site names the directory *python-lib* and names the pointer file *all.pth*. This pointer file can be named anything as long as the suffix is *.pth*. Inside the pointer file you need to explicitly name the zip archive that you had created for the *Lib* directory contents. Let's assume you named it *lib.zip*, in this case we will put the text "lib.zip" without the quotes into the *.pth* file. Now if we add the modjy *demo_app.py* demonstration application to the project then our directory structure should look as follows:

```
modjy_app
  demo_app.py
  WEB-INF
    lib
      jython.jar
      appengine-api-1.0-sdk-1.2.2.jar
  python-lib
    lib.zip
    all.pth
```

Now if we run the application using Tomcat it should run as expected. Likewise, we can run it using the Google App Engine SDK web server and it should provide the expected results.

Summary

The Google App Engine is certainly an important deployment target for Jython. Google offers free hosting for smaller applications, and they also base account pricing on bandwidth. No doubt that it is a good way to put up a small site, and possibly build on it later. Most importantly, you can deploy Django, Pylons, and other applications via Jython to the App Engine by setting up your App Engine applications like the examples I had shown in this chapter.

Java Store

Another deployment target that is hot off the presses at the time of this book is the Java Store or Java Warehouse. This is a new concept brought to market by Sun Microsystems in order to help Java software developers market their applications via a single shop that is available online via a web start application. Similar to other application venues, The Java Store is a store front application where people can go to search for applications that have been submitted by developers. The Java Warehouse is the repository of applications that are contained within the Java Store. This looks to be a very promising target for Java and Jython developers alike. It *should* be as easy as generating a JAR file that contains a Jython application and deploying it to the Java Store. Unfortunately, since the program is still in alpha mode at this time I am unable to provide any specific details on distributing Jython applications via the Java Store. However, there are future plans to make alternative VM language applications easily deployable to the Java Warehouse. At this time, it is certainly possible to deploy a Jython application to the warehouse, but it can only deploy as a Java application. As of the time of this writing, only Java and JavaFX applications are directly deployable to the Java Warehouse. Please note that since this product is still in alpha mode, this book will not discuss such aspects of the program as memberships or fees that may be incurred for hosing your applications on the Java Store.

The requirements for publishing an application to the warehouse are as follows:

- Your application packed in a single jar file
- Descriptive text to document your application
- Graphic image files used for icons and to give the consumer an idea of your application's look.

In chapter 13, we took a look at packaging and distributing Jython GUI applications in a JAR file. When a Jython application is packaged in a JAR file then it is certainly possible to use Java Web Start to host the application via the web. On the other hand, if one wishes to make a Jython GUI application available for purchase or for free, the Java Store would be another way of doing so. One likely way to deploy applications in a single JAR is to use the method discussed in chapter 13, but there are other solutions as well. For instance, one could use the *One-Jar* product to create

a single JAR file containing all of the necessary Jython code as well as other JAR files essential to the application. In the following section, we will discuss deployment of a Jython application using One-JAR so that you can see some similarities and differences to using the Jython standalone JAR technique.

Deploying a Single JAR

In order to deploy an application to the Java Warehouse, it must be packaged as a single JAR file. We've already discussed packaging Jython applications into a JAR file using the Jython standalone method in chapter 13. In this section, you will learn how to make use of the One-JAR (<http://one-jar.sourceforge.net/>) product to distribute client-based Jython applications. In order to get started, you will need to grab a copy of One-JAR. There are a few options available on the download site, but for our purposes we will package an application using the source files for One-JAR. Once downloaded, the source for the project should look as follows.

```
src
  com
    simontuffs
      onejar
        Boot.java
        Handler.java
        IProperties.java
        JarClassLoader.java
```

This source code for the One-Jar project must reside within the JAR file that we will build. Next, we need to create separate source directories for both our Jython source and our Java source. Likewise, we will create a separate source directory for the One-Jar source. Lastly, we'll create a *lib* directory into which we will place all of the required JAR files for the application. In order to run a Jython application, we'll need to package the Jython project source into a JAR file for our application. We will not need to use the entire *jython.jar*, but rather only a standalone version of it. The easiest way to obtain a standalone Jython JAR is to run the installer and choose the standalone option. After this is done, simply add the resulting *jython.jar* to the *lib* directory of application. In the end, the directory structure should resemble the following.

```
one-jar-jython-example
  java
  lib
    jython.jar
  LICENSE.txt
  onejar
    src
      com
        one-jar-license.txt
        simontuffs
          onejar
            Boot.java
            Handler.java
            IProperties.java
            JarClassLoader.java
  src
```

As you can see from the depiction of the file structure above, the *src* directory will contain our Jython source files. The *LICENSE.txt* included in this example was written by Ryan McGuire (<http://www.enigmacurry.com>). He has a detailed explanation of using One-Jar on his blog, and I've replicated some of his work in this example...including a version of the *build.xml* that we will put together in order to build the application. Let's take a look at the build file that we will use to build the application JAR. In this example I am using Apache Ant for the build system, but you could choose something different if you'd like.

build.xml


```

<project name="JythonSwingApp" default="dist" basedir=".">

  <!-- #####
        These two properties are the only ones you are
        likely to want to change for your own projects:      -->
  <property name="jar.name" value="JythonSwingApp.jar" />
  <property name="java-main-class" value="Main" />
  <!-- ##### -->

  <!-- Below here you dont' need to change for simple projects -->
  <property name="src.dir" location="src"/>
  <property name="java.dir" location="java"/>
  <property name="onejar.dir" location="onejar"/>
  <property name="java-build.dir" location="java-build"/>
  <property name="build.dir" location="build"/>
  <property name="lib.dir" location="lib"/>

  <path id="classpath">
    <fileset dir="${lib.dir}" includes="**/*.jar"/>
  </path>

  <target name="clean">
    <delete dir="${java-build.dir}"/>
    <delete dir="${build.dir}"/>
    <delete file="${jar.name}"/>
  </target>

  <target name="dist" depends="clean">
    <!-- prepare the build directory -->
    <mkdir dir="${build.dir}/lib"/>
    <mkdir dir="${build.dir}/main"/>
    <!-- Build java code -->
    <mkdir dir="${java-build.dir}"/>
    <javac srcdir="${java.dir}" destdir="${java-build.dir}" classpathref="classpath"/>
    <!-- Build main.jar -->
    <jar destfile="${build.dir}/main/main.jar" basedir="${java-build.dir}">
      <manifest>
        <attribute name="Main-Class" value="Main" />
      </manifest>
    </jar>
    <delete file="${java-build.dir}"/>
    <!-- Add the python source -->
    <copy todir="${build.dir}">
      <fileset dir="${src.dir}"/>
    </copy>
    <!-- Add the libs -->
    <copy todir="${build.dir}/lib">
      <fileset dir="${lib.dir}"/>
    </copy>
    <!-- Compile OneJar -->
    <javac srcdir="${onejar.dir}" destdir="${build.dir}" classpathref="classpath"/>
    <!-- Copy the OneJar license file -->
    <copy file="${onejar.dir}/one-jar-license.txt" tofile="${build.dir}/one-jar-
    ↪license.txt" />
    <!-- Build the jar -->
    <jar destfile="${jar.name}" basedir="${build.dir}">
      <manifest>
        <attribute name="Main-Class" value="com.simontuffs.onejar.Boot" />

```

```
        <attribute name="Class-Path" value="lib/jython-full.jar" />
    </manifest>
</jar>
<!-- clean up -->
<delete dir="${java-build.dir}" />
<delete dir="${build.dir}" />
</target>

</project>
```

Since this is a Jython application, we can use as much Java source as we'd like. In this example, we will only use one Java source file *Main.java* to “drive” our application. In this case, we'll use the *PythonInterpreter* inside of our *Main.java* to invoke our simple Jython Swing application. Now let's take a look at the *Main.java* source.

Main.java

```
import org.python.core.PyException;
import org.python.util.PythonInterpreter;

public class Main {
    public static void main(String[] args) throws PyException{
        PythonInterpreter intrp = new PythonInterpreter();
        intrp.exec("import JythonSimpleSwing as jy");
        intrp.exec("jy.JythonSimpleSwing().start()");
    }
}
```

Now that we've written the driver class, we'll place it into our *java* source directory. As stated previously, we'll place our Jython code into the *src* directory. In this example we are using the same simple Jython swing application that I wrote for chapter 13.

JythonSimpleSwing.py

```
import sys
sys.packageManager.makeJavaPackage("javax.swing", "java.awt", None)
import javax.swing as swing
import java.awt as awt

class JythonSimpleSwing(object):
    def __init__(self):
        self.frame=swing.JFrame(title="My Frame", size=(300,300))
        self.frame.defaultCloseOperation=swing.JFrame.EXIT_ON_CLOSE;
        self.frame.layout=awt.BorderLayout()
        self.panel1=swing.JPanel(awt.BorderLayout())
        self.panel2=swing.JPanel(awt.GridLayout(4,1))
        self.panel2.preferredSize = awt.Dimension(10,100)
        self.panel3=swing.JPanel(awt.BorderLayout())

        self.title=swing.JLabel("Text Rendering")
        self.button1=swing.JButton("Print Text", actionPerformed=self.printMessage)
        self.button2=swing.JButton("Clear Text", actionPerformed=self.clearMessage)
        self.textField=swing.JTextField(30)
        self.outputText=swing.JTextArea(4,15)

        self.panel1.add(self.title)
        self.panel2.add(self.textField)
        self.panel2.add(self.button1)
```

```

self.panel2.add(self.button2)
self.panel3.add(self.outputText)

self.frame.contentPane.add(self.panel1, awt.BorderLayout.PAGE_START)
self.frame.contentPane.add(self.panel2, awt.BorderLayout.CENTER)
self.frame.contentPane.add(self.panel3, awt.BorderLayout.PAGE_END)

def start(self):
    self.frame.visible=1

def printMessage(self, event):
    print "Print Text!"
    self.text = self.textField.getText()
    self.outputText.append(self.text)

def clearMessage(self, event):
    self.outputText.text = ""

```

In order to import the swing and awt packages, we need to make use of the *sys.packagemanager.makeJavaPackage* utility. For some reason the application would run fine stand alone without using this utility, but when placed into a JAR using this method we need to assist in loading the packages. At this time, the application is ready build using Ant. In order to run the build, simply traverse into the directory that contains *build.xml* and initiate the *ant* command. The resulting JAR can be run using the following syntax:

```
java -jar JythonSwingApp.jar
```

In some situations, such as deploying via web start, this JAR file will also need to be signed. There are many resources online that explain the signing of JAR files that topic will not be covered in this text. The JAR is now ready to be deployed and used on other machines. This method will be a good way to package an application for distribution via the Java Store.

Mobile

Mobile applications are the way of the future. At this time, there are a couple of different options for developing mobile applications using Jython. One way to develop mobile applications using Jython is to make use of the JavaFX API from Jython. Since JavaFX is all Java behind the scenes, it would be fairly simple to make use of the JavaFX API using Jython code. However, this technique is not really a production-quality result in my opinion for a couple of reasons. First, the JavaFX scripting language makes GUI development quite easy. While it is possible (see <http://wiki.python.org/jython/JythonMonthly/Articles/December2007/2> for more details), the translation of JavaFX API using Jython would not be as easy as making use of the JavaFX script language. The second reason this is not feasible at the time of this writing is that JavaFX is simply not available on all mobile devices at this time. It is really just becoming available to the mobile world at this time and will take some time to become acclimated.

Another way to develop mobile applications using Jython is to make use of the Android operating system which is available via Google. Android is actively being used on mobile devices today, and it's use is continuing to grow. Although in early stages, there is a project known as *Jythonroid* that is an implementation of Jython for the Android Dalvik Virtual Machine. Unfortunately, it was not under active development at the time of this writing, although some potential does exist for getting the project on track.

If you are interested in mobile development using Jython, please pay close attention to the two technologies discussed in this section. They are the primary deployment targets for Jython in the mobile world. As for the *Jythonroid* project, it is open source and available to developers. Interested parties may begin working on it again to make it functional and bring it up to date with the latest Android SDK.

Conclusion

Deploying Jython applications is very much like Java application deployment. For those of you who are familiar with Java application servers, deploying a Jython application should be a piece of cake. On the contrary, for those of you who are not familiar with Java application deployment this topic may take a bit of getting used to. In the end, it is easy to deploy a Jython web or client application using just about any of the available Java application servers that are available today.

Deploying Jython web applications is universally easy to do using the WAR file format. As long as *jython.jar* is either in the application server classpath or packaged along with the web application, Jython servlets should function without issue. We also learned that it is possible to deploy a JAR file containing a Jython GUI application via Java web start technology. Using a JNLP deployment file is quite easy to do, the trick to deploying Jython via a JAR file is to set the file up correctly. Once completed, an HTML page can be used to reference the JNLP and initiate the download of the JAR to the client machine.

Lastly, this section discussed use of the Google App Engine for deploying Jython servlet applications. While the Google App Engine environment is still relatively new at the time of this writing, it is an exceptional deployment target for any Python or Java application developer. Using a few tricks with the object factory technique, it is possible to deploy Jython servlets and use them directly or via a JSP file on the App Engine. Stay tuned for more deployment targets to become available for Jython in the coming months and years. As cloud computing and mobile devices are becoming more popular, the number of deployment targets will continue to grow and Jython will be more useful with each one.

Chapter 18: Testing and Continuous Integration

Nowadays, automated testing is a fundamental activity in software development. In this chapter you will see a survey of the tools available for Jython in this field, from common tools used in the Python world to aid with unit testing to more complex tools available in the Java world which can be extended or driven using Jython.

Python Testing Tools

UnitTest

First we will take a look at the most classic test tool available in Python: unittest. It follows the conventions of most “xUnit” incarnations (like JUnit): You subclass from `TestCase` class, write test methods (which must have a name starting by “test” and optionally override the methods `setUp()` and `tearDown()` which are executed around the test methods,. And you can use the multiple `assert*()` methods provided by `TestCase`. Here is an very simple test case for the some functions of the built-in `math` module:

```
import math
import unittest

class TestMath(unittest.TestCase):
    def testFloor(self):
        self.assertEqual(1, math.floor(1.01))
        self.assertEqual(0, math.floor(0.5))
        self.assertEqual(-1, math.floor(-0.5))
        self.assertEqual(-2, math.floor(-1.1))

    def testCeil(self):
        self.assertEqual(2, math.ceil(1.01))
        self.assertEqual(1, math.ceil(0.5))
        self.assertEqual(0, math.ceil(-0.5))
        self.assertEqual(-1, math.ceil(-1.1))
```

There are many other assertion methods besides `assertEqual()`, of course. Here is a list with the rest of the available assertion methods:

- `assertNotEqual(a, b)`: The opposite of `assertEqual()`
- `assertAlmostEqual(a, b)`: Only used for numeric comparison. It adds a sort of tolerance for insignificant differences, by subtracting its first two arguments after rounding them to the seventh decimal place, and later comparing the result to zero. You can specify a different number of decimal places in the third argument. This is useful for comparison of floating point numbers.
- `assertNotAlmostEqual(a, b)`: The opposite of `assertAlmostEqual()`
- `assert_(x)`: Accepts a boolean argument expecting it to be `True`. You can use it to write other checks like “greater than”, or to check boolean functions/attributes (The trailing underscore is needed because `assert` is a keyword).
- `assertFalse(x)`: The opposite of `assert_()`.
- `assertRaises(exception, callable)`: Used to assert that an exception passed as the first argument is thrown when invoking the callable specified as the second argument. The rest of arguments passed to `assertRaises` is passed on to the callable.

As an example, let’s extend our test of mathematical functions using some of these other assertion functions:

```
import math
import unittest
import operator

class TestMath(unittest.TestCase):

    # ...

    def testMultiplication(self):
        self.assertAlmostEqual(0.3, 0.1 * 3)

    def testDivision(self):
        self.assertRaises(ZeroDivisionError, operator.div, 1, 0)
        # The same assertion using a different idiom:
        self.assertRaises(ZeroDivisionError, lambda: 1 / 0)
```

Now, you may be wondering how to run this test case. The simple answer is to add the following to the file in which we defined it:

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

Finally, just run the module. Say, if you wrote all this code on a file named `test_math.py`, then run:

```
$ jython test_math.py
```

And you will see this output:

```
....
-----
Ran 4 tests in 0.005s

OK
```

Each dot about the dash line represent a successfully ran test. Let see what happens if we add a test that fails. Change

the invocation `assertAlmostEqual()` method in `testMultiplication()` to use `assertEqual()` instead. If you run the module again, you will see the following output:

```
...F
=====
FAIL: testMultiplication (__main__.TestMath)
-----
Traceback (most recent call last):
  File "test_math.py", line 22, in testMultiplication
    self.assertEqual(0.3, 0.1 * 3)
AssertionError: 0.3 != 0.30000000000000004

-----
Ran 4 tests in 0.030s

FAILED (failures=1)
```

As you can see, the last dot is now an “F”, and an explanation of the failure is printed, pointing out that 0.3 and 0.30000000000000004 are not equal. The last line also shows the grand total of 1 failure.

By the way, now you can imagine why using `assertEquals(x, y)` is better than `assert_(x == y)`: if the test fails, `assertEquals()` provides helpful information, which `assert_()` can’t possibly provide by itself. To see this in action, let’s change `testMultiplication()` to use `assert_()`:

```
class TestMath(unittest.TestCase):

    #...

    def testMultiplication(self):
        self.assert_(0.3 == 0.1 * 3)
```

If you run the test again, the output will be:

```
...F
=====
FAIL: testMultiplication (__main__.TestMath)
-----
Traceback (most recent call last):
  File "test_math.py", line 24, in testMultiplication
    self.assert_(0.3 == 0.1 * 3)
AssertionError

-----
Ran 4 tests in 0.054s

FAILED (failures=1)
```

Now all what we have is the traceback and the “AssertionError” message. No extra information is provided to help us diagnostic the failure, as it was the case when we use `assertEqual()`. That’s why all the specialized `assert*()` methods are so helpful. Actually, with the exception of `assertRaises()` all assertion methods accept an extra parameter meant to be the debugging message which will be shown in case the test fails. That lets you write helper methods like:

```
class SomeTestCase(unittest.TestCase):
    def assertGreaterThan(a, b):
        self.assert_(a > b, '%d isn't greater than %d')

    def testSomething(self):
```

```
self.assertGreaterThan(10, 4)
```

As your application gets bigger, the number of test cases will grow too. Eventually, you may not want to keep all the tests on one python module, for maintainability reasons.

Let's create a new module, named `test_lists.py` with the following test code:

```
import unittest

class TestLists(unittest.TestCase):
    def setUp(self):
        self.list = ['foo', 'bar', 'baz']

    def testLen(self):
        self.assertEqual(3, len(self.list))

    def testContains(self):
        self.assert_('foo' in self.list)
        self.assert_('bar' in self.list)
        self.assert_('baz' in self.list)

    def testSort(self):
        self.assertNotEqual(['bar', 'baz', 'foo'], self.list)
        self.list.sort()
        self.assertEqual(['bar', 'baz', 'foo'], self.list)
```

Note: In the previous code you can see an example on a `setUp()` method, which allows us to avoid repeating the same initialization code on each `test*()` method.

And, restoring our math tests to a good state, the `test_math.py` will contain the following:

```
import math
import unittest
import operator

class TestMath(unittest.TestCase):
    def testFloor(self):
        self.assertEqual(1, math.floor(1.01))
        self.assertEqual(0, math.floor(0.5))
        self.assertEqual(-1, math.floor(-0.5))
        self.assertEqual(-2, math.floor(-1.1))

    def testCeil(self):
        self.assertEqual(2, math.ceil(1.01))
        self.assertEqual(1, math.ceil(0.5))
        self.assertEqual(0, math.ceil(-0.5))
        self.assertEqual(-1, math.ceil(-1.1))

    def testDivision(self):
        self.assertRaises(ZeroDivisionError, operator.div, 1, 0)
        # The same assertion using a different idiom:
        self.assertRaises(ZeroDivisionError, lambda: 1 / 0)

    def testMultiplication(self):
        self.assertAlmostEqual(0.3, 0.1 * 3)
```


Now, how do we run, in one pass, tests defined in different modules? One option is to manually build a *test suite*. A test suite is a simply collection of test cases (and/or other test suites) which, when ran, will run all the test cases (and/or test suites) contained by it. Note that a new test case instance is built for each test method, so suites have already been build under the hood every time you have run a test module. Our work, then, is to “paste” the suites together.

Let’s build suites using the interactive interpreter!

First, import the involved modules:

```
>>> import unittest, test_math, test_lists
```

Then, we will obtain the test suites for each one of our test modules (which were implicitly created when running them using the `unittest.main()` shortcut), using the `unittest.TestLoader` class:

```
>>> loader = unittest.TestLoader()
>>> math_suite = loader.loadTestsFromModule(test_math)
>>> lists_suite = loader.loadTestsFromModule(test_lists)
```

Now we build a new suite which combine these suites:

```
>>> global_suite = unittest.TestSuite([math_suite, lists_suite])
```

And finally, we run the suite:

```
>>> unittest.TextTestRunner().run(global_suite)
.....
-----
Ran 7 tests in 0.010s

OK
<unittest._TextTestResult run=7 errors=0 failures=0>
```

Or, if you feel like wanting a more verbose output:

```
>>> unittest.TextTestRunner(verbosity=2).run(global_suite)
testCeil (test_math.TestMath) ... ok
testDivision (test_math.TestMath) ... ok
testFloor (test_math.TestMath) ... ok
testMultiplication (test_math.TestMath) ... ok
testContains (test_lists.TestLists) ... ok
testLen (test_lists.TestLists) ... ok
testSort (test_lists.TestLists) ... ok

-----
Ran 7 tests in 0.020s

OK
<unittest._TextTestResult run=7 errors=0 failures=0>
```

Using this low level knowledge about loaders, suites and runner you can easily write a script to run the tests of any project. Obviously, the details of the script will vary from project to project depending the way in which you decide to organize your tests.

On the other hand, typically you won’t write custom scripts to run all your tests. Using test tools which do automatic test discovery will be a much convenient approach. We will look one of them shortly. But first, I must show you other testing tool very popular in the Python world: doctests.

The magic of doctests is that it encourages the inclusion of these examples by doubling them as tests. Let's save our example code as `even.py` and add the following snippet at the end:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Then, run it:

```
$ jython even.py
```

And well, doctests are a bit shy and don't show any output on success. But to convince you that it is indeed testing our code, run it with the `-v` option:

[illegible]

```
7 passed and 0 failed.  
Test passed.
```

Doctests are a very, very convenient way to do testing, since the interactive examples can be directly copy-pasted from the interactive shell, transforming the manual testing in documentation examples and automated tests in one shot.

You don't really *need* to include doctests as part of the documentation of the feature they test. Nothing stops you to write the following code in, say, the `test_math_using_doctest.py` module:

```
"""  
Doctests equivalent to test_math unittests seen in the previous section.  
  
>>> import math  
  
Tests for floor():  
  
>>> math.floor(1.01)  
1  
>>> math.floor(0.5)  
0  
>>> math.floor(-0.5)  
-1  
>>> math.floor(-1.1)  
-2  
  
Tests for ceil():  
  
>>> math.ceil(1.01)  
2  
>>> math.ceil(0.5)  
1  
>>> math.ceil(-0.5)  
0  
>>> math.ceil(-1.1)  
-1  
  
Test for division:  
  
>>> 1 / 0  
Traceback (most recent call last):  
...  
ZeroDivisionError: integer division or modulo by zero  
  
Test for floating point multiplication:  
  
>>> (0.3 - 0.1 * 3) < 0.0000001  
True  
  
"""  
if __name__ == "__main__":  
    import doctest  
    doctest.testmod()
```

One thing to note on the last test in the previous example, is that in some cases doctests are not the most clean way to express a test. Also note that if that test fails you will *not* get useful information from the failure. It will tell you that the output was `False` when `True` was expected, without the extra details that `assertAlmostEquals()` would give you. The morale of the history is to realize that doctest is just another tool in the toolbox, which can fit very well in some cases and not fit well in others.

Warning: Speaking of doctests gotchas: The use of dictionary outputs in doctests is a very common error that breaks the portability of your doctests across Python implementations (e.g. Jython, CPython and IronPython). The trap here is that the **order of dict keys is implementation-dependent**, so the test may pass when working on some implementation and fail horribly on others. The workaround is to convert the dict to a sequence of tuples and sort them, using `sorted(mydict.items())`.

That shows the big downfall of doctests: It always does a textual comparison of the expression, converting the result to string. It isn't aware of the objects structure.

To take advantage of doctests we have to follow some simple rules, like using the `>>>` prompt and leaving a blank line between sample output and the next paragraph. But if you think about it, it's the same kind of sane rules that makes the documentation readable by people.

The only common rule not shown by the examples shown in this section is the way to write expressions which are written in more than one line. As you may expect, you have to follow the same convention used by the interactive interpreter: start the continuation lines with an ellipsis (`"..."`). For example:

```
"""
Addition is commutative:

>>> ((1 + 2) ==
...   (2 + 1))
True
"""
```

A Complete Example

Having seen the two test frameworks used in the Python world, let's see them applied to a more meaningful program. We will write code to check for solutions of the eight-queens chess puzzle. The idea of the puzzle is to place eight queens in a chessboard, with no queen attacking each other. Queens can attack any piece placed in the same row, column or diagonals. The figure [Eight queens solution](#) shows one of the solutions of the puzzle.

I like to use doctests to check the contract of the program with the outside, and unittest for what we could see as the internal tests. I do that because external interfaces tend to be clearly documented, and automated testing of the examples in the documentation is always a great thing. On the other hand, unittests shine on pointing us to the very specific source of a bug, or at the very least on providing more useful debugging information than doctests.

Note: In practice, both type of tests have strengths and weakness, and you may find some cases in which you will prefer the readability and simplicity of doctests and only use them on your project. Or you will favor the granularity and isolation of unittests and only use them on your project. As many things in life, it's a trade-off.

We'll develop this program in a test-driven development fashion. Test will be written first, as a sort of specification for our program, and code will be written later to fulfill the tests requirements.

Let's start by specifying the public interface of our puzzle checker, which will live on the `eightqueen` package. This is the start of the main module, `eightqueen.checker`:

```
"""
eightqueen.checker: Validates solutions for the eight queens puzzle.

Provides the function is_solution(board) to determine if a board represents a
valid solution of the puzzle.

The chess board is represented by list of 8 strings, each string of length
```

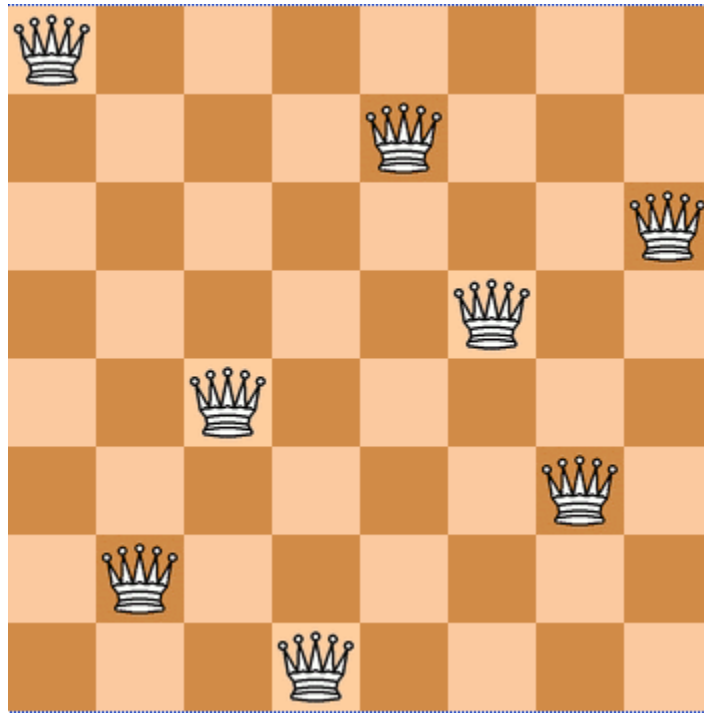


Fig. 5.1: Eight queens solution

8. Positions occupied by a Queen are marked by the character 'Q', and empty spaces are represented by an space character.

Here is a valid board:

```
>>> board = ['Q      ',
...          ' Q     ',
...          '  Q    ',
...          '   Q   ',
...          '    Q  ',
...          '     Q ',
...          '      Q',
...          '       Q']
```

Naturally, it is not a correct solution:

```
>>> is_solution(board)
False
```

Here is a correct solution:

```
>>> is_solution(['Q      ',
...              '     Q ',
...              '    Q  ',
...              '   Q   ',
...              '  Q    ',
...              ' Q     ',
...              '      Q',
...              '       Q'])
True
```

Malformed boards are rejected and a `ValueError` is thrown:

```
>>> is_solution([])
Traceback (most recent call last):
...
ValueError: Malformed board
```

Only 8 x 8 boards are supported.

```
>>> is_solution(['Q   ',
...             ' Q ',
...             ' Q ',
...             '  Q'],)
Traceback (most recent call last):
...
ValueError: Malformed board
```

And they must only contains `Qs` and `spaces`:

```
>>> is_solution(['X   ',
...             '  X ',
...             '   X ',
...             '  X ',
...             ' X  ',
...             '  X ',
...             ' X  ',
...             '  X '],)
Traceback (most recent call last):
...
ValueError: Malformed board
```

And the total number of `Qs` must be eight:

```
>>> is_solution(['QQQQQQQQ',
...             'Q      ',
...             '      ',
...             '      ',
...             '      ',
...             '      ',
...             '      ',
...             '      '],)
Traceback (most recent call last):
...
ValueError: There must be exactly 8 queens in the board
```

```
>>> is_solution(['QQQQQQQ ',
...             '      ',
...             '      ',
...             '      ',
...             '      ',
...             '      ',
...             '      ',
...             '      '],)
Traceback (most recent call last):
...
ValueError: There must be exactly 8 queens in the board
```

```
"""
```

That’s a good start: we know what we have to build. The doctests play the role of a more precise problem statement. Actually, it’s an executable problem statement which can be used to verify our solution to the problem.

Now we will specify the “internal” interface which shows how we can solve the problem of writing the solution checker. It’s a common practice to write the unit tests on a separate module. So here is the code for `eightqueens.test_checker`:

```
import unittest
from eightqueens import checker

BOARD_TOO_SMALL = ['Q' * 3 for i in range(3)]
BOARD_TOO_BIG = ['Q' * 10 for i in range(10)]
BOARD_WITH_TOO_MANY_COLS = ['Q' * 9 for i in range(8)]
BOARD_WITH_TOO_MANY_ROWS = ['Q' * 8 for i in range(9)]
BOARD_FULL_OF_QS = ['Q' * 8 for i in range(8)]
BOARD_FULL_OF_CRAP = [chr(65 + i) * 8 for i in range(8)]
BOARD_EMPTY = [' ' * 8 for i in range(8)]

BOARD_WITH_QS_IN_THE_SAME_ROW = [
    'Q   Q   ',
    '    Q   ',
    '   Q    ',
    '    Q   ',
    '  Q     ',
    '   Q    ',
    '    Q   ',
    '   Q    '
]

BOARD_WITH_WRONG_SOLUTION = BOARD_WITH_QS_IN_THE_SAME_ROW

BOARD_WITH_QS_IN_THE_SAME_COL = [
    'Q     ',
    '   Q   ',
    '    Q  ',
    '  Q    ',
    '   Q   ',
    '    Q  ',
    '  Q    ',
    '   Q   '
]

BOARD_WITH_QS_IN_THE_SAME_DIAG_1 = [
    '      Q',
    '     Q',
    '    Q',
    '   Q',
    '  Q',
    ' Q',
    'Q'
]

BOARD_WITH_QS_IN_THE_SAME_DIAG_2 = [
    '      Q',
    '     Q',
    '    Q',
    '   Q',
    '  Q',
    ' Q',
    'Q'
]

BOARD_WITH_QS_IN_THE_SAME_DIAG_3 = [
    '      Q',
    '     Q',
    '    Q',
    '   Q',
    '  Q',
    ' Q',
    'Q'
]
```



```
BOARD_WITH_QS_IN_THE_SAME_DIAG_3 = [
    '      Q ',
    '     .   ',
    '    .    ',
    '   .     ',
    '  .      ',
    '.       ',
    '        ',
    '       ' ]

BOARD_WITH_QS_IN_THE_SAME_DIAG_4 = [
    '      Q ',
    '     .   ',
    '    .    ',
    '   .     ',
    '  .      ',
    '.       ',
    'Q      ',
    '       ' ]

BOARD_WITH_QS_IN_THE_SAME_DIAG_5 = [
    '      Q ',
    '     .   ',
    '    .    ',
    '   .     ',
    '  .      ',
    '.       ',
    'Q      ',
    '     Q  ',
    '    .   ',
    '   .    ',
    '  .     ',
    '.      ',
    'Q     ' ]

BOARD_WITH_SOLUTION = [
    'Q      ',
    '     .   ',
    '    .    ',
    '   .     ',
    '  .      ',
    '.       ',
    'Q      ',
    '     Q  ',
    '    .   ',
    '   .    ',
    '  .     ',
    '.      ',
    'Q     ' ]


class ValidationTest(unittest.TestCase):
    def testValidateShape(self):
        def assertNotValidShape(board):
            self.assertFalse(checker._validate_shape(board))

        # Some invalid shapes:
        assertNotValidShape([])
        assertNotValidShape(BOARD_TOO_SMALL)
        assertNotValidShape(BOARD_TOO_BIG)
        assertNotValidShape(BOARD_WITH_TOO_MANY_COLS)
        assertNotValidShape(BOARD_WITH_TOO_MANY_ROWS)

        def assertValidShape(board):
            self.assertTrue(checker._validate_shape(board))

        assertValidShape(BOARD_WITH_SOLUTION)
        # Shape validation doesn't care about board contents:
        assertValidShape(BOARD_FULL_OF_QS)
```

```
assertValidShape(BOARD_FULL_OF_CRAP)

def testValidateContents(self):
    # Valid content => only 'Q' and ' ' in the board
    self.assertFalse(checker._validate_contents(BOARD_FULL_OF_CRAP))
    self.assert_(checker._validate_contents(BOARD_WITH_SOLUTION))
    # Content validation doesn't care about the number of queens:
    self.assert_(checker._validate_contents(BOARD_FULL_OF_QS))

def testValidateQueens(self):
    self.assertFalse(checker._validate_queens(BOARD_FULL_OF_QS))
    self.assertFalse(checker._validate_queens(BOARD_EMPTY))
    self.assert_(checker._validate_queens(BOARD_WITH_SOLUTION))
    self.assert_(checker._validate_queens(BOARD_WITH_WRONG_SOLUTION))

class PartialSolutionTest(unittest.TestCase):
    def testRowsOK(self):
        self.assert_(checker._rows_ok(BOARD_WITH_SOLUTION))
        self.assertFalse(checker._rows_ok(BOARD_WITH_QS_IN_THE_SAME_ROW))

    def testColsOK(self):
        self.assert_(checker._cols_ok(BOARD_WITH_SOLUTION))
        self.assertFalse(checker._cols_ok(BOARD_WITH_QS_IN_THE_SAME_COL))

    def testDiagonalsOK(self):
        self.assert_(checker._diagonals_ok(BOARD_WITH_SOLUTION))
        self.assertFalse(
            checker._diagonals_ok(BOARD_WITH_QS_IN_THE_SAME_DIAG_1))
        self.assertFalse(
            checker._diagonals_ok(BOARD_WITH_QS_IN_THE_SAME_DIAG_2))
        self.assertFalse(
            checker._diagonals_ok(BOARD_WITH_QS_IN_THE_SAME_DIAG_3))
        self.assertFalse(
            checker._diagonals_ok(BOARD_WITH_QS_IN_THE_SAME_DIAG_4))
        self.assertFalse(
            checker._diagonals_ok(BOARD_WITH_QS_IN_THE_SAME_DIAG_5))

class SolutionTest(unittest.TestCase):
    def testIsSolution(self):
        self.assert_(checker.is_solution(BOARD_WITH_SOLUTION))

        self.assertFalse(checker.is_solution(BOARD_WITH_QS_IN_THE_SAME_COL))
        self.assertFalse(checker.is_solution(BOARD_WITH_QS_IN_THE_SAME_ROW))
        self.assertFalse(checker.is_solution(BOARD_WITH_QS_IN_THE_SAME_DIAG_5))

        self.assertRaises(ValueError, checker.is_solution, BOARD_TOO_SMALL)
        self.assertRaises(ValueError, checker.is_solution, BOARD_FULL_OF_CRAP)
        self.assertRaises(ValueError, checker.is_solution, BOARD_EMPTY)
```

These unit tests propose a way to solve the problem, decomposing it in two big tasks (input validation and the actual verification of solutions) and each task is decomposed on a smaller portion meant to be implemented by a function. In some way, they are an executable design of the solution.

So we have a mix of doctests and unit tests. How do we run all of them in one shot? Previously I showed you how to manually compose a test suite for unit tests belonging to different modules, so that may be an answer. And indeed, there is a way to add doctests to test suites: `doctest.DocTestSuite(module_with_doctests)`. But, since

we are working on a more real testing example, we will use a real world solution to this problem (as you can imagine, people got tired of the tedious work and more automated solutions appeared).

Nose

Nose is a tool for test discovery and execution. By default, nose tries to run tests on any module whose name starts with “test”. You can override that, of course. In our case, the example code of the previous section follows the convention (the test module is named `eightqueens.test_checker`).

We will use `setuptools` to install nose. Refer to Appendix A for instructions on how to install `setuptools` if you haven’t installed it yet.

Once you have `setuptools` installed, run:

```
$ easy_install nose
```

Note: I’m assuming that the `bin` directory of the Jython installation is on your `PATH`. If it’s not, you will have to explicitly type that path preceding each command like `jython` or `easy_install` with that path (i.e., you will need to type something like `/path/to/jython/bin/easy_install` instead of just `easy_install`)

Once nose is installed, an executable named `nosetests` will appear on the `bin/` directory of your Jython installation. Let’s try it, locating ourselves on the parent directory of `eightqueens` and running:

```
$ nosetests --with-doctest
```

By default nose do *not* run doctests, so we have to explicitly enable the doctest plugin that comes built in with nose.

Back to our example, here is the shortened output after running nose:

```
FEFFFFFFE

[Snipped output]

-----
Ran 8 tests in 1.133s
FAILED (errors=7, failures=1)
```

Of course all of our tests (6 unit tests and 1 doctest) failed. It’s time to fix that. But first, let’s run nose again *without* the doctests, since we will follow the unit tests to construct the solution. And we know that as long as our unit tests fail, the doctest will also likely fail. Once all unit tests pass, we can check our whole program against the high level doctest and see if we missed something or did it right. Here is the nose output for the unit tests:

```
$ nosetests
EEEEEEEE
=====
ERROR: testIsSolution (eightqueens.test_checker.SolutionTest)
-----
Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 149, in testIsSolution
    self.assert_(checker.is_solution(BOARD_WITH_SOLUTION))
AttributeError: 'module' object has no attribute 'is_solution'

=====
ERROR: testColsOK (eightqueens.test_checker.PartialSolutionTest)
-----
Traceback (most recent call last):
```

```

File "/path/to/eightqueens/test_checker.py", line 100, in testColsOK
    self.assert_(checker._cols_ok(BOARD_WITH_SOLUTION))
AttributeError: 'module' object has no attribute '_cols_ok'

=====
ERROR: testDiagonalsOK (eightqueens.test_checker.PartialSolutionTest)
-----

Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 104, in testDiagonalsOK
    self.assert_(checker._diagonals_ok(BOARD_WITH_SOLUTION))
AttributeError: 'module' object has no attribute '_diagonals_ok'

=====
ERROR: testRowsOK (eightqueens.test_checker.PartialSolutionTest)
-----

Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 96, in testRowsOK
    self.assert_(checker._rows_ok(BOARD_WITH_SOLUTION))
AttributeError: 'module' object has no attribute '_rows_ok'

=====
ERROR: testValidateContents (eightqueens.test_checker.ValidationTest)
-----

Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 81, in testValidateContents
    self.assertFalse(checker._validate_contents(BOARD_FULL_OF_CRAP))
AttributeError: 'module' object has no attribute '_validate_contents'

=====
ERROR: testValidateQueens (eightqueens.test_checker.ValidationTest)
-----

Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 88, in testValidateQueens
    self.assertFalse(checker._validate_queens(BOARD_FULL_OF_QS))
AttributeError: 'module' object has no attribute '_validate_queens'

=====
ERROR: testValidateShape (eightqueens.test_checker.ValidationTest)
-----

Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 65, in testValidateShape
    assertNotValidShape([])
  File "/path/to/eightqueens/test_checker.py", line 62, in assertNotValidShape
    self.assertFalse(checker._validate_shape(board))
AttributeError: 'module' object has no attribute '_validate_shape'

-----

Ran 7 tests in 0.493s

FAILED (errors=7)

```

Let's start clearing the failures by coding the validation functions specified by the `ValidationTest`. That is, the `_validate_shape()`, `_validate_contents()` and `validate_queens()` functions, in the `eightqueens.checker` module:

```

def _validate_shape(board):
    return (board and
            len(board) == 8 and

```

```

        all(len(row) == 8 for row in board))

def _validate_contents(board):
    for row in board:
        for square in row:
            if square not in ('Q', ' '):
                return False
        return True

def _count_queens(row):
    n = 0
    for square in row:
        if square == 'Q':
            n += 1
    return n

def _validate_queens(board):
    n = 0
    for row in board:
        n += _count_queens(row)
    return n == 8

```

And now run nose again:

```

$ nosetests

EEEE...
=====
ERROR: testIsSolution (eightqueens.test_checker.SolutionTest)
-----
Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 149, in testIsSolution
    self.assert_(checker.is_solution(BOARD_WITH_SOLUTION))
AttributeError: 'module' object has no attribute 'is_solution'

=====
ERROR: testColsOK (eightqueens.test_checker.PartialSolutionTest)
-----
Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 100, in testColsOK
    self.assert_(checker._cols_ok(BOARD_WITH_SOLUTION))
AttributeError: 'module' object has no attribute '_cols_ok'

=====
ERROR: testDiagonalsOK (eightqueens.test_checker.PartialSolutionTest)
-----
Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 104, in testDiagonalsOK
    self.assert_(checker._diagonals_ok(BOARD_WITH_SOLUTION))
AttributeError: 'module' object has no attribute '_diagonals_ok'

=====
ERROR: testRowsOK (eightqueens.test_checker.PartialSolutionTest)
-----
Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 96, in testRowsOK
    self.assert_(checker._rows_ok(BOARD_WITH_SOLUTION))
AttributeError: 'module' object has no attribute '_rows_ok'

```

```
-----  
Ran 7 tests in 0.534s
```

```
FAILED (errors=4)
```

We passed all the validation tests! Now we should implement the functions `_rows_ok()`, `_cols_ok()` and `_diagonals_ok()` to pass `PartialSolutionTest`:

```
def _scan_ok(board, coordinates):  
    queen_already_found = False  
    for i, j in coordinates:  
        if board[i][j] == 'Q':  
            if queen_already_found:  
                return False  
            else:  
                queen_already_found = True  
    return True  
  
def _rows_ok(board):  
    for i in range(8):  
        if not _scan_ok(board, [(i, j) for j in range(8)]):  
            return False  
    return True  
  
def _cols_ok(board):  
    for j in range(8):  
        if not _scan_ok(board, [(i, j) for i in range(8)]):  
            return False  
    return True  
  
def _diagonals_ok(board):  
    for k in range(8):  
        # Diagonal: (0, k), (1, k + 1), ..., (7 - k, 7)...  
        if not _scan_ok(board, [(i, k + i) for i in range(8 - k)]):  
            return False  
        # Diagonal: (k, 0), (k + 1, 1), ..., (7, 7 - k)  
        if not _scan_ok(board, [(k + j, j) for j in range(8 - k)]):  
            return False  
  
        # Diagonal: (0, k), (1, k - 1), ..., (k, 0)  
        if not _scan_ok(board, [(i, k - i) for i in range(k + 1)]):  
            return False  
  
        # Diagonal: (7, k), (6, k - 1), ..., (k, 7)  
        if not _scan_ok(board, [(7 - j, k + j) for j in range(8 - k)]):  
            return False  
    return True
```

Let's try nose again:

```
$ nosetests  
  
...E...  
=====  
ERROR: testIsSolution (eightqueens.test_checker.SolutionTest)  
-----
```

```

Traceback (most recent call last):
  File "/path/to/eightqueens/test_checker.py", line 149, in testIsSolution
    self.assert_(checker.is_solution(BOARD_WITH_SOLUTION))
AttributeError: 'module' object has no attribute 'is_solution'

-----
Ran 7 tests in 0.938s

FAILED (errors=1)

```

Finally, we have to assemble the pieces together to pass the test for `is_solution()`:

```

def is_solution(board):
    if not _validate_shape(board) or not _validate_contents(board):
        raise ValueError("Malformed board")
    if not _validate_queens(board):
        raise ValueError("There must be exactly 8 queens in the board")
    return _rows_ok(board) and _cols_ok(board) and _diagonals_ok(board)

```

And we can hope that all test pass now:

```

$ nosetests

.....
-----
Ran 7 tests in 0.592s

OK

```

Indeed, they all pass. Moreover, we probably also pass the “problem statement”, test, expressed in our doctest:

```

$ nosetests --with-doctest

.....
-----
Ran 8 tests in 1.523s

OK

```

Objective accomplished! We have come up with a nicely documented and tested module, using the two testing tools shipped with the Python language, and Nose to run all our tests without manually building suites.

Integration with Java?

You may be wondering how to integrate the testing frameworks of Python and Java. It is possible to write JUnit tests in Jython, but it’s not really interesting, considering that you can test Java classes using unittest and doctest. The following is a perfectly valid doctest:

```

"""
Tests for Java's DecimalFormat

>>> from java.text import DecimalFormat

A format for money:

>>> dolarFormat = DecimalFormat("$ ###,###.##")

```

```
The decimal part is only printed if needed:

>>> dolarFormat.format(1000)
u'$ 1.000'

Rounding is used when there are more decimal numbers than those defined by the
format:

>>> dolarFormat.format(123456.789)
u'$ 123.456,79'

The format can be used as a parser:

>>> dolarFormat.parse('$ 123')
123L

The parser ignores the unparseable text after the number:

>>> dolarFormat.parse("$ 123abcd")
123L

However, if it can't parse a number, it throws a ParseException:

>>> dolarFormat.parse("abcd")
Traceback (most recent call last):
...
ParseException: java.text.ParseException: Unparseable number: "abcd"
"""
```

So you can use all what you learned on this chapter to test code written in Java. Personally, I find this a very powerful tool for Java development: easy, flexible and unceremonious testing using Jython and Python testing tools!

Continuous Integration

Martin Fowler defines Continuous Integration as “a software development practice where members of a team integrate their work frequently [...]. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible”. Some software development teams report to have used this practice as early as in the 1960, however it only became mainstream when advocated as part of the Extreme Programming practices. Nowadays, it is a widely applied practice, and in the Java world there is a wealth of tools to help with the technical challenge involved by it.

Hudson

One tool that currently has a lot of momentum, growing a important user base is Hudson. Among its prominent features are the ease of installation and configuration, and the ease to deploy it in a distributed, master/slaves environment for cross-platform testing.

But, in my opinion, Hudson’s main strength is its highly modular, plugin-based architecture, which has resulted in the creation of plugins to support most of the version control, build and reporting tools, and many languages. One of them is the Jython plugin, which allows you to use the Python language to drive your builds.

You can find a more details about the Hudson project on its homepage at <https://hudson.dev.java.net/>. I will go to the point and show how to test Jython applications using it.

Getting Hudson

Grab the latest version of Hudson from <http://hudson-ci.org/latest/hudson.war>. You can deploy it to any servlet container like Tomcat or Glassfish. But one of the cool features of Hudson is that you can test it by simply running:

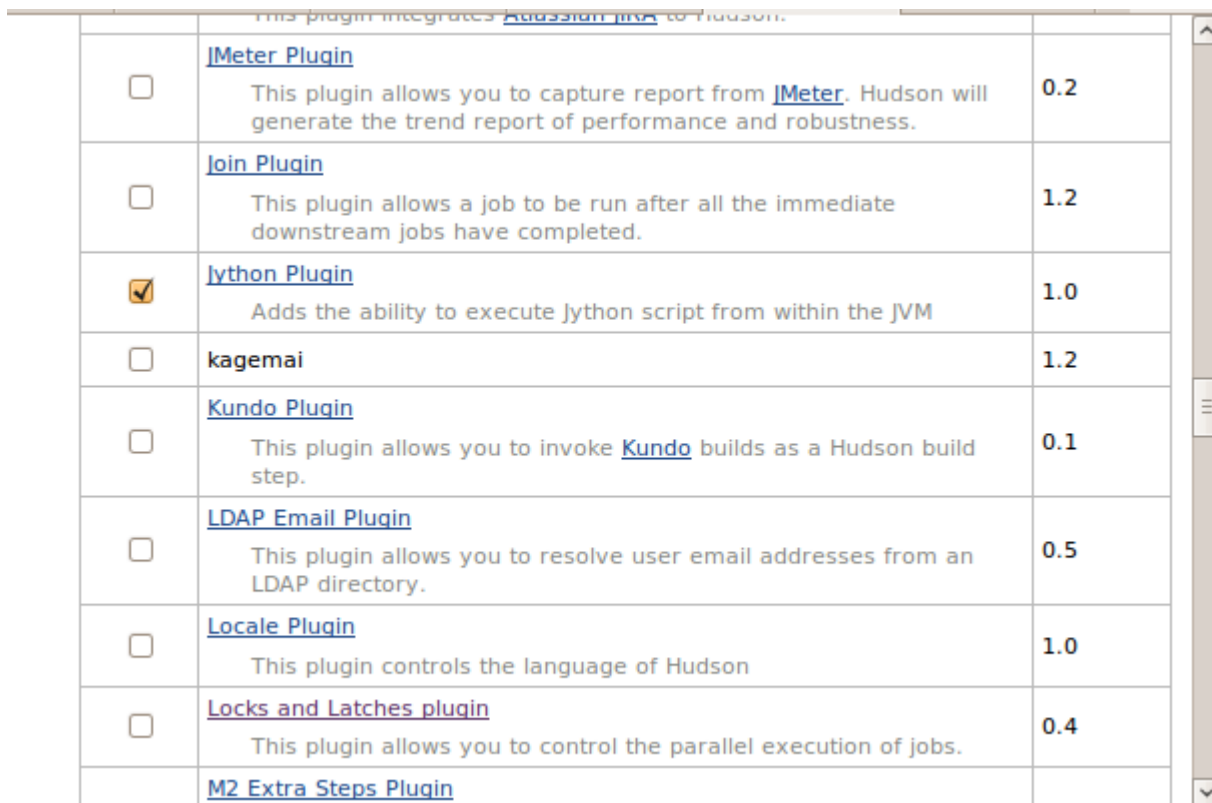
```
$ java -jar hudson.war
```

After a few seconds, you will see some logging output on the console, and Hudson will be up and running. If you visit <http://localhost:8080/> you will get a welcome page inviting you to start using Hudson creating new jobs. .. warning:

Be careful: The default mode of operation of Hudson fully trusts its users, letting them to execute **any** command they want on the server, **with** the privileges of the user running Hudson. You can **set** stricter access control policies on the "Configure System" section of the "Manage Hudson" page.

Installing the Jython Plugin

Before creating jobs, we will install the Jython plugin. Click on the "Manage Hudson" link on the left side menu. Then click "Manage Plugins". Now go to the "Available" tab. You will see a very long list of plugins (I told you this was the greatest Hudson strength!). Find the "Jython Plugin", click on the checkbox at its left, as shown on the figure *Selecting the Jython Plugin*. then scroll to the end of the page and click the "Install" button.



The screenshot shows the 'Available' tab of the 'Manage Plugins' page in Hudson. A table lists various plugins with checkboxes for selection. The 'Jython Plugin' is selected, indicated by a checked checkbox. The table includes columns for the plugin name, description, and version number.

Checkbox	Plugin Name	Description	Version
<input type="checkbox"/>	JMeter Plugin	This plugin allows you to capture report from JMeter . Hudson will generate the trend report of performance and robustness.	0.2
<input type="checkbox"/>	Join Plugin	This plugin allows a job to be run after all the immediate downstream jobs have completed.	1.2
<input checked="" type="checkbox"/>	Jython Plugin	Adds the ability to execute jython script from within the JVM	1.0
<input type="checkbox"/>	kagemai		1.2
<input type="checkbox"/>	Kundo Plugin	This plugin allows you to invoke Kundo builds as a Hudson build step.	0.1
<input type="checkbox"/>	LDAP Email Plugin	This plugin allows you to resolve user email addresses from an LDAP directory.	0.5
<input type="checkbox"/>	Locale Plugin	This plugin controls the language of Hudson	1.0
<input type="checkbox"/>	Locks and Latches plugin	This plugin allows you to control the parallel execution of jobs.	0.4
<input type="checkbox"/>	M2 Extra Steps Plugin		

Fig. 5.2: Selecting the Jython Plugin.

You will see a bar showing the progress of the download and installation progress, and after little while you will be presented with an screen like shown on the figure *Jython Plugin Successfully Installed* notifying you that the process

finished. Press the “Restart” button, wait a little bit and you will see the welcome screen again. Congratulations, you now have a Jython-powered Hudson!



Fig. 5.3: Jython Plugin Successfully Installed

Creating a Hudson Job for a Jython Project

Let’s follow now the suggestion of the welcome screen and click the “create new job” link. A job roughly corresponds to the instructions needed by Hudson to build a project. It includes:

- The location from where the source code of the project should be obtained, and how often.
- How to start the build process for the project
- How to collect information after the build process has finished

After clicking the “create new job” link (equivalent to the “New Job” entry on the left side menu) you will be asked for a name and type for the Job. We will use the eightqueens project built on the previous section, so name the project “eightqueens”, select the “Build a free-style software project” option and press the “OK” button.

In the next screen, we need to setup an option on the “Source Code Management” section. You may want to experiment with your own repositories here (by default only CVS and Subversion are supported, but there are plugins for all the other VCSs used out there). For our example, I’ve hosted the code on a Subversion repository at <http://kenai.com/svn/jythonbook~eightqueens/>. So select “Subversion” and enter <http://kenai.com/svn/jythonbook~eightqueens/trunk/eightqueens/> as the “Repository URL”.

Note: Using the public repository will be enough to get a feeling of Hudson and its support of Jython. However, I encourage you to create your own repository so you can play freely with continuous integration, for example committing bad code to see how failures are handled.

In the “Build Triggers” section we have to specify when automated builds will happen. We will poll the repository so that a new build will be started after any change. Select “Poll SCM” and enter “@hourly” on the “Schedule” box (If you want to know all the options for the schedule, click the help icon at the right of the box).

In the “Build” section we must tell Hudson how to build our project. By default Hudson supports Shell scripts, Windows Batch files and Ant scripts as build steps. For projects in which you mix Java and Python code and drive the build process with an ant file, the default Ant build step will suffice. In our case, we wrote our app in pure Python code, so we will use the Jython plugin which adds the “Execute Jython script” build step.

So click on “Add Build Step” and then select “Execute Jython script”. We will use our knowledge of test suites gained on the *UnitTest* section, the following script will be enough to run our tests:

```
import os, sys, unittest, doctest
from eightqueens import checker, test_checker

loader = unittest.TestLoader()
suite = unittest.TestSuite([loader.loadTestsFromModule(test_checker),
                        doctest.DocTestSuite(checker)])
result = unittest.TextTestRunner().run(suite)
print result
if not result.wasSuccessful():
    sys.exit(1)
```

The figure *Hudson Job Configuration* shows how the page looks so far for the “Source Code Management”, “Build Triggers” and “Build” sections.

The screenshot displays the Hudson Job Configuration interface, divided into three main sections:

- Source Code Management:**
 - Repository type: ☒ Subversion
 - Repository URL:
 - Local module directory (optional):
 - Use update: ☒
 - Repository browser: (Auto)
- Build Triggers:**
 - ☐ Build after other projects are built
 - ☐ Build periodically
 - ☒ Poll SCM
 - Schedule:
- Build:**
 - Build step: **Execute Jython script**
 - Script:

```
loader = unittest.TestLoader()
suite = unittest.TestSuite([loader.loadTestsFromModule(test_checker),
                        doctest.DocTestSuite(checker)])
result = unittest.TextTestRunner().run(suite)
if not result.wasSuccessful():
    sys.exit(1)
```

Buttons for "Add build step", "Delete", and "Advanced..." are visible at the bottom of the configuration area.

Fig. 5.4: Hudson Job Configuration

The next section, titled “Post-build Actions” let you specify action to carry once the build has finished, ranging from collecting results from reports generated by static-analysis tools or test runners to send emails notifying someone of build breakage. We will left these options blank by now. Click the “Save” button at the bottom of the page.

At this point Hudson will show the job's main page. But it won't contain anything useful, since Hudson is waiting for the hourly trigger to poll the repository and kick the build. But we don't need to wait if we don't want to: just click the "Build Now" link on the left-side menu. Shortly, a new entry will be shown on the "Build History" box (also on the left side, below the menu), as shown in the figure *The First Build of our First Job*.



Fig. 5.5: The First Build of our First Job.

If you click on the link that just appeared there you will be directed to the page for the build we just made. If you click on the "Console Output" link on the left side menu you will see what's shown in the figure *Console Output for the Build*.

As you would expect, it shows that our eight tests (remember that we had seven unit tests and the module doctest) all passed.

Using Nose on Hudson

You may be wondering why we crafted a custom build script instead of using nose, since *I* stated that using nose was much better than manually creating suites.

The problem is that the Jython runtime provided by the Jython Hudson plugin comes without any extra library, so we can't assume the existence of nose. One option would be to include nose with the source tree on the repository, but it is not convenient.

One way to overcome the problem is to script the installation of nose on the build script. Go back to the Job (also called "Project" by the Hudson user interface), select "Configure" on the left side menu, go to the "Build" section of the configuration and change the Jython script for our job to:

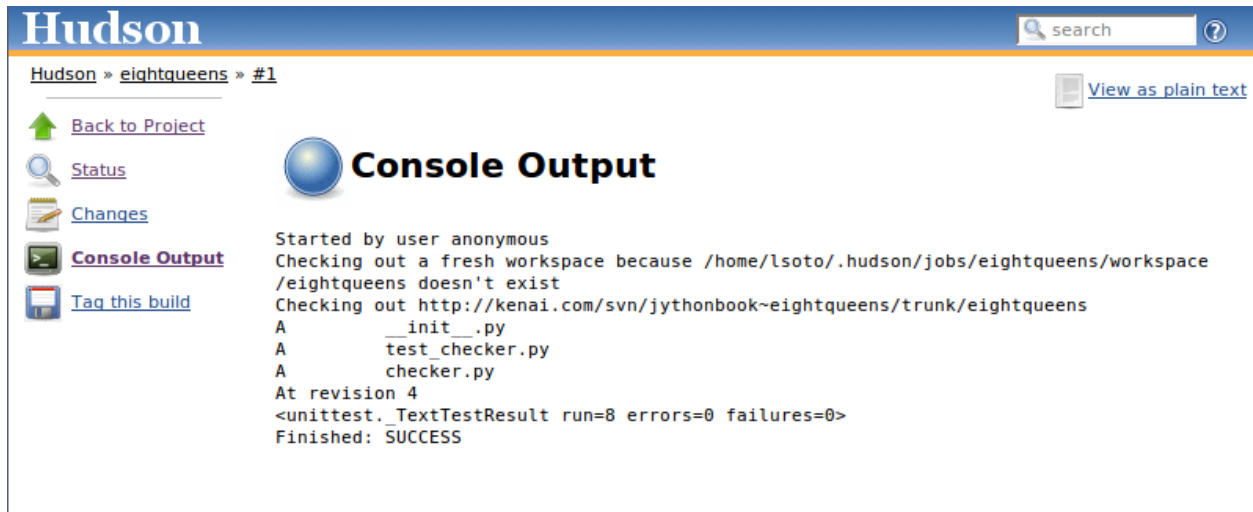


Fig. 5.6: Console Output for the Build

```
# Setup the environment
import os, sys, site, urllib2, tempfile
print "Base dir", os.getcwd()
site_dir = os.path.join(os.getcwd(), 'site-packages')
if not os.path.exists(site_dir): os.mkdir(site_dir)
site.addsitedir(site_dir)
sys.executable = ''
os.environ['PYTHONPATH'] = ' '.join(sys.path)

# Get ez_setup:
ez_setup_path = os.path.join(site_dir, 'ez_setup.py')
if not os.path.exists(ez_setup_path):
    f = file(ez_setup_path, 'w')
    f.write(urllib2.urlopen('http://peak.telecommunity.com/dist/ez_setup.py').read())
    f.close()

# Install nose if not present
try:
    import nose
except ImportError:
    import ez_setup
    ez_setup.main(['--install-dir', site_dir, 'nose'])
    for mod in sys.modules.keys():
        if mod.startswith('nose'):
            del sys.modules[mod]
    for path in sys.path:
        if path.startswith(site_dir):
            sys.path.remove(site_dir)
    site.addsitedir(site_dir)
    import nose

# Run Tests!
nose.run(argv=['nosetests', '-v', '--with-doctest', '--with-xunit'])
```

The first half of the script is plumbing to download setuptools (ez_setup) and set an environment in which it will work. Then, we check for the availability of nose, and if it's not present we install it using setuptools.

The interesting part if the last line:

```
nose.run(argv=['nosetests', '-v', '--with-doctest', '--with-xunit'])
```

Here we are invoking nose from python code, but using the command line syntax. Note the usage of the `--with-xunit` option. It generates JUnit-compatible XML reports for our tests, which can be read by Hudson to generate very useful test reports. By default, nose will generate a file called `nosetests.xml` on the current directory.

To let Hudson know where the report can be found scroll to the “Post Build Actions” section in the configuration, check the “Publish JUnit test result reports” and enter “nosetests.xml” on the “Test Report XMLs” input box. Press “Save”. If Hudson points you that `nosetests.xml` “doesn’t match anything”, don’t worry and just press “Save” again. Of course it doesn’t match anything *yet* since we haven’t run the build again.

Trigger the build again, and after the build is finished, click on the link for it (on the “Build History” box or going to the job page and following the “Last build [...]” permalink). The figure *Nose’s Output on Hudson* shows what you see if you look at the “Console Output” and the figure *Hudson’s Test Reports* what you see on the “Test Results” page.

Navigation on your test results is a very powerful feature of Hudson. But it shines when you have failures or tons of tests, which is not the case on this example. But I wanted to show it in action, so I fabricated some failures on the code to show you some screenshots. Look at figure *Test Report Showing Failures* and figure *Graph of Test Results Over Time* to get an idea of what you get from Hudson.

We had to use a slightly more complicated script to use Nose and Hudson together, but it has the advantage that it will probably work untouched for a long time, unlike the original script manually built the suite, which would have to be modified each time a new test module is created.

Conclusion

Testing is a fertile ground for Jython usage, since you can exploit the flexibility of Python to write concise tests for Java APIs which also tend to be more readable than the ones written with JUnit. Doctests, in particular don’t have a parallel on the Java world and can be a powerful way to introduce the practice of automated testing on people who want it to be simple and easy.

Integration with continuous integration tools, and Hudson in particular let’s you get the maximum from your tests, avoiding test breakages to go unnoticed and representing a live history of your project health and evolution.

Chapter 19: Concurrency

Supporting concurrency is increasingly important. In the past, mainstream concurrent programming generally meant ensuring that the code interacting with relatively slow network, disk, database, and other I/O resources did not unduly slow things down. Exploiting parallelism was typically only seen in such domains as scientific computing with the apps running on supercomputers.

But there are new factors at work now. The semiconductor industry continues to work feverishly to uphold Moore’s Law of exponential increase in chip density. Chip designers used to apply this bounty to speeding up an individual CPU. But for a variety of reasons, this old approach no longer works as well. So now chip designers are cramming chips with more CPUs and hardware threads. Speeding up execution means harnessing the parallelism of the hardware. And it is now our job as software developers to do that work.

The Java platform can help out here. The Java platform is arguably the most robust environment for running concurrent code today, and this functionality can be readily be used from Jython. The problem remains that writing concurrent code is still not easy. This difficulty is especially true with respect to a concurrency model based on threads, which is what today’s hardware natively exposes.

Console Output

```

Started by user anonymous
Updating http://kenai.com/svn/jythonbook~eightqueens/trunk/eightqueens
At revision 8
no change for http://kenai.com/svn/jythonbook~eightqueens/trunk/eightqueens since the previous build
Base dir /home/lsoto/.hudson/jobs/eightqueens/workspace
Downloading http://pypi.python.org/packages/2.5/s/setuptools/setuptools-0.6c9-py2.5.egg
Creating /home/lsoto/.hudson/jobs/eightqueens/workspace/site-packages/site.py
Searching for nose
Reading http://pypi.python.org/simple/nose/
Reading http://somethingaboutorange.com/mrl/projects/nose/
Best match: nose 0.11.1
Downloading http://somethingaboutorange.com/mrl/projects/nose/nose-0.11.1.tar.gz
Processing nose-0.11.1.tar.gz
Running nose-0.11.1/setup.py -q bdist_egg --dist-dir /tmp/easy_install-0glD0D/nose-0.11.1/egg-dist-tmp-kYpY2p
no previously-included directories found matching 'doc/.build'
Adding nose 0.11.1 to easy-install.pth file
Installing nosetests-2.5 script to /home/lsoto/.hudson/jobs/eightqueens/workspace/site-packages
Installing nosetests script to /home/lsoto/.hudson/jobs/eightqueens/workspace/site-packages

Installed /home/lsoto/.hudson/jobs/eightqueens/workspace/site-packages/nose-0.11.1-py2.5.egg
Processing dependencies for nose
Finished processing dependencies for nose
Processing setuptools-0.6c9-py2.5.egg
Copying setuptools-0.6c9-py2.5.egg to /home/lsoto/.hudson/jobs/eightqueens/workspace/site-packages
Adding setuptools 0.6c9 to easy-install.pth file
Installing easy_install script to /home/lsoto/.hudson/jobs/eightqueens/workspace/site-packages
Installing easy_install-2.5 script to /home/lsoto/.hudson/jobs/eightqueens/workspace/site-packages

Installed /home/lsoto/.hudson/jobs/eightqueens/workspace/site-packages/setuptools-0.6c9-py2.5.egg
Processing dependencies for setuptools==0.6c9
Finished processing dependencies for setuptools==0.6c9
Doctest: eightqueens.checker ... ok
testColsOK (eightqueens.test_checker.PartialSolutionTest) ... ok
testDiagonalsOK (eightqueens.test_checker.PartialSolutionTest) ... ok
testRowsOK (eightqueens.test_checker.PartialSolutionTest) ... ok
testIsSolution (eightqueens.test_checker.SolutionTest) ... ok
testValidateContents (eightqueens.test_checker.ValidationTest) ... ok
testValidateQueens (eightqueens.test_checker.ValidationTest) ... ok
testValidateShape (eightqueens.test_checker.ValidationTest) ... ok

-----
XML: nosetests.xml
-----
Ran 8 tests in 0.590s

OK
Recording test results
Finished: SUCCESS

```

Fig. 5.7: Nose's Output on Hudson

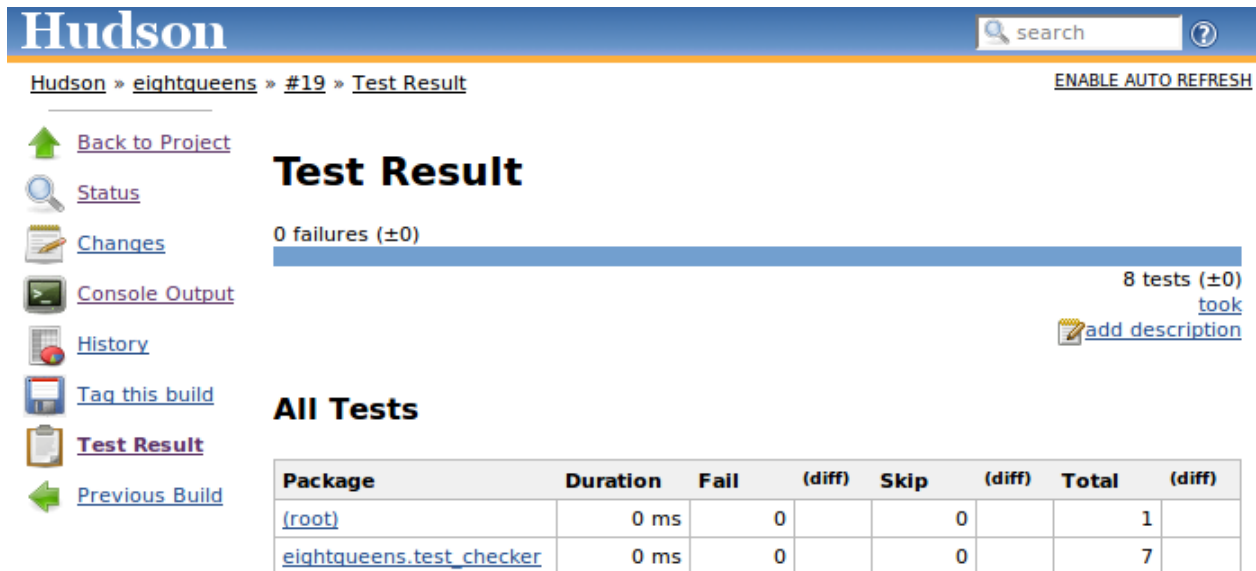


Fig. 5.8: Hudson's Test Reports

This means we have to be concerned with thread safety, which arises as an issue because of the existence of mutable objects that are shared between threads. (Mutable state might be avoidable in functional programming, but it would be hard to avoid in any but the most trivial Python code.) And if you attempt to solve concurrency issues through synchronization, you run into other problems. Besides the potential performance hit, there are opportunities for deadlock and livelock.

Note: Implementations of the JVM, like HotSpot, can often avoid the overhead of synchronization. We will discuss what's necessary for this scenario to happen later in this chapter.

Given all of these issues, it has been argued that threading is just too difficult to get right. We're not convinced by that argument. Useful concurrent systems have been written on the JVM, and that includes apps written in Jython. Key success factors in writing such code include:

- Keep the concurrency simple.
- Use tasks, which can be mapped to a thread pool.
- Use immutable objects where possible.
- Avoid unnecessary sharing of mutable objects.
- Minimize sharing of mutable objects. Queues and related objects – like synchronization barriers – provide a structured mechanism to hand over objects between threads. This can enable a design where an object is visible to only one thread when its state changes.
- Code defensively. Make it possible to cancel or interrupt tasks. Use timeouts.

Java or Python APIs?

One issue that you will have to consider in writing concurrent code is how much to make your implementation dependent on the Java platform. Here are our recommendations:

Test Result

2 failures (±0)

8 tests (±0)

[took](#) [add description](#)

All Failed Tests

Test Name	Duration	Age
<<< eightqueens.eightqueens.checker Stack Trace Traceback (most recent call last): File "/home/lsoto/.hudson/plugins/jython/WEB-INF/lib/jython-2.5.0.jar Lib/unittest\$py.class", line 260, in run File "/home/lsoto/.hudson/plugins/jython/WEB-INF/lib/jython-2.5.0.jar Lib/doctest\$py.class", line 2138, in runTest AssertionError: Failed doctest test for eightqueens.checker File "/home/lsoto/.hudson/jobs/test/workspace/eightqueens/checker.py", line 0, in checker ----- File "/home/lsoto/.hudson/jobs/test/workspace/eightqueens/checker.py", line 96, in eightqueens.checker Failed example: True Expected: False Got: True >>> eightqueens.test_checker.SolutionTest.eightqueens.test_checker.SolutionTest.testIsSolution	0.0	2
	0.0	9

All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Total	(diff)
(root)	0 ms	1		0		1	
eightqueens.test_checker	0 ms	1		0		7	

[Hudson ver. 1.320](#)

Fig. 5.9: Test Report Showing Failures

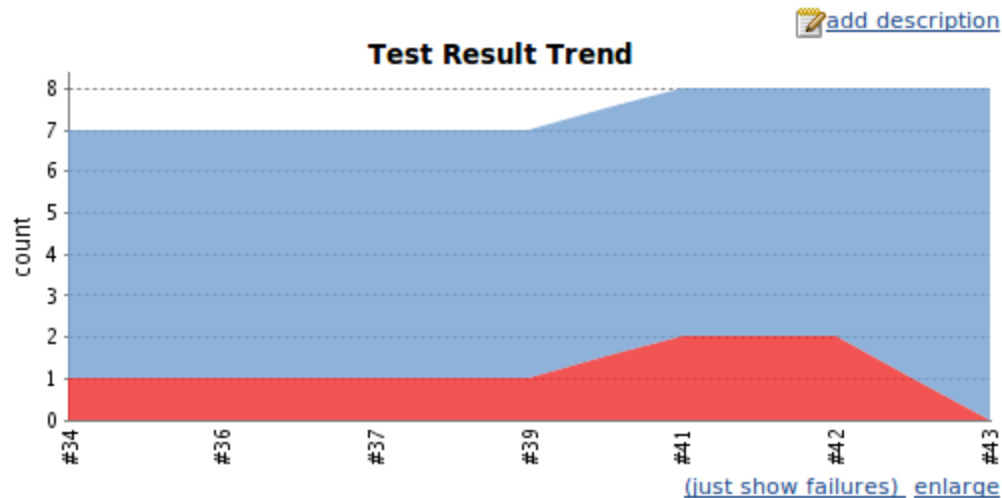


Fig. 5.10: Graph of Test Results Over Time

- If you are porting an existing Python code base that uses concurrency, you can just use the standard Python `threading` module. Such code can still interoperate with Java, because Jython threads are always mapped to Java threads. (If you are coming from Java, you will recognize this API, since it is substantially based on Java's.)
- Jython implements `dict` and `set` by using Java's `ConcurrentHashMap`. This means you can just use these standard Python types, and still get high performance concurrency. (They are also atomic like in CPython, as we will describe.)
- You can also any use of the collections from `java.util.concurrent`. So if it fits your app's needs, you may want to consider using such collections as `CopyOnWriteArrayList` and `ConcurrentSkipListMap` (new in Java 6). The [Google Collections Library](#) is another good choice that works well with Jython.
- Use higher-level primitives from Java instead of creating your own. This is particular true of the executor services for running and managing tasks against thread pools. So for example, avoid using `threading.Timer`, because you can use timed execution services in its place. But still use `threading.Condition` and `threading.Lock`. In particular, these constructs have been optimized to work in the context of a with-statement, as we will discuss.

In practice, using Java's support for higher level primitives should not impact the portability of your code so much. Using tasks in particular tends to keep all of this well-isolated. And such thread safety considerations as thread confinement and safe publication remain the same.

Lastly, remember you can always mix and match.

Working with Threads

Creating threads is easy, perhaps too easy. This example downloads a web page concurrently:

```
.. literalinclude:: src/chapter19/test_thread_creation.py
```

Be careful not to inadvertently invoke the function; `target` takes a reference to the function object (typically a name if a normal function). Calling the function instead creates an amusing bug where your target function runs now, so everything looks fine at first. But no concurrency is happening, because the function call is actually being run by the invoking thread, not this new thread.

The target function can be a regular function, or an object that is callable (implements `__call__`). This later case can make it harder to see that the target is a function object!

To wait for a thread to complete, call `join` on it. This enables working with the concurrent result. The only problem is getting the result. As we will see, publishing results into variables is safe in Jython, but it's not the nicest way either.

Daemon Threads

Daemon threads present an alluring alternative to managing the lifecycle of threads. A thread is set to be a daemon thread before it is started:

```
# create a thread t
t.setDaemon(True)
t.start()
```

Daemon status is inherited by any child threads. Upon JVM shutdown, any daemon threads are simply terminated, without an opportunity – or need – to perform cleanup or orderly shutdown.

This lack of cleanup means it's important that daemon threads never hold any external resources, such as database connections or file handles. Any such resource will not be properly closed upon a JVM exit. For similar reasons, a daemon thread should never make an import attempt, as this can interfere with Jython's orderly shutdown.

In production, the only use case for daemon threads is when they are strictly used to work with in-memory objects, typically for some sort of housekeeping. For example, you might use them to maintain a cache or compute an index.

Having said that, daemon threads are certainly convenient when playing around with some ideas. Maybe your lifecycle management of a program is to use "Control-C" to terminate. Unlike regular threads, running daemon threads won't get in the way and prevent JVM shutdown. Likewise, a latter example demonstrating deadlock uses daemon threads to enable shutdown without waiting on these deadlocked threads.

With that in mind, it's generally best not use daemon threads. At the very least, serious thought should be given to their usage.

Thread Locals

The `threading.local` class enables a letting each thread have its own instances of some objects in an otherwise shared environment. Its usage is deceptively simple. Simply create an instance of `threading.local`, or a subclass, and assign it to a variable or other name. This variable could be global, or part of some other namespace. So far, this is just like working with any other object in Python.

Threads then can share the variable, but with a twist: each thread will see a different, thread-specific version of the object. This object can have arbitrary attributes added to it, each of which will not be visible to other threads.

Other options include subclassing `threading.local`. As usual, this allows you to define defaults and specify a more nuanced properties model. But one unique, and potentially useful, aspect is that any attributes specified in `__slots__` will be *shared* across threads.

However, there's a big problem when working with thread locals. Usually they don't make sense because threads are not the right scope. An object or a function is, especially through a closure. If you are using thread locals, you are implicitly adopting a model where threads are partitioning the work. But then you are binding the given piece of work to a thread. This makes using a thread pool problematic, because you have to clean up after the thread.

Having said they, thread locals might be useful in certain cases. One common scenario is one where your code is being called by a component that you didn't write. You may need to access a thread-local singleton. And of course, if you are using code whose architecture mandates thread locals, it's just something you will have to work with.

But often this is unnecessary. Your code may be different, but Python gives you good tools to avoid action at a distance. You can use closures, decorators, even sometimes selectively monkey patching modules. Take advantage of the fact that Python is a dynamic language, with strong support for metaprogramming. And remember that the Jython implementation makes these techniques accessible when working with even recalcitrant Java code.

In the end, thread locals are an interesting aside. They do not work well in a task-oriented model, because you don't want to associate context with a worker thread that will be assigned to arbitrary tasks. Without a lot of care, this can make for a confused mess.

No Global Interpreter Lock

Jython lacks the global interpreter lock (GIL), which is an implementation detail of CPython. For CPython, the GIL means that only one thread *at a time* can run Python code. This restriction also applies to much of the supporting runtime as well as extension modules that do not release the GIL. (Unfortunately development efforts to remove the GIL in CPython have so far only had the effect of slowing down Python execution significantly.)

The impact of the GIL on CPython programming is that threads are not as useful as they are in Jython. Concurrency will only be seen in interacting with I/O as well as scenarios where computation is performed by an extension module on data structures managed outside of CPython's runtime. Instead, developers typically will use a process-oriented model to evade the restrictiveness of the GIL.

Again, Jython does not have the straightjacket of the GIL. This is because all Python threads are mapped to Java threads and use standard Java garbage collection support (the main reason for the GIL in CPython is because of the reference counting GC system). The important ramification here is that you can use threads for compute-intensive tasks that are written in Python.

Module Import Lock

Python does, however, define a *module import lock*, which is implemented by Jython. This lock is acquired whenever an import of any name is made. This is true whether the import goes through the import statement, the equivalent `__import__` builtin, or related code. It's important to note that even if the corresponding module has already been imported, the module import lock will still be acquired, if only briefly.

So don't write code like this in a hot loop, especially in threaded code:

```
def slow_things_way_down():
    from foo import bar, baz
    ...
```

It may still make sense to defer your imports. Such deferral can decrease the start time of your app. Just keep in mind that thread(s) performing such imports will be forced to run single threaded because of this lock. So it might make sense for your code to perform deferred imports in a background thread:

```
.. literalinclude:: src/chapter19/background_import.py
```

So as you can see, you need to do at least two imports of the a given module; one in the background thread; the other in the actual place(s) where the module's namespace is being used.

Here's why we need the module import lock. Upon the first import, the import procedure runs the (implicit) top-level function of the module. Even though many modules are often declarative in nature, in Python all definitions are done at runtime. Such definitions potentially include further imports (recursive imports). And the top-level function can certainly perform much more complex tasks. The module import lock simplifies this setup so that it's safely published. We will discuss this concept further later in this chapter.

Note that in the current implementation, the module import lock is global for the entire Jython runtime. This may change in the future.

Working with Tasks

It's usually best to avoid managing the lifecycle of threads directly. Instead, the task model often provides a better abstraction.

Tasks describe the asynchronous computation to be performed. Although there are other options, the object you submit to be executed should implement Java's `Callable` interface (a `call` method without arguments), as this best maps into working with a Python method or function. Tasks move through the states of being created, submitted (to an executor), started, and completed. Tasks can also be cancelled or interrupted.

Executors run tasks using a set of threads. This might be one thread, a thread pool, or as many threads as necessary to run all currently submitted tasks concurrently. The specific choice comprises the executor policy. But generally you want to use a thread pool so as to control the degree of concurrency.

Futures allow code to access the result of a computation – or an exception, if thrown – in a task only at the point when it's needed. Up until that point, the using code can run concurrently with that task. If it's not ready, a wait-on dependency is introduced.

We are going to look at how we can use this functionality by using the example of downloading web pages. We will wrap this up so it's easy to work with, tracking the state of the download, as well as any timing information:

```
.. literalinclude:: src/chapter19/downloader.py
```

In Jython any other task could be done in this fashion, whether it is a database query or a computationally intensive task written in Python. It just needs to support the `Callable` interface.

Next, we need to create the futures. Upon completion of a future, either the result is returned, or an exception is thrown into the caller. This exception will be one of:

- `InterruptedException`
- `ExecutionException`. Your code can retrieve the underlying exception with the `cause` attribute.

(This pushing of the exception into the asynchronous caller is thus similar to how a coroutine works when `send` is called on it.)

Now we have what we need to multiplex the downloads of several web pages over a thread pool:

```
.. literalinclude:: src/chapter19/test_futures.py
```

Up until the `get` method on the returned future, the caller run concurrently with this task. The `get` call then introduces a wait-on dependency on the task's completion. (So this is like calling `join` on the supporting thread.)

Shutting down a thread pool should be as simple as calling the `shutdown` method on the pool. However, you may need to take in account this shutdown can happen during extraordinary times in your code. Here's the Jython version of a robust shutdown function, `shutdown_and_await_termination`, as provided in the standard Java docs:

```
.. literalinclude:: src/chapter19/shutdown.py
```

The `CompletionService` interface provides a nice abstraction to working with futures. The scenario is that instead of waiting for all the futures to complete, as our code did with `invokeAll`, or otherwise polling them, the completion service will push futures as they are completed onto a synchronized queue. This queue can then be consumed, by consumers running in one or more threads:

```
.. literalinclude:: src/chapter19/test_completion.py
```

This setup enables a natural flow. Although it may be tempting to then schedule everything through the completion service's queue, there are limits. For example, if you're writing a scalable web spider, you would want to externalize this work queue. But for simple manangement, it would certainly suffice.

Why Use Tasks Instead of Threads

A common practice too often seen in production code is the addition of threading in a haphazard fashion:

- Heterogeneous threads. Perhaps you have one thread that queries the database. And another that rebuilds an associated index. What happens when you need to add another query?
- Dependencies are managed through a variety of channels, instead of being formally structured. This can result in a rats' nest of threads synchronizing on a variety of objects, often with timers and other event sources thrown in the mix.

It's certainly possible to make this sort of setup work. Just debug away. But using tasks, with explicit wait-on dependencies and time scheduling, makes it far simpler to build a simple, scalable system.

Thread Safety

Thread safety addresses such questions as:

- Can the (unintended) interaction of two or more threads corrupt a mutable object? This is especially dangerous for a collection like a list or a dictionary, because such corruption could potentially render the underlying data structure unusable or even produce infinite loops when traversing it.
- Can an update get lost? Perhaps the canonical example is incrementing a counter. In this case, there can be a data race with another thread in the time between retrieving the current value, and then updating with the incremented value.

Jython ensures that its underlying mutable collection types – `dict`, `list`, and `set` – cannot be corrupted. But updates still might get lost in a data race.

However, other Java collection objects that your code might use would typically not have such no-corruption guarantees. If you need to use `LinkedHashMap`, so as to support an ordered dictionary, you will need to consider thread safety if it will be both shared and mutated.

Here's a simple test harness we will use in our examples. `ThreadSafetyTestCase` subclasses `unittest.TestCase`, adding a new method `assertContended`:

```
.. literalinclude:: src/chapter19/threadsafety.py
```

This new method runs a target function and asserts that all threads properly terminate. Then the testing code needs to check for any other invariants.

For example, we use this idea in Jython to test that certain operations on the `list` type are atomic. The idea is to apply a sequence of operations that perform an operation, then reverse it. One step forward, one step back. The net result should be right where you started, an empty list, which is what the test code asserts:

```
.. literalinclude:: src/chapter19/test_list.py
```

Of course these concerns do not apply at all to immutable objects. Commonly used objects like strings, numbers, datetimes, tuples, and frozen sets are immutable. And you can create your own immutable objects too.

There are a number of other strategies in solving thread safety issues. We will look at them as follows:

- Synchronization
- Atomicity
- Thread Confinement
- Safe Publication

Synchronization

We use synchronization to control the entry of threads into code blocks corresponding to synchronized resources. Through this control we can prevent data races, assuming a correct synchronization protocol. (This can be a big assumption!)

A `threading.Lock` ensures entry by only one thread. (In Jython, but unlike CPython, such locks are always reentrant; there's no distinction between `threading.Lock` and `threading.RLock`.) Other threads have to wait until that thread exits the lock. Such explicit locks are the simplest and perhaps most portable synchronization to perform.

You should generally manage the entry and exit of such locks through a with-statement; failing that, you must use a try-finally to ensure that the lock is always released when exiting a block of code.

Here's some example code using the with-statement. The code allocates a lock, then shares it amongst some tasks:

```
.. literalinclude:: src/chapter19/test_lock.py
   :pyobject: LockTestCase.test_with_lock
```

Alternatively, you can do this with try-finally:

```
.. literalinclude:: src/chapter19/test_lock.py
   :pyobject: LockTestCase.test_try_finally_lock
```

But don't do this. It's actually slower than the with-statement. And using the with-statement version also results in more idiomatic Python code.

Another possibility is to use the `synchronize` module, which is specific to Jython. This module provides a `make_synchronized` decorator function, which wraps any callable in Jython in a `synchronized` block:

```
.. literalinclude:: src/chapter19/test_synchronized.py
```

In this case, you don't need to explicitly release anything. Even in the the case of an exception, the synchronization lock is always released upon exit from the function. Again, this version is also slower than the with-statement form, and it doesn't use explicit locks.

Jython's current runtime (as of 2.5.1) can execute the with-statement form more efficiently through both runtime support and how this statement is compiled. The reason is that most JVMs can perform analysis on a chunk of code (the *compilation unit*, including any inlining) to avoid synchronization overhead, so long as two conditions are met. First, the chunk contains both the lock and unlock. And second, the chunk is not too long for the JVM to perform its analysis. The with-statement's semantics make it relatively easy for us to do that when working with built-in types like `threading.Lock`, while avoiding the overhead of Java runtime reflection.

In the future, support of the new `invokedynamic` bytecode should collapse these performance differences.

The `threading` module offers portability, but it's also minimalist. You may want to use the synchronizers in `Java.util.concurrent`, instead of their wrapped versions in `threading`. In particular, this approach is necessary if you want to wait on a lock with a timeout. And you may want to use factories like `Collections.synchronizedMap`, when applicable, to ensure the underlying Java object has the desired synchronization.

Deadlocks

But use synchronizatn carefully. This code will always eventually deadlock:

```
.. literalinclude:: src/chapter19/deadlock.py
```


Deadlock results from a cycle of any length of wait-on dependencies. For example, Alice is waiting on Bob, but Bob is waiting on Alice. Without a timeout or other change in strategy – Alice just gets tired of waiting on Bob! – this deadlock will not be broken.

Avoiding deadlocks can be done by never acquiring locks such that a cycle like that can be created. If we rewrote the example so that locks are acquired in the same order (Bob always allows Alice to go first), there would be no deadlocks. However, this ordering is not always so easy to do. Often, a more robust strategy is to allow for timeouts.

Other Synchronization Objects

The `Queue` module implements a first-in, first-out synchronized queue. (Synchronized queues are also called blocking queues, and that’s how they are described in `java.util.concurrent`.) Such queues represent a thread-safe way to send objects from one or more producing threads to one or more consuming threads.

Often, you will define a poison object to shut down the queue. This will allow any consuming, but waiting threads to immediately shut down. Or just use Java’s support for executors to get an off-the-shelf solution.

If you need to implement another policy, such as last-in, first-out or based on a priority, you can use the comparable synchronized queues in `java.util.concurrent` as appropriate. (Note these have since been implemented in Python 2.6, so they will be made available when Jython 2.6 is eventually released.)

`Condition` objects allow for one thread to notify another thread that’s waiting on a condition to wake up; `notifyAll` is used to wake up all such threads. Along with `Queue`, this is probably the most versatile of the synchronizing objects for real usage.

`Condition` objects are always associated with a `Lock`. Your code needs to bracket waiting and notifying the condition by acquiring the corresponding lock, then finally (as always!) releasing it. As usual, this is easiest done in the context of the `with`-statement.

For example, here’s how we actually implement a `Queue` in the standard library of Jython (just modified here to use the `with`-statement). We can’t use a standard Java blocking queue, because the requirement of being able to join on the queue when there’s no more work to be performed requires a third condition variable:

```
.. literalinclude:: src/chapter19/Queue.py
```

There are other mechanisms to synchronize, including exchangers, barriers, latches, etc. You can use semaphores to describe scenarios where it’s possible for multiple threads to enter. Or use locks that are set up to distinguish reads from writes. There are many possibilities for the Java platform. In our experience, Jython should be able to work with any of them.

Atomic Operations

An atomic operation is inherently thread safe. Data races and object corruption do not occur. And it’s not possible for other threads to see an inconsistent view.

Atomic operations are therefore simpler to use than synchronization. In addition, atomic operations will often use underlying support in the CPU, such as a `compare-and-swap` instruction. Or they may use locking too. The important thing to know is that the lock is not directly visible. Also, if synchronization is used, it’s not possible to expand the scope of the synchronization. In particular, callbacks and iteration are not feasible.

Python guarantees the atomicity of certain operations, although at best it’s only informally documented. Fredrik Lundh’s article on “Thread Synchronization Methods in Python” summarizes the mailing list discussions and the state of the CPython implementation. Quoting his article, the following are atomic operations for Python code:

- Reading or replacing a single instance attribute
- Reading or replacing a single global variable

- Fetching an item from a list
- Modifying a list in place (e.g. adding an item using `append`)
- Fetching an item from a dictionary
- Modifying a dictionary in place (e.g. adding an item, or calling the `clear` method)

Although unstated, this also applies to equivalent ops on the builtin `set` type.

For CPython, this atomicity emerges from combining its Global Interpreter Lock (GIL), the Python bytecode virtual machine execution loop, and the fact that types like `dict` and `list` are implemented natively in C and do not release the GIL.

Despite the fact that this is in some sense accidentally emergent, it is a useful simplification for the developer. And it's what existing Python code expects. So this is what we have implemented in Jython.

In particular, because `dict` is a `ConcurrentHashMap`, we also expose the following methods to atomically update dictionaries:

```
* ``setifabsent``
* ``update``
```

It's important to note that iterations are not atomic, even on a `ConcurrentHashMap`.

Atomic operations are useful, but they are pretty limited too. Often, you still need to use synchronization to prevent data races. And this has to be done with care to avoid deadlocks and starvation.

Thread Confinement

Thread confinement is often the best solution to resolve most of the problems seen in working with mutable objects. In practice, you probably don't need to share a large percentage of the mutable objects used in your code. Very simply put, if you don't share, then thread safety issues go away.

Not all problems can be reduced to using thread confinement. There are likely some shared objects in your system, but in practice most can be eliminated. And often the shared state is someone else's problem:

- Intermediate objects don't require sharing. For example, if you are building up a buffer that is only pointed to by a local variable, you don't need to synchronize. It's an easy prescription to follow, so long as you are not trying to keep around these intermediate objects to avoid allocation overhead. Don't do that.
- Producer-consumer. Construct an object in one thread, then hand it off to another thread. You just need to use an appropriate synchronizer object, such as a `Queue`.
- Application containers. The typical database-driven web applications makes for the classic case. For example, if you are using `modjy`, then the database connection pools and thread pools are the responsibility of the servlet container. And they are not directly observable. (But don't do things like share database connections across threads.) Caches and databases then are where you will see shared state.
- Actors. The actor model is another good example. Send and receive messages to an actor (effectively an independent thread) and let it manipulate any objects it owns on your behalf. Effectively this reduces the problem to sharing one mutable object, the message queue. The message queue can then ensure any accesses are appropriately serialized, so there are no thread safety issues.

Unfortunately thread confinement is not without issues in Jython. For example, if you use `StringIO`, you have to pay the cost that this class uses `list`, which is synchronized. Although it's possible to further optimize the Jython implementation of the Python standard library, if a section of code is hot enough, you may want to consider rewriting that in Java to ensure no additional synchronization overhead.

Lastly, thread confinement is not perfect in Python, because of the possibility of introspecting on frame objects. This means your code can see local variables in other threads, and the objects they point to. But this is really more of an issue for how optimizable Jython is when run on the JVM. It won't cause thread safety issues if you don't exploit this loophole. We will discuss this more in the section on the Python Memory Model.

Python Memory Model

Reasoning about concurrency in Python is easier than in Java. This is because the memory model is not as surprising to our conventional reasoning about how programs operate. However, this also means that Python code sacrifices significant performance to keep it simpler.

Here's why. In order to maximize Java performance, it's allowed for a CPU to arbitrarily re-order the operations performed by Java code, subject to the constraints imposed by *happens-before* and *synchronizes-with* relationships. (The published [Java memory model](#) goes into more details on these constraints.)

Although such reordering is not visible within a given thread, the problem is that it's visible to other threads. Of course, this visibility only applies to changes made to non-local objects; thread confinement still applies.

In particular, this means you cannot rely on the apparent sequential ordering of Java code when looking at two or more threads.

Python is different. The fundamental thing to know about Python, and what we have implemented in Jython, is that setting any attribute in Python is a volatile write; and getting any attribute is a volatile read. This is because Python attributes are stored in dictionaries, and in Jython, this follows the semantics of the backing `ConcurrentHashMap`. So `get` and `set` are volatile.

So this means that Python code has sequential consistency. Execution follows the ordering of statements in the code. There are no surprises here.

And this means that *safe publication* is pretty much trivial in Python, when compared to Java. Safe publication means the thread safe association of an object with a name. Because this is always a memory-fenced operation in Python, your code simply needs to ensure that the object itself is built in a thread-safe fashion; then publish it all at once by setting the appropriate variable to this object.

If you need to create module-level objects – singletons – then you should do this in the top-level script of the module so that the module import lock is in effect.

Interruption

Long-threading threads should provide some opportunity for cancellation. The typical pattern is something like this:

```
class DoSomething(Runnable):
    def __init__(self):
        cancelled = False

    def run(self):
        while not self.cancelled:
            do_stuff()
```

Remember, Python variables are always volatile, unlike Java. There are no problems with using a `cancelled` flag like this.

Thread interruption allows for even more responsive cancellation. In particular, if a thread is waiting on most any synchronizers, such as a condition variable or on file I/O, this action will cause the waited-on method to exit with an `InterruptedException`. (Unfortunately lock acquisition, except under certain cases such as using `lockInterruptibly` on the underlying Java lock, is not interruptible.)

Although Python's `threading` module does not itself support interruption, it is available through the standard Java thread API. First, let's import this class. We will rename it to `JThread` so it doesn't conflict with Python's version:

```
from java.lang import Thread as JThread
```

As we have seen, you can use Java threads as if they are Python threads. So logically you should be able to do the converse: use Python threads as if they are Java threads. Therefore it would be nice to make calls like `JThread.interrupt(obj)`.

Incidentally, this formulation, instead of `obj.interrupt()`, looks like a static method on a class, as long as we pass in the object as the first argument. This adaptation is a good use of Python's explicit self.

But there's a problem here. As of the latest released version (Jython 2.5.1), we forgot to include an appropriate `__tojava__` method on the `Thread` class! So this looks like you can't do this trick after all.

Or can you? What if you didn't have to wait until we fix this bug? You could explore the source code – or look at the class with `dir`. One possibility would be to use the nominally private `_thread` attribute on the `Thread` object. After all `_thread` is the attribute for the underlying Java thread. Yes, this is an implementation detail, but it's probably fine to use. It's not so likely to change.

But we can do even better. We can *monkey patch* the `Thread` class such that it has an appropriate `__tojava__` method, but only if it doesn't exist. So this patching is likely to work with a future version of Jython because we are going to fix this missing method before we even consider changing its implementation and removing `_thread`.

So here's how we can monkey patch, following a recipe of Guido van Rossum:

```
.. literalinclude:: src/chapter19/monkeypatch.py
```

This `monkeypatch_method` decorator allows us to add a method to a class after the fact. (This is what Ruby developers call *opening* a class.) Use this power with care. But again, you shouldn't worry too much when you keep such fixes to a minimum, especially when it's essentially a bug fix like this one. In our case, we will use a variant, the `monkeypatch_method_if_not_set` decorator, to ensure we only patch if it has not been fixed by a later version.

Putting it all together, we have this code:

```
.. literalinclude:: src/chapter19/interrupt.py
```

(It does rely on the use of `threading.Condition` to have something to wait on. We will talk about condition variables later.)

Lastly, you could simply access interruption through the `cancel` method provided by a `Future`. No need to monkey patch!

Conclusion

Jython can fully take advantage of the underlying Java platform's support for concurrency. You can also use the standard Python threading constructs, which in most cases just wrap the corresponding Java functionality. The standard mutable Python collection types have been implemented in Jython with concurrency in mind. And Python's sequential consistency removes some potential bugs.

But concurrent programming is still not easy to get right, either in Python or in Java. You should consider higher-level concurrency primitives, such as tasks. And you should be disciplined in how your code shares mutable state.

Appendix A: Using Other Tools with Jython

The primary focus of this appendix is to provide information on using some external python packages with Jython. In some circumstances, the tools must be used or installed a bit differently on Jython than on CPython, and those differences will be noted. Since there is a good deal of documentation on the usage of these tools available on the web, this appendix will focus on using the tool specifically with Jython. However, relevant URLs will be cited for finding more documentation on each of the topics.

setuptools

Setuptools is a library that builds upon distutils, the standard Python distribution facility. It offers some advanced tools like `easy_install`, a command to automatically download and install a given Python package and its dependencies.

To get setuptools, download `ez_setup.py` from http://peak.telecommunity.com/dist/ez_setup.py. Then, go to the directory where you left the downloaded file and execute:

```
$ jython ez_setup.py
```

The output will be similar to the following:

```
Downloading http://pypi.python.org/packages/2.5/s/setuptools/setuptools-0.6c9-py2.5.
→egg
Processing setuptools-0.6c9-py2.5.egg
Copying setuptools-0.6c9-py2.5.egg to /home/lsoto/jython2.5.0/Lib/site-packages
Adding setuptools 0.6c9 to easy-install.pth file
Installing easy_install script to /home/lsoto/jython2.5.0/bin
Installing easy_install-2.5 script to /home/lsoto/jython2.5.0/bin

Installed /home/lsoto/jython2.5.0/Lib/site-packages/setuptools-0.6c9-py2.5.egg
Processing dependencies for setuptools==0.6c9
Finished processing dependencies for setuptools==0.6c9
```

As you can read on the output, the `easy_install` script has been installed to the `bin` directory of the Jython installation (`/home/lisoto/jython2.5.0/bin` in the example above). If you work frequently with Jython it's a good idea to prepend this directory to the `PATH` environment variable, so you don't have to type the whole path each time you want to use `easy_install` or other scripts installed to this directory. From now on I'll assume that this is the case. If you don't want to prepend Jython's `bin` directory to your `PATH` for any reason, remember to type the complete path on each example (i.e., type `/path/to/jython/bin/easy_install` when I say `easy_install`).

OK, so now you have `easy_install`. What's next? Let's grab a Python library with it! For example, let's say that we need to access twitter from a program written in Jython and we want to use python-twitter project, located at <http://code.google.com/p/python-twitter/>.

Without `easy_install` you would go to that URL, read the building instructions and after downloading the latest version and executing a few commands you should be ready to go. Except that libraries often depend on other libraries (as the case with python-twitter which depends on simplejson) so you would have to repeat this boring process a few times.

With `easy_install` you simply run:

```
$ easy_install python-twitter
```

And you get the following output:

```
Searching for python-twitter
Reading http://pypi.python.org/simple/python-twitter/
Reading http://code.google.com/p/python-twitter/
Best match: python-twitter 0.6
Downloading http://python-twitter.googlecode.com/files/python-twitter-0.6.tar.gz
Processing python-twitter-0.6.tar.gz
Running python-twitter-0.6/setup.py -q bdist_egg --dist-dir /var/folders/mQ/
↳mQkMnKiaE583pWpee85FFk+++TI/-Tmp-/easy_install-FU5COZ/python-twitter-0.6/egg-dist-
↳tmp-EeR4RD
zip_safe flag not set; analyzing archive contents...
Unable to analyze compiled code on this platform.
Please ask the author to include a 'zip_safe' setting (either True or False) in the_
↳package's setup.py
Adding python-twitter 0.6 to easy-install.pth file

Installed /home/lisoto/jython2.5.0/Lib/site-packages/python_twitter-0.6-py2.5.egg
Processing dependencies for python-twitter
Searching for simplejson
Reading http://pypi.python.org/simple/simplejson/
Reading http://undefined.org/python/#simplejson
Best match: simplejson 2.0.9
Downloading http://pypi.python.org/packages/source/s/simplejson/simplejson-2.0.9.tar.
↳gz#md5=af5e67a39ca3408563411d357e6d5e47
Processing simplejson-2.0.9.tar.gz
Running simplejson-2.0.9/setup.py -q bdist_egg --dist-dir /var/folders/mQ/
↳mQkMnKiaE583pWpee85FFk+++TI/-Tmp-/easy_install-VgAKxa/simplejson-2.0.9/egg-dist-tmp-
↳jcntqu
*****
WARNING: The C extension could not be compiled, speedups are not enabled.
Failure information, if any, is above.
I'm retrying the build without the C extension now.
*****
*****
WARNING: The C extension could not be compiled, speedups are not enabled.
Plain-Python installation succeeded.
*****
```

```
Adding simplejson 2.0.9 to easy-install.pth file

Installed /home/lisoto/jython2.5.0/Lib/site-packages/simplejson-2.0.9-py2.5.egg
Finished processing dependencies for python-twitter
```

The output is a bit verbose, but it gives you a detailed idea of the steps automated by `easy_install`. Let's review it piece by piece:

```
Searching for python-twitter
Reading http://pypi.python.org/simple/python-twitter/
Reading http://code.google.com/p/python-twitter/
Best match: python-twitter 0.6
Downloading http://python-twitter.googlecode.com/files/python-twitter-0.6.tar.gz
```

We asked for “python-twitter”, which is looked up on PyPI, the Python Package Index which lists all the Python packages produced by the community. The version 0.6 was selected since it was the most recent version at the time I ran the command.

Let's see what's next on the `easy_install` output:

```
Running python-twitter-0.6/setup.py -q bdist_egg --dist-dir /var/folders/mQ/
↳mQkMnKiaE583pWpee85FFk+++TI/-Tmp-/easy_install-FU5COZ/python-twitter-0.6/egg-dist-
↳tmp-EeR4RD
zip_safe flag not set; analyzing archive contents...
Unable to analyze compiled code on this platform.
Please ask the author to include a 'zip_safe' setting (either True or False) in the_
↳package's setup.py
Adding python-twitter 0.6 to easy-install.pth file

Installed /home/lisoto/jython2.5.0/Lib/site-packages/python_twitter-0.6-py2.5.egg
```

Nothing special here: it ran the needed commands to install the library. The next bits are more interesting:

```
Processing dependencies for python-twitter
Searching for simplejson
Reading http://pypi.python.org/simple/simplejson/
Reading http://undefined.org/python/#simplejson
Best match: simplejson 2.0.9
Downloading http://pypi.python.org/packages/source/s/simplejson/simplejson-2.0.9.tar.
↳gz#md5=af5e67a39ca3408563411d357e6d5e47
```

As you can see, the dependency on `simplejson` was discovered and, since it is not already installed it is being downloaded. Next we see:

```
Processing simplejson-2.0.9.tar.gz
Running simplejson-2.0.9/setup.py -q bdist_egg --dist-dir /var/folders/mQ/
↳mQkMnKiaE583pWpee85FFk+++TI/-Tmp-/easy_install-VgAKxa/simplejson-2.0.9/egg-dist-tmp-
↳jcntqu
*****
WARNING: The C extension could not be compiled, speedups are not enabled.
Failure information, if any, is above.
I'm retrying the build without the C extension now.
*****
*****
WARNING: The C extension could not be compiled, speedups are not enabled.
Plain-Python installation succeeded.
*****
Adding simplejson 2.0.9 to easy-install.pth file
```

```
Installed /home/lisoto/jython2.5.0/Lib/site-packages/simplejson-2.0.9-py2.5.egg
```

The warnings are produced because the `simplejson` installation tries to compile a C extension which for obvious reasons only works with CPython and not with Jython.

Finally, we see:

```
Finished processing dependencies for python-twitter
```

Which signals the end of the automated installation process for `python-twitter`. You can test that it was successfully installed by running Jython and doing an `import twitter` on the interactive interpreter.

As noted above `easy_install` will try to get the latest version for the library you specify. If you want a particular version, for example the 0.5 release of `python-twitter` then you can specify it in this way:

```
$ easy_install python-twitter==0.5
```

Warning: Note that it's not a good idea to have two version of the same library installed at the same time. Take a look at the [virtualenv](#) section below for a solution to the problem of running different programs requiring different versions of the same library.

For debugging purposes is always useful to know where the bits installed using `easy_install` go. As you can stop of the install output, they are installed into `<path-to-jython>/Lib/site-packages/<name_of_library>-<version>.egg` which may be a directory or a compressed zip file. Also, `easy_install` adds an entry to the file `<path-to-jython>/Lib/site-packages/easy-install.pth`, which ends up adding the directory or zip file to `sys.path` by default.

Unfortunately `setuptools` don't provide any automated way to uninstall packages. You will have to manually delete the package egg directory or zip file and remove the associated line on `easy-install.pth`.

virtualenv

Oftentimes it is nice to have separate versions of tools running on the same machine. The `virtualenv` tool provides a way to create a virtual Python environment that can be used for various purposes including installation of different package versions. Virtual environments can also be nice for those who do not have administrative access for a particular Python installation but still need to have the ability to install packages to it, such is often the case when working with domain hosts. Whatever the case may be, the `virtualenv` tool provides a means for creating one or more virtual environments for a particular Python installation so that the libraries can be installed into controlled environments other than the global site-packages area for your Python or Jython installation. The release of Jython 2.5.0 opened new doors for the possibility of using such tools as `virtualenv`.

To use `virtualenv` with Jython, we first need to obtain it. The easiest way to do so is via the Python Package Index. As you had learned in the previous section, `easy_install` is the way to install packages from the PyPI. The following example shows how to install `virtualenv` using `easy_install` with Jython.

```
:: jython easy_install.py virtualenv
```

Once installed, it is quite easy to use the tool for creation of a virtual environment. The virtual environment will include a Jython executable along with an installation of `setuptools` and it's own site-packages directory. This was done so that you have the ability to install different packages from the PyPI to your virtual environment. Let's create an environment named `JY2.5.1Env` using the `virtualenv.py` module that exists within our Jython environment.


```
jython <<path to Jython>>/jython2.5.1/Lib/site-packages/virtualenv-1.3.3-py2.5.egg/
↳virtualenv.py JY2.5.1Env
New jython executable in JY2.5.1Env/bin/jython
Installing setuptools.....done.
```

Now a new directory named JY2.5.1Env should have been created within your current working directory. You can run Jython from this virtual environment by simply invoking the executable that was created. The virtualenv tool allows us the ability to open a terminal and designate it to be used for our virtual Jython environment exclusively via the use of the *activate* command. To do so, open up a terminal and type the following:

```
source <path-to-virtual-environment/JY2.5.1Env/bin/activate
```

Once this is done, you should notice that the command line is preceeded with the name of the virtual environment that you have activated. Any Jython shell or tool used in this terminal will now be using the virtual environment. This is an excellent way to run a tool using two different versions of a particular library or for running a production and development environment side-by-side. If you run the *easy_install.py* tool within the activated virtual environment terminal then the tool(s) will be installed into the virtual environment. There can be an unlimited number of virtual environments installed on a particular machine. To stop using the virtual environment within the terminal, simply type:

```
deactivate
```

Now your terminal should go back to normal use and default to the global Jython installation. Once deactivated any of the Jython references made will call the global installation or libraries within the global site-packages area. It should be noted that when you create a virtual environment, it automatically inherits all of the packages used by the global installation. Therefore if you have a library installed in your global site-packages area then it can be used from the virtual environment right away. A good practice is to install only essential libraries into your global Jython environment and then install one-offs or test libraries into virtual environments.

It is useful to have the ability to list installations that are in use within a particular environment. One way to do this is to install the *yolk* utility and make use of its *-l* command. In order to install *yolk*, you must grab a copy of the latest version of Jython beyond 2.5.1 as there has been a patch submitted that corrects some functionality which is used by *yolk*. You must also be running with JDK 1.6 or above as the patched version of Jython makes use of the *webbrowser* module. The *webbrowser* module makes use of some *java.awt.Desktop* features that are only available in JDK 1.6 and beyond. To install *yolk*, use the *ez_install.py* script as we've shown previously.

```
jython ez_install.py yolk
```

Once installed you can list the package installations for your Jython installations by issuing the *-l* command as follows:

```
yolk -l
Django          - 1.0.2-final - non-active development (/jython2.5.1/Lib/site-
↳packages)
Django          - 1.0.3      - active development (/jython2.5.1/Lib/site-packages/
↳Django-1.0.3-py2.5.egg)
Django          - 1.1       - non-active development (/jython2.5.1/Lib/site-
↳packages)
SQLAlchemy      - 0.5.4p2 - active development (/jython2.5.1/Lib/site-packages)
SQLAlchemy      - 0.6beta1 - non-active development (/jython2.5.1/Lib/site-
↳packages)
django-jython   - 0.9       - active development (/jython2.5.1/Lib/site-packages/
↳django_jython-0.9-py2.5.egg)
django-jython   - 1.0b1    - non-active development (/jython2.5.1/Lib/site-
↳packages)
nose            - 0.11.1   - active development (/jython2.5.1/Lib/site-packages/
↳nose-0.11.1-py2.5.egg)
setuptools      - 0.6c9    - active
setuptools      - 0.6c9    - active
```

```
snakefight      - 0.4          - active development (/jython2.5.1/Lib/site-packages/
↳snakefight-0.4-py2.5.egg)
virtualenv      - 1.3.3       - active development (/jython2.5.1/Lib/site-packages/
↳virtualenv-1.3.3-py2.5.egg)
wsgiref         - 0.1.2       - active development (/jython2.5.1/Lib)
yolk            - 0.4.1        - active
```

As you can see, all installed packages will be listed. If you are using yolk from within a virtual environment then you will see all packages installed in that virtual environment as well as those installed into the global environment.

Similarly to `setuptools`, there is no way to automatically uninstall `virtualenv`. You must also manually delete the package egg directory or zip file as well as remove references within `easy-install.pth`.

Appendix B: Jython Cookbook - A compilation of community submitted code examples

There are a plethora of examples for using Jython that can be found on the web. This appendix is a compilation of some of the most useful examples that we have found. There are hundreds of examples available on the web. These that were chosen are focused on topics that are not widely covered elsewhere on the web.

Unless otherwise noted, each of these examples have been originally authored for working on versions of Jython prior to 2.5.x but we have tested each of them using Jython 2.5.1 and function as advertised.

Logging

Using log4j With Jython - Josh Juneau

Are you still using the Jython `print` command to show your errors? How about in a production environment, are you using any formal logging? If not, you should be doing so...and the Apache `log4j` API makes it easy to do so. Many Java developers have grown to love the `log4j` API and it is utilized throughout much of the community. That is great news for Jython developers since we've got direct access to Java libraries!

Setting Up Your Environment

The most difficult part about using `log4j` with Jython is the setup. You must ensure that the `log4j.jar` archive resides somewhere within your Jython `PATH` (usually this entails setting the `CLASSPATH` to include necessary files). You then set up a properties file for use with `log4j`. Within the properties file, you can include appender information, where logs should reside, and much more.

Example properties file:

```
log4j.rootLogger=debug, stdout, R

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=C:\\Jython\\testlog4j.log

log4j.appender.R.MaxFileSize=100KB
# Keep one backup file
```

```
log4j.appender.R.MaxBackupIndex=1

log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n
```

You are now ready to use log4j in your Jython application. As you can see, if you've ever used log4j with Java, it is pretty much the same.

Using log4j in a Jython Application

Once again, using log4j within a Jython application is very similar to its usage in the Java world.

First, you must import the log4j packages:

```
from org.apache.log4j import *
```

Second, you obtain a new logger for your class or module and set up a PropertyConfigurator:

```
logger = Logger.getLogger("myClass")
# Assume that the log4j properties resides within a folder named "utilities"
PropertyConfigurator.configure(sys.path[0] + "/utilities/log4j.properties")
```

Lastly, use log4j:

```
# Example module within the class:
def submitDocument(self, event):
    try:
        # Assume we perform some SQL here
    except SQLException, ex:
        self.logger.error("docPanel#submitDocument ERROR: %s" % (ex))
```

Your logging will now take place within the file you specified in the properties file for log4j.appender.R.File.

Using log4j in Jython Scripts

Many may ask, why in the world would you be interested in logging information about your scripts? Most of the time a script is executed interactively via the command line. However, there are plenty of instances where it makes sense to have the system invoke a script for you. As you probably know, this technique is used quite often within an environment to run nightly tasks, or even daily tasks which are automatically invoked on a scheduled basis. For these cases, it can be extremely useful to log errors or information using log4j. Some may even wish to create a separate automated task to email these log files after the tasks complete.

The overall implementation is the same as above, the most important thing to remember is that you must have the log4j.jar archive and properties file within your Jython path. Once this is ready to go you can use log4j in your script.

```
from org.apache.log4j import *
logger = Logger.getLogger("scriptname")
PropertyConfigurator.configure("C:\path_to_properties\log4j.properties")
logger.info("Test the logging")
```

Author: Josh Juneau URL: <http://wiki.python.org/jython/JythonMonthly/Articles/August2006/1>

Another log4j Example - Greg Moore

This example require several things.

- log4j on the classpath
- log4j.properties (below) in the same directory as the example

- example.xml (below) in the same directory as the example below
- And of course Jython

log4j Example

This is a simple example to show how easy it is to use log4j in your own scripts. The source is well documented but if you have any questions or want to more info use your favorite search engine and type in log4j.

```
from org.apache.log4j import *

class logtest:
    def __init__(self):
        log.info("start of Logtest")
        log.debug('just before file read')
        try:
            log.warn('file read proceeding to processing')
            xmlStringData = open('example.xml').read()
        except:
            #yes, more could have been done here but this is just an example
            log.error('file read FAILURE')
            log.info('file read proceeding to processing')
            # since this is just an example processing would go here.
            log.warn('its just an example, OK?')
            pi = 3.141592681
            msg = 'do you like?' + str(pi)
            log.info(msg)
            log.debug('lets try to parse the string')
            if '[CDATA' in xmlStringData:
                log.warn('No CDATA section.')
                #say good bye and close the log file.
                log.info('That all. The End. Good Bye')
                log.shutdown()

if __name__ == '__main__':
    # loggingTest is just a string that identifies this log.
    log = Logger.getLogger("loggingTest")
    #use the config data in the properties file
    PropertyConfigurator.configure('log4j.properties')
    log.info('This is the start of the log file')
    logit = logtest()
    print '\n\nif you change the log level in the properties'
    print "file you'll get varing amouts of log data."
```

log4j.properties

This file is required by the code above. it need to be in the same directory as the example however It can be anywhere as long as you provide a full path to the file. It configures how log4j operates. If it is not found it defaults to a default logging level. Since this is for example purposes the file below is larger then really needed.

```
#define logging level and output
log4j.rootLogger=debug, stdout, LOGFILE
#log4j.rootLogger=info, LOGFILE
# this 2 lines tie the apache logging into log4j
log4j.logger.org.apache.axis.SOAPPart=DEBUG
log4j.logger.httpclient.wire.header=info
log4j.logger.org.apache.commons.httpclient=DEBUG

# where is the logging going.
```

```
# This is for std out and defines the log output format
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{HH:mm:ss,SSS} | %p | [%c] %m%n %t

#log it to a file as well. and define a filename, max file size and number of backups
log4j.appender.LOGFILE=org.apache.log4j.RollingFileAppender
log4j.appender.LOGFILE.File=jythonTest.log
log4j.appender.LOGFILE.MaxFileSize=100KB
# Keep one backup file
log4j.appender.LOGFILE.MaxBackupIndex=1

log4j.appender.LOGFILE.layout=org.apache.log4j.PatternLayout
# Pattern for logfile - only diff is that date is added
log4j.appender.LOGFILE.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} | %p | [%c] %m
↪ %n
# Other Examples: only time, loglog level, loggerName
#log4j.appender.LOGFILE.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss},%p,%c %m%n
#above plus filename, lineNumber, Class Name, method name
#log4j.appender.LOGFILE.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss},%p,%c,%F,%L,
↪ %C{l},%M %m%n
```

Example xml file

This is here for completeness. Any text file could be use with the example above by changing the ‘open’ line in the above line.

```
<?xml version="1.0" encoding="utf-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <GetXmlReport xmlns="http://localhost/Services/GetXmlReport">
      <xmlrequest>
        <Inquiry>
          <Client>
            <Type>W</Type>
          </Client>
          <Report>I</Report>
          <Provider>
            <ProviderID>TU</ProviderID>
          </Provider>
          <ClientInfo>
            <Name>
              <First>Cathrine</First>
              <Middle />
              <Surname>Knight</Surname>
            </Name>
            <Account>34-5424-77</Account>
            <DateOfBirth>10/12/1938</DateOfBirth>
            <Address>
              <Line1>4780 Centerville</Line1>
              <CityStPostal>Saint Paul, MN 55127</CityStPostal>
            </Address>
          </ClientInfo>
        </Inquiry>
      </xmlrequest>
```

```
</GetXmlReport>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Author: Greg Moore URL: <http://wiki.python.org/jython/Log4jExample>

Working with Spreadsheets

Below are a few Apache Poi examples. These examples requires Apache Poi installed and on the classpath.

Create Spreadsheet

This is from the Jython mailing list and was posted September 2007

This is based on Java code at <http://officewriter.softartisans.com/OfficeWriter-306.aspx> and converted to Jython by Alfonso Reyes

```
#jython poi example. from Jython mailing list

from java.io import FileOutputStream
from java.util import Date
from java.lang import System, Math
from org.apache.poi.hssf.usermodel import *
from org.apache.poi.hssf.util import HSSFColor

startTime = System.currentTimeMillis()

wb = HSSFWorkbook()
fileOut = FileOutputStream("POIOut2.xls")

# Create 3 sheets
sheet1 = wb.createSheet("Sheet1")
sheet2 = wb.createSheet("Sheet2")
sheet3 = wb.createSheet("Sheet3")
sheet3 = wb.createSheet("Sheet4")

# Create a header style
styleHeader = wb.createCellStyle()
fontHeader = wb.createFont()
fontHeader.setBoldweight(2)
fontHeader.setFontHeightInPoints(14)
fontHeader.setFontName("Arial")
styleHeader.setFont(fontHeader)

# Create a style used for the first column
style0 = wb.createCellStyle()
font0 = wb.createFont()
font0.setColor(HSSFColor.RED.index)
style0.setFont(font0)

# Create the style used for dates.
styleDates = wb.createCellStyle()
styleDates.setDataFormat(HSSFDataFormat.getBuiltinFormat("m/d/yy h:mm"))

# create the headers
```

```

rowHeader = sheet1.createRow(1)
# String value
cell10 = rowHeader.createCell(0)
cell10.setCellStyle(styleHeader)
cell10.setCellValue("Name")

# numbers
for i in range(0, 8, 1):
    cell = rowHeader.createCell((i + 1))
    cell.setCellStyle(styleHeader)
    cell.setCellValue("Data " + str( (i + 1)) )

# Date
cell110 = rowHeader.createCell(9)
cell110.setCellValue("Date")
cell110.setCellStyle(styleHeader)

for i in range(0, 100, 1):
    # create a new row
    row = sheet1.createRow(i + 2)
    for j in range(0, 10, 1):
        # create each cell
        cell = row.createCell(j)
        # Fill the first column with strings
        if j == 0:
            cell.setCellValue("Product " + str(i))
            cell.setCellStyle(style0)
        # Fill the next 8 columns with numbers.
        elif j < 9:
            cell.setCellValue( (Math.random() * 100))
        # Fill the last column with dates.
        else:
            cell.setCellValue(Date())
            cell.setCellStyle(styleDates)

# Summary row
rowSummary = sheet1.createRow(102)
sumStyle = wb.createCellStyle()
sumFont = wb.createFont()
sumFont.setBoldweight( 5)
sumFont.setFontHeightInPoints(12)
sumStyle.setFont(sumFont)
sumStyle.setFillPattern(HSSFCellStyle.FINE_DOTS)
sumStyle.setFillForegroundColor(HSSFColor.GREEN.index)
cellSum0 = rowSummary.createCell( 0)
cellSum0.setCellValue("TOTALS:")
cellSum0.setCellStyle(sumStyle)

# numbers
# B
cellB = rowSummary.createCell( 1)
cellB.setCellStyle(sumStyle)
cellB.setCellFormula("SUM(B3:B102)")

```

Read an Excel file

Posted to the Jython-users mailing list by Alfonso Reyes on October 14, 2007 This Jython code will open and read an existant Excel file you can download the file at <http://www.nabble.com/file/p13199712/Book1.xls>

```
"""    read.py
Read an existant Excel file (Book1.xls) and show it on the screen
"""

from org.apache.poi.hssf.usermodel import *
from java.io import FileInputStream

file = "H:Book1.xls"
print file
fis = FileInputStream(file)
wb = HSSFWorkbook(fis)
sheet = wb.getSheetAt(0)

# get No. of rows
rows = sheet.getPhysicalNumberOfRows()
print wb, sheet, rows
cols = 0 # No. of columns
tmp = 0
# This trick ensures that we get the data properly even if it
# does not start from first few rows
for i in range(0, 10,1):
    row = sheet.getRow(i)
    if(row != None):
        tmp = sheet.getRow(i).getPhysicalNumberOfCells()
        if tmp > cols:
            cols = tmp
print cols

for r in range(0, rows, 1):
    row = sheet.getRow(r)
    print r
    if(row != None):
        for c in range(0, cols, 1):
            cell = row.getCell(c)
            if cell != None:
                print cell

#wb.close()
fis.close()
```

URL: <http://wiki.python.org/jython/PoiExample>

Jython and XML - Greg Moore

Element tree

Here is a simple example of using element tree with Jython. Element tree is useful for storing hierarchical data structures, such as simplified XML infosets, into memory and then save them to disk. More information on element tree is at <http://effbot.org/zone/element-index.htm>.

Download element tree from <http://effbot.org/downloads/>

```
from elementtree import ElementTree as ET

root = ET.Element("html")
head = ET.SubElement(root, "head")
title = ET.SubElement(head, "title")
title.text = "Page Title"
body = ET.SubElement(root, "body")
```



```
body.set("bgcolor", "#ffffff")
body.text = "Hello, World!"
tree = ET.ElementTree(root)
tree.write("page.xhtml")

import sys
tree.write(sys.stdout)
```

which produces:

```
<html><head><title>Page Title</title></head><body bgcolor="#ffffff">Hello, World!</
↪body></html>
```

Author: Greg Moore URL: <http://wiki.python.org/jython/XmlRelatedExamples>

Writing and Parsing RSS with ROME - Josh Juneau

Introduction

RSS is an old technology now...it has been around for years. However, it is a technology which remains very useful for disseminating news and other information. The ROME project on java.net is helping to make parsing, generating, and publishing RSS and Atom feeds a breeze for any Java developer.

Since I am particularly fond of translating Java to Jython code, I've taken simple examples from the Project ROME wiki and translated Java RSS reader and writer code into Jython. It is quite easy to do, and it only takes a few lines of code.

Keep in mind that you would still need to build a front-end viewer for such an RSS reader, but I think you will get the idea of how easy it can be just to parse a feed with Project ROME and Jython.

Setting Up The CLASSPATH

In order to use this example, you must obtain the ROME and JDOM jar files and place them into your CLASSPATH.

Windows:

```
set CLASSPATH=C:\Jython\Jython2.2\rome-0.9.jar;%CLASSPATH%
set CLASSPATH=C:\Jython\Jython2.2\jdom.jar;%CLASSPATH%
```

OSX:

```
export CLASSPATH=/path/to/rome-0.9.jar:/path/to/jdom.jar
```

Parsing Feeds

Parsing feeds is easy with ROME. Using ROME with Jython makes it even easier with the elegant Jython syntax. I am not a professional Python or Jython programmer, I am a Java programmer by profession, so my Jython interpretation may be even wordier than it should be.

I took the FeedReader example from the ROME site and translated it into Jython below. You can copy and paste the following code into your own FeedReader.py module and run it to parse feeds. However, the output is unformatted and ugly...creating a good looking front end is up to you.

FeedReader.py

```
#####
# File: FeedReader.py
#
# This module can be used to parse an RSS feed
```

```
#####
from java.net import URL
from java.io import InputStreamReader
from java.lang import Exception
from java.lang import Object
from com.sun.syndication.feed.synd import SyndFeed
from com.sun.syndication.io import SyndFeedInput
from com.sun.syndication.io import XmlReader

class FeedReader(Object):
    def __init__(self, url):
        self.inUrl = url

    def readFeed(self):
        ok = False
        #####
        # If url passed in is blank, then use a default
        #####
        if self.inUrl != '':
            rssUrl = self.inUrl
        else:
            rssUrl = "http://www.dzone.com/feed/frontpage/java/rss.xml"
        #####
        # Parse feed located at given URL
        #####
        try:
            feedUrl = URL(rssUrl)
            input = SyndFeedInput()
            feed = input.build(XmlReader(feedUrl))
            #####
            # Do something here with feed data
            #####
            print(feed)
            ok = True
        except Exception, e:
            print 'An exception has occurred', e
        if ok != True:
            print 'An error has occurred in this reader'

if __name__ == "__main__":
    reader = FeedReader('')
    reader.readFeed()
    print '*****Command Complete...RSS has been parsed*****'
```

Creating Feeds

Similar to parsing a feed, writing a feed is also quite easy. When one creates a feed, it appears to be a bit more complex than parsing, but if you are familiar with XML and it's general structure then it should be relatively easy.

Creating a feed is a three step process. You must first create the feed element itself, then you must add individual feed entries, and lastly you must publish the XML.

FeedWriter.py

```
#####
# File: FeedReader.py
#
# This module can be used to create an RSS feed
#####
```

```

from com.sun.syndication.feed.synd import *
from com.sun.syndication.io import SyndFeedOutput
from java.io import FileWriter
from java.io import Writer
from java.text import DateFormat
from java.text import SimpleDateFormat
from java.util import ArrayList
from java.util import List
from java.lang import Object

class FeedWriter(Object):
    #####
    # Set up the date format
    #####
    def __init__(self, type, name):
        self.DATE_PARSER = SimpleDateFormat('yyyy-MM-dd')
        self.feedType = type
        self.fileName = name

    def writeFeed(self):
        ok = False
        try:
            #####
            # Create the feed itself
            #####
            feed = SyndFeedImpl()
            feed.feedType = self.feedType
            feed.title = 'Sample Feed (created with ROME)'
            feed.link = 'http://rome.dev.java.net'
            feed.description = 'This feed has been created using ROME and Jython'

            #####
            # Add entries to the feed
            #####
            entries = ArrayList()
            entry = SyndEntryImpl()
            entry.title = 'ROME v1.0'
            entry.link = 'http://wiki.java.net/bin/view/Javawsxml/Rome01'
            entry.publishedDate = self.DATE_PARSER.parse("2004-06-08")
            description = SyndContentImpl()
            description.type = 'text/plain'
            description.value = 'Initial Release of ROME'
            entry.description = description
            entries.add(entry)

            entry = SyndEntryImpl()
            entry.title = 'ROME v2.0'
            entry.link = 'http://wiki.java.net/bin/view/Javawsxml/Rome02'
            entry.publishedDate = self.DATE_PARSER.parse("2004-06-16")
            description = SyndContentImpl()
            description.type = 'text/plain'
            description.value = 'Bug fixes, minor API changes and some new features'
            entry.description = description
            entries.add(entry)

            entry = SyndEntryImpl()
            entry.title = 'ROME v3.0'
            entry.link = 'http://wiki.java.net/bin/view/Javawsxml/Rome03'

```

```
entry.publishedDate = self.DATE_PARSER.parse("2004-07-27")
description = SyndContentImpl()
description.type = 'text/plain'
description.value = '<p>More Bug fixes, mor API changes, some new features_
↳and some Unit testing</p>'
entry.description = description
entries.add(entry)

feed.entries = entries
#####
# Publish the XML
#####
writer = FileWriter(self.fileName)
output = SyndFeedOutput()
output.output(feed,writer)
writer.close()

print('The feed has been written to the file')

ok = True

except Exception, e:
    print 'There has been an exception raised',e

if ok == False:
    print 'Feed Not Printed'

if __name__ == "__main__":
    #####
    # You must change his file location
    # if not using Windows environment
    #####
    writer = FeedWriter('rss_2.0', 'C:\\TEMP\\testRss.xml')
    writer.writeFeed()
    print '*****Command Complete...RSS XML has been_
↳created*****'
```

After you have created the XML, you'll obviously need to place it on a web server somewhere so that others can use your feed. The FeedWriter.py module would probably be one module amongst many in an application for creating and managing RSS Feeds, but you get the idea.

Conclusion

As you can see, using the ROME library to work with RSS feeds is quite easy. Using the ROME library within a Jython application is straight forward. As you have now seen how easy it is to create and parse feeds, you can apply these technologies to a more complete RSS management application if you'd like. The world of RSS communication is at your fingertips!

Author: Josh Juneau URL: <http://wiki.python.org/jython/JythonMonthly/Articles/October2007/1>

Using the CLASSPATH - Steve Langer

Introduction

During Oct-Nov 2006 there was a thread in the jython-users group titled "adding JARs to sys.path". More accurately the objective there was to add JARs to the sys.path at runtime. Several people asked the question, "Why would you want to do that?" Well there are at least 2 good reasons. First, if you want to distribute a jython or Java package

that includes non-standard Jars in it. Perhaps you want to make life easier for the target user and not demand that they know how to set environment variables. A second even more compelling reason is when there is no normal user account to provide environment variables.

“What?”, you ask. Well, in my case I came upon this problem in the following way. I am working on an open source IHE Image Archive Actor and needed a web interface. I’m using AJAX on the client side to route database calls through CGI to a jython-JDBC enabled API. Testing the jython-JDBC API from the command line worked fine, I had the PostGres driver in my CLASSPATH. But, when called via the web interface I got “zxJDBC error, postGres driver not found” errors. Why? Because APACHE was calling the API and APACHE is not a normal account with environment variables.

What to do?

The jython-users thread had many suggestions but none were found to work. For books, Chapter 11 of O’Reilly’s “Jython Essentials” mentions under “System and File Modules” that “... to load a class at runtime one also needs an appropriate class loader.” Of course, no mention is made beyond that. After a while, it occurred to me that perhaps someone in the Java world had found a similar problem and had solved it. Then all that would be required is to translate that solution. And that is exactly what happened.

Method

For brevity we will not repeat the original Java code here. This is how I call the Jython class (note that one can use either addFile or addURL depending on whether the Jar is on a locally accesable file system or remote server).

```
import sys
from com.ziclix.python.sql import zxJDBC

d,u,p,v = "jdbc:postgresql://localhost/img_arc2","postgres","","org.postgresql.Driver"

try :
    # if called from command line with .login CLASSPATH setup right,this works
    db = zxJDBC.connect(d, u, p, v)
except:
    # if called from Apache or account where the .login has not set CLASSPATH
    # need to use run-time CLASSPATH Hacker
    try :
        jarLoad = classPathHacker()
        a = jarLoad.addFile("/usr/share/java/postgresql-jdbc3.jar")
        db = zxJDBC.connect(d, u, p, v)
    except :
        sys.exit ("still failed \n%s" % (sys.exc_info() ))
```

And here is the class “classPathHacker” which is what the original author called his solution. In fact, you can simply Google on “classPathHacker” to find the Java solution.

```
class classPathHacker :
#####
# from http://forum.java.sun.com/thread.jspa?threadID=300557
#
# Author: SG Langer Jan 2007 translated the above Java to this
#       Jython class
# Purpose: Allow runtime additions of new Class/jars either from
#         local files or URL
#####
    import java.lang.reflect.Method
    import java.io.File
    import java.net.URL
    import java.net.URLClassLoader
    import jarray
```

```
def addFile (self, s):
#####
# Purpose: If adding a file/jar call this first
#         with s = path_to_jar
#####
        module = "utils:classPathHacker: addFile"

        # make a URL out of 's'
        f = self.java.io.File (s)
        u = f.toURL ()
        a = self.addURL (u)
        return a

def addURL (self, u):
#####
# Purpose: Call this with u= URL for
#         the new Class/jar to be loaded
#####
        module = "utils:classPathHacker: addURL"

        parameters = self.jarray.array([self.java.net.URL], self.java.lang.
↪Class)

        sysloader = self.java.lang.ClassLoader.getSystemClassLoader()
        sysclass = self.java.net.URLClassLoader
        method = sysclass.getDeclaredMethod("addURL", parameters)
        a = method.setAccessible(1)
        jar_a = self.jarray.array([u], self.java.lang.Object)
        b = method.invoke(sysloader, jar_a)
        return u
```

Conclusions

That's it. Depressingly short for what it does, but then that's another proof of the power of this language. I hope you find this as powerful and useful as I have. It allows the possibility of distributing jython packages with all their file dependencies within the installation directory, freeing the user or developer from the need to alter user environment variables, which should lead to more programmer control and thus higher reliability.

Author: Steve Langer URL: <http://wiki.python.org/jython/JythonMonthly/Articles/January2007/3>

Ant

The following Ant example works with Jython version 2.2.1 and earlier only due to the necessary jythonc usage. Jythonc is no longer distributed with Jython as of 2.5.0. This example could be re-written using object factories to work with current versions of Jython.

Writing Ant Tasks With Jython - Ed Takema

Ant is the current tool of choice for java builds. This is so partially because it was the first java oriented build tool on the scene and because the reigning champion *Make* was getting long in the tooth and had fallen out of favour with the java crowd. But Java builds are getting more and more difficult and these days there is general dissatisfaction with ant1. Note particularly Bruce Eckel's Comments and Martin Fowler's further comments. The comments to Bruce Eckel's posting show similar frustrations. Fowler summarizes the issues like this:

... Simple builds are easy to express as a series of tasks and dependencies. For such builds the facilities of ant/make work well. But more complex builds require conditional logic, and that requires more general programming language

constructs - and that's where ant/make fall down. Ken Arnold's article *The Sum of Ant* led me to Jonathon Simon's article *Scripting with Jython Instead of XML* and got me thinking about extending ant with Jython. Simon's article presents a technique to drive Ant tasks, testing, etc all from Jython. What I am presenting is a technique to embed Jython scripts into Ant which is admittedly backwards from Simon's approach, but hopefully adds power and flexibility to ant builds.

My experience working with large builds automated through ant is not dissimilar to what Fowler is referring to. Eventually, builds need to do either a lot of odd conditional logic in the xml file and ends up burying the logic in scripts, or in a large number of custom tasks written in java. This is particularly the case if your builds include non-java source that ant just isn't smart about building. In one case in particular, the set of custom tasks for the build is really its own system with maintenance and staff costs that are quite substantial. A large number of scripts can quickly become a problem for enterprise build systems as they are difficult to standardize and cross platform issues are always looming.

Fortunately, all is not lost. Ant continues to evolve and version 1.6 was a significant step forward for large build systems. Mike Spille, in his article *ANT's Finally a Real Build Tool*, demonstrates that the new `<import>` tag now allows build managers to write truly modular and standardized build systems based on Ant! As Ant grows up, more and more of these issues will get resolved.

One of the strengths that Make always had was the ability to easily call scripts and command utilities. This is something that is definitely possible with Ant script/exec tasks, but it feels very un-java. What we need is an elegant way to add adhoc behaviour to Ant builds ... in a java-ish way.

Writing Custom Ant Tasks

What I think can do the job is to take a more considered approach to using a scripting tool inside an ant build. Rather than just create a mishmash of scripts that are called from exec or script tasks, I suggest that we write custom ant build tasks in a high level scripting language...in this case, Jython.

Writing custom ant tasks allows a build manager to leverage the huge number of already written tasks in their builds while writing what naturally belongs in a more flexible tool in custom ant tasks that can themselves then be reused, are as cross platform as java itself, and wholly integrated into Ant. Because Ant uses java introspection to determine the capabilities of custom tasks, Jython is the perfect tool to accomplish this. All we need to do is ensure that the methods that Ant expects are present in the Jython classes and Ant won't notice the difference.

What we will implement is the perennial SimpleTask which is nothing more than a 'Hello World' for ant. It should be sufficient to demonstrate the key steps.

Setup Development Environment

To compile the jython source in this article you will need to add the ant.jar file to your classpath. This will make it available to Jython to extend which we'll do below. To do that define your classpath:

```
<DOS>
set CLASSPATH=c:\path\to\ant\lib\ant.jar
```

```
<UNIX>
export CLASSPATH=/path/to/ant/lib/ant.jar
```

SimpleTask Jython Class

The following is a very simple Ant task written in Jython(python). Save this as SimpleTask.py

```
from org.apache.tools.ant import Task

class SimpleTask(Task):

    message = ""
```

```
def execute(self):
    """@sig public void execute()"""
    Task.log(self, "Message: " + self.message)

def setMessage(this, aMessage):
    """@sig public void setMessage(java.lang.String str)"""
    this.message = aMessage
```

This simple Jython class extends the ant Task superclass. For each of the properties we want to support for this task, we write a setXXXXXX method where XXXXXX corresponds to the property we are going to set in the ant build file. Ant creates an object from the class, calls the setXXXXXX methods to setup the properties and then calls the execute method (actually, it calls the perform method on the Task superclass which calls the execute() method). So lets try it out.

Compiling Jython Code To A Jar

To build this into a jar file for use in Ant, do the following:

```
jythonc -a -c -d -j myTasks.jar SimpleTask.py
```

This will produce a jar file myTasks.jar and include the jython core support classes in the jar. Copy this jar file into your ant installation's lib directory. In my case I copy it to c:toolsantlib.

Build.XML file to use the Task

Once you've got that working, here is a very simple test ant build file to test your custom jython task.

```
<project name="ant jython demo" default="testit" basedir=".">

  <!-- Define the tasks we are building -->
  <taskdef name="Simple" classname="SimpleTask" />

  <!-- Test Case starts here -->
  <target name="testit">
    <Simple message="Hello World!" />
  </target>

</project>
```

A Task Container Task

All right, that is a pretty simple task. What else can we do? Well, the sky is the limit really. Here is an example of a task container. In this case, the task holds references to a set of other tasks (SimpleTask tasks in this case):

```
from org.apache.tools.ant import Task
from org.apache.tools.ant import TaskContainer

class SimpleContainer(TaskContainer):

    subtasks = []

    def execute(this):
        """@sig public void execute()"""

        for task in this.subtasks:
            task.perform()

    def createSimpleTask(self):
```



```

        """@sig public java.lang.Object createSimpleTask() """

        task = SimpleTask()
        self.subtasks.append(task)
        return task

class SimpleTask(Task):

    message = ""

    def execute(self):
        """@sig public void execute() """
        Task.log(self, "Message: " + self.message)

    def setMessage(this, aMessage):
        """@sig public void setMessage(java.lang.String str) """
        this.message = aMessage

```

The SimpleContainer extends the TaskContainer java class. Its createSimpleTask method creates a SimpleTask object and returns it to Ant so its properties can be set. Then when all the tasks have been added to the container and their properties set, the execute method on the SimpleContainer class is called which in turn calls the perform method on each of the contained tasks. Note that the perform method is inherited from the Task superclass and it in turn calls the execute method which we have overridden.

Build.XML file to use the TaskContainer

Here is a ant build file to test your custom jython task container. Note that you don't need to include a task definition for the contained SimpleTask unless you want to use it directly. The createSimpleTask factory method does it for you.

```

<project name="ant jython demo" default="testit" basedir=".">

    <!-- Define the tasks we are building -->
    <taskdef name="Container" classname="SimpleContainer" />

    <!-- Test Case starts here -->
    <target name="testit">

        <Container>

            <SimpleTask message="hello" />
            <SimpleTask message="there" />

        </Container>

    </target>

</project>

```

Things To Look Out For

As I learned this technique I discovered that the magic doc strings are really necessary to force Jython to put the right methods in the generated java classes. For example:

```

"""@sig public void execute() """

```

This is primarily due to Ant's introspection that looks for those specific methods and signatures. These docstrings are required or Ant won't recognize the classes as Ant tasks.

I also learned that for Jython to extend a java class, it must specifically import the java classes using this syntax:

```
from org.apache.tools.ant import Task
from org.apache.tools.ant import TaskContainer

class MyTask(Task):
    ...
You can not use this syntax:

import org.apache.tools.ant.Task
import org.apache.tools.ant.TaskContainer

class MyTask(org.apache.tools.ant.Task):
    ...
```

This is because, for some reason, Jython doesn't figure out that MyTask is extending this java class and so doesn't generate the right Java wrapper classes. You will know that this working right when you see output like the following when you run the jythonc compiler:

```
processing SimpleTask

Required packages:
  org.apache.tools.ant

Creating adapters:

Creating .java files:
  SimpleTask module
  SimpleTask extends org.apache.tools.ant.Task <<<
```

Summary

So there you have it. Here is a quick summary then of why this is a helpful technique.

First, it is a lot faster to write ant tasks that integrate with third party tools and systems using a glue language and python/jython is excellent at that. That is really my prime motivation for trying out this technique.

Secondly, Jython has the advantage over other scripting languages (which could be run using Ant's exec or script tasks) because it can be tightly integrated with Ant (i.e. use the same logging methods, same settings, etc). This makes it easier to build a standardized build environment.

Finally, and related to the last point, Jython can be compiled to java byte code which runs like any java class file. This means you don't have to have jython installed to use the custom tasks and your custom task, if written well, can run on a wide variety of platforms.

I think this is a reasonable way to add flexibility and additional integration points to Ant builds.

Author: Ed Taekema URL: <http://www.fishandcross.com/articles/AntTasksWithJython.html>

Attribution

Throughout the process of writing this book, many individuals from the community have taken time out to help us out. We'd like to give a special thanks to the Python and Jython community as a whole for actively participating in the feedback and contributions for this book via the mailing lists.

We also thank the Python and Jython developers for producing this great programming language. So much hard work has gone into the Python language that it has become one of the most solid and widely used object oriented programming languages available today. The Jython developers have done a tremendous job bringing the Jython of

today up-to-speed with current versions of the Python language. The Jython community seems to be working harder than ever before at making it a viable option for programming modern Python and Java applications alike. Special thanks to those developers who have helped out with this book and still kept Jython advancing.

The authors of The Definitive Guide to Jython would like to offer an extra special thanks to the rest of the individuals in this section. Without your insight or contributions portions of this book may not have been possible.

James Gardner - Thanks to James Gardner for his help in providing us with a working toolset and knowledge to convert from restructured text format to MS Word for the Apress editing process. The open source book was written completely in restructured text, and converted to .doc format for the Apress drafts and editing procedures. James offered great insight on how he used this process with The Pylons Book, and he also provided us with a conversion utility that was used for his book.

CHAPTER 7

Indices and tables

- `genindex`
- `modindex`
- `search`

P

Python Enhancement Proposals

PEP 236, [35](#)