Table of Contents

Table of Contents

Table of Contents

Dive Into Python

28 July 2002

This book lives at http://diveintopython.org/. If you're reading it somewhere else, you may not have the latest version.

Preface

This book is not for newbies, for wimps, or For Dummies. It assumes a lot about you.

- You know at least one real object−oriented language, like Java, C++, or Delphi.
- You know at least one scripting language, like Perl, Visual Basic, or JavaScript.
- You have already installed Python. See the home page for links to download Python for your favorite operating system. Python 2.0 or later is required; Python 2.2.1 is recommended. Where there are important differences between 2.0 and 2.2.1, they are clearly noted in the text.
- You have downloaded the example programs used in this book.

If you're just getting started programming, that does not mean that you can't learn Python. Python is an easy language to learn, but you should probably learn it somewhere else. I highly recommend *Learning to Program* and *How to Think Like a Computer Scientist*, and Python.org has links to other introductions to Python programming for non−programmers.

Let's dive in.

Chapter 1. Getting To Know Python

1.1. Diving in

Here is a complete, working Python program.

It probably makes absolutely no sense to you. Don't worry about that; we're going to dissect it line by line. But read through it first and see what, if anything, you can make of it.

Example 1.1. `odbchelper.py`

If you have not already done so, you can download this and other examples used in this book.

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Now run this program and see what happens.

> **Tip: Run module (Windows)**
> In the Python IDE on Windows, you can run a module with File−>Run... (**Ctrl−R**). Output is displayed in the interactive window.
>
> **Tip: Run module (Mac OS)**
> In the Python IDE on Mac OS, you can run a module with Python−>Run window... (**Cmd−R**), but there is an important option you must set first. Open the module in the IDE, pop up the module's options menu by clicking the black triangle in the upper−right corner of the window, and make sure "Run as __main__" is checked. This setting is saved with the module, so you only have to do this once per module.
>
> **Tip: Run module (UNIX)**
> On UNIX−compatible systems (including Mac OS X), you can run a module from the command line: **`python odbchelper.py`**

Example 1.2. Output of `odbchelper.py`

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

1.2. Declaring functions

Python has functions like most other languages, but it does not have separate header files like C++ or `interface`/`implementation` sections like Pascal. When you need a function, just declare it and code it.

Example 1.3. Declaring the `buildConnectionString` function

```
def buildConnectionString(params):
```

Several things to note here. First, the keyword `def` starts the function declaration, followed by the function name, followed by the arguments in parentheses. Multiple arguments (not shown here) are separated with commas.

Second, the function doesn't define a return datatype. Python functions do not specify the datatype of their return value; they don't even specify whether they return a value or not. In fact, every Python function returns a value; if the function ever executes a `return` statement, it will return that value, otherwise it will return `None`, the Python null value.

> **Note: Python vs. Visual Basic: return values**
> In Visual Basic, functions (that return a value) start with `function`, and subroutines (that do not return a value) start with `sub`. There are no subroutines in Python. Everything is a function, all functions return a value (even if it's `None`), and all functions start with `def`.

Third, the argument, `params`, doesn't specify a datatype. In Python, variables are never explicitly typed. Python figures out what type a variable is and keeps track of it internally.

> **Note: Python vs. Java: return values**
> In Java, C++, and other statically−typed languages, you must specify the datatype of the function return value and each function argument. In Python, you never explicitly specify the datatype of anything. Based on what value you assign, Python keeps track of the datatype internally.

Addendum. An erudite reader sent me this explanation of how Python compares to other programming languages:

statically typed language
: A language in which types are fixed at compile time. Most statically typed languages enforce this by requiring you to declare all variables with their datatypes before using them. Java and C are statically typed languages.

dynamically typed language
: A language in which types are discovered at execution time; the opposite of statically typed. VBScript and Python are dynamically typed, because they figure out what type a variable is when you first assign it a value.

strongly typed language
: A language in which types are always enforced. Java and Python are strongly typed. If you have an integer, you can't treat it like a string without explicitly converting it (more on how to do this later in this chapter).

weakly typed language
: A language in which types may be ignored; the opposite of strongly typed. VBScript is weakly typed. In VBScript, you can concatenate the string `'12'` and the integer `3` to get the string `'123'`, then treat that as the integer `123`, all without any explicit conversion.

So Python is both *dynamically typed* (because it doesn't use explicit datatype declarations) and *strongly typed* (because once a variable has a datatype, it actually matters).

1.3. Documenting functions

You can document a Python function by giving it a `doc string`.

Example 1.4. Defining the `buildConnectionString` function's `doc string`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
```

Triple quotes signify a multi−line string. Everything between the start and end quotes is part of a single string, including carriage returns and other quote characters. You can use them anywhere, but you'll see them most often used when defining a `doc string`.

> **Note: Python vs. Perl: quoting**
> Triple quotes are also an easy way to define a string with both single and double quotes, like `qq/.../` in Perl.

Everything between the triple quotes is the function's `doc string`, which documents what the function does. A `doc string`, if it exists, must be the first thing defined in a function (*i.e.* the first thing after the colon). You don't technically have to give your function a `doc string`, but you always should. I know you've heard this in every programming class you've ever taken, but Python gives you an added incentive: the `doc string` is available at runtime as an attribute of the function.

> **Note: Why doc strings are a Good Thing**
> Many Python IDEs use the `doc string` to provide context−sensitive documentation, so that when you type a function name, its `doc string` appears as a tooltip. This can be incredibly helpful, but it's only as good as the `doc string`s you write.

Further reading

- *Python Style Guide* discusses how to write a good `doc string`.
- *Python Tutorial* discusses conventions for spacing in `doc strings`.

1.4. Everything is an object

In case you missed it, I just said that Python functions have attributes, and that those attributes are available at runtime.

A function, like everything else in Python, is an object.

Example 1.5. Accessing the `buildConnectionString` function's `doc string`

```
>>> import odbchelper                              (1)
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> print odbchelper.buildConnectionString(params) (2)
server=mpilgrim;uid=sa;database=master;pwd=secret
>>> print odbchelper.buildConnectionString.__doc__ (3)
Build a connection string from a dictionary

Returns string.
```

(1) The first line imports the `odbchelper` program as a module. Once you import a module, you can reference any of its public functions, classes, or attributes. Modules can do this to access functionality in other modules, and you can do it in the IDE too. This is an important concept, and we'll talk more about it later.

(2) When you want to use functions defined in imported modules, you have to include the module name. So you can't just say `buildConnectionString`, it has to be `odbchelper.buildConnectionString`. If you've used classes in Java, this should feel vaguely familiar.

(3) Instead of calling the function like you would expect to, we asked for one of the function's attributes, `__doc__`.

> **Note: Python vs. Perl: import**
> `import` in Python is like `require` in Perl. Once you `import` a Python module, you access its functions with *`module.function`*; once you `require` a Perl module, you access its functions with *`module`*`::`*`function`*.

Before we go any further, I want to briefly mention the library search path. Python looks in several places when you try to import a module. Specifically, it looks in all the directories defined in `sys.path`. This is just a list, and you can easily view it or modify it with standard list methods. (We'll learn more about lists later in this chapter.)

Example 1.6. Import search path

```
>>> import sys                 (1)
>>> sys.path                   (2)
['', '/usr/local/lib/python2.2', '/usr/local/lib/python2.2/plat-linux2',
'/usr/local/lib/python2.2/lib-dynload', '/usr/local/lib/python2.2/site-packages',
'/usr/local/lib/python2.2/site-packages/PIL', '/usr/local/lib/python2.2/site-packages/piddle']
>>> sys                        (3)
<module 'sys' (built-in)>
>>> sys.path.append('/my/new/path') (4)
```

(1) Importing the `sys` module makes all of its functions and attributes available.

(2) `sys.path` is a list of directory names that constitute the current search path. (Yours will look different, depending on your operating system, what version of Python you're running, and where it was originally installed.) Python will look through these directories (in this order) for a `.py` file matching the module name you're trying to import.

(3) Actually, I lied; the truth is more complicated than that, because not all modules are stored as `.py` files. Some, like the `sys` module, are "built−in modules"; they are actually baked right into Python itself. Built−in modules behave just like regular modules, but their Python source code is not available, because they are not written in Python! (The `sys` module is written in C.)

(4) You can add a new directory to Python's search path at runtime by appending the directory name to `sys.path`, and then Python will look in that directory as well, whenever you try to import a module. The effect lasts as long as Python is running. (We'll talk more about `append` and other list methods later in this chapter.)

Everything in Python is an object, and almost everything has attributes and methods.[1] All functions have a built−in attribute `__doc__`, which returns the `doc string` defined in the function's source code. The `sys` module is an object which has (among other things) an attribute called `path`. And so forth.

This is so important that I'm going to repeat it in case you missed it the first few times: *everything in Python is an object*. Strings are objects. Lists are objects. Functions are objects. Even modules are objects.

Further reading

- *Python Reference Manual* explains exactly what it means to say that everything in Python is an object, because some people are pedantic and like to discuss this sort of thing at great length.
- eff−bot summarizes Python objects.

1.5. Indenting code

Python functions have no explicit `begin` or `end`, no curly braces that would mark where the function code starts and stops. The only delimiter is a colon (":") and the indentation of the code itself.

Example 1.7. Indenting the `buildConnectionString` function

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Code blocks (functions, `if` statements, `for` loops, etc.) are defined by their indentation. Indenting starts a block and unindenting ends it; there are no explicit braces, brackets, or keywords. This means that whitespace is significant, and must be consistent. In this example, the function code (including the `doc string`) is indented 4 spaces. It doesn't have to be 4, it just has to be consistent. The first line that is not indented is outside the function.

After some initial protests and several snide analogies to Fortran, you will make peace with this and start seeing its benefits. One major benefit is that all Python programs look similar, since indentation is a language requirement and not a matter of style. This makes it easier to read and understand other people's Python code.

> **Note: Python vs. Java: separating statements**
> Python uses carriage returns to separate statements and a colon and indentation to separate code blocks. C++ and Java use semicolons to separate statements and curly braces to separate code blocks.

Further reading

- *Python Reference Manual* discusses cross−platform indentation issues and shows various indentation errors.
- *Python Style Guide* discusses good indentation style.

1.6. Testing modules

Python modules are objects and have several useful attributes. You can use this to easily test your modules as you write them.

Example 1.8. The `if __name__` trick

```
if __name__ == "__main__":
```

Some quick observations before we get to the good stuff. First, parentheses are not required around the `if` expression. Second, the `if` statement ends with a colon, and is followed by indented code.

> **Note: Python vs. C: comparison and assignment**
> Like C, Python uses == for comparison and = for assignment. Unlike C, Python does not support in−line assignment, so there's no chance of accidentally assigning the value you thought you were comparing.

So why is this particular `if` statement a trick? Modules are objects, and all modules have a built−in attribute `__name__`. A module's `__name__` depends on how you're using the module. If you `import` the module, then `__name__` is the module's filename, without directory path or file extension. But you can also run the module directly as a standalone program, in which case `__name__` will be a special default value, `__main__`.

Example 1.9. An imported module's `__name__`

```
>>> import odbchelper
>>> odbchelper.__name__
'odbchelper'
```

Knowing this, you can design a test suite for your module within the module itself by putting it in this `if` statement. When you run the module directly, `__name__` is `__main__`, so the test suite executes. When you import the module, `__name__` is something else, so the test suite is ignored. This makes it easier to develop and debug new modules before integrating them into a larger program.

> **Tip: if __name__ on Mac OS**
> On MacPython, there is an additional step to make the `if __name__` trick work. Pop up the module's options menu by clicking the black triangle in the upper−right corner of the window, and make sure Run as __main__ is checked.

Further reading

- *Python Reference Manual* discusses the low−level details of importing modules.

1.7. Introducing dictionaries

A short digression is in order, because you need to know about dictionaries, tuples, and lists (oh my!). If you're a Perl hacker, you can probably skim the bits about dictionaries and lists, but you should still pay attention to tuples.

One of Python's built−in datatypes is the dictionary, which defines one−to−one relationships between keys and values.

> **Note: Python vs. Perl: dictionaries**
> A dictionary in Python is like a hash in Perl. In Perl, variables which store hashes always start with a `%` character; in Python, variables can be named anything, and Python keeps track of the datatype internally.

> **Note: Python vs. Java: dictionaries**
> A dictionary in Python is like an instance of the `Hashtable` class in Java.

> **Note: Python vs. Visual Basic: dictionaries**
> A dictionary in Python is like an instance of the `Scripting.Dictionary` object in Visual Basic.

Example 1.10. Defining a dictionary

```
>>> d = {"server":"mpilgrim", "database":"master"} (1)
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]                                    (2)
'mpilgrim'
>>> d["database"]                                  (3)
'master'
>>> d["mpilgrim"]                                  (4)
Traceback (innermost last):
```

```
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```

(1) First, we create a new dictionary with two elements and assign it to the variable `d`. Each element is a key−value pair, and the whole set of elements is enclosed in curly braces.

(2) `'server'` is a key, and its associated value, referenced by `d["server"]`, is `'mpilgrim'`.

(3) `'database'` is a key, and its associated value, referenced by `d["database"]`, is `'master'`.

(4) You can get values by key, but you can't get keys by value. So `d["server"]` is `'mpilgrim'`, but `d["mpilgrim"]` raises an exception, because `'mpilgrim'` is not a key.

Example 1.11. Modifying a dictionary

```
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["database"] = "pubs" (1)
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa"        (2)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

(1) You can not have duplicate keys in a dictionary. Assigning a value to an existing key will wipe out the old value.

(2) You can add new key−value pairs at any time. This syntax is identical to modifying existing values. (Yes, this will annoy you someday when you think you are adding new values but are actually just modifying the same value over and over because your key isn't changing the way you think it is.)

Note that the new element (key `'uid'`, value `'sa'`) appears to be in the middle. In fact, it was just a coincidence that the elements appeared to be in order in the first example; it is just as much a coincidence that they appear to be out of order now.

> **Note: Dictionaries are unordered**
> Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered. This is an important distinction which will annoy you when you want to access the elements of a dictionary in a specific, repeatable order (like alphabetical order by key). There are ways of doing this, they're just not built into the dictionary.

Example 1.12. Mixing datatypes in a dictionary

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
>>> d["retrycount"] = 3 (1)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d[42] = "douglas"   (2)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 42: 'douglas', 'retrycount': 3}
```

(1) Dictionaries aren't just for strings. Dictionary values can be any datatype, including strings, integers, objects, or even other dictionaries. And within a single dictionary, the values don't all have to be the same type; you can mix and match as needed.

(2) Dictionary keys are more restricted, but they can be strings, integers, and a few other types (more on this later). You can also mix and match key datatypes within a dictionary.

Example 1.13. Deleting items from a dictionary

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 42: 'douglas', 'retrycount': 3}
>>> del d[42] (1)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d.clear() (2)
>>> d
{}
```

(1) `del` lets you delete individual items from a dictionary by key.

(2) `clear` deletes all items from a dictionary. Note that the set of empty curly braces signifies a dictionary with no items.

Example 1.14. Strings are case−sensitive

```
>>> d = {}
>>> d["key"] = "value"
>>> d["key"] = "other value" (1)
>>> d
{'key': 'other value'}
>>> d["Key"] = "third value" (2)
>>> d
{'Key': 'third value', 'key': 'other value'}
```

(1) Assigning a value to an existing dictionary key simply replaces the old value with a new one.

(2) This is not assigning a value to an existing dictionary key, because strings in Python are case−sensitive, so `'key'` is not the same as `'Key'`. This creates a new key/value pair in the dictionary; it may look similar to you, but as far as Python is concerned, it's completely different.

Further reading

- *How to Think Like a Computer Scientist* teaches about dictionaries and shows how to use dictionaries to model sparse matrices.
- Python Knowledge Base has lots of example code using dictionaries.
- Python Cookbook discusses how to sort the values of a dictionary by key.
- *Python Library Reference* summarizes all the dictionary methods.

1.8. Introducing lists

Lists are Python's workhorse datatype. If your only experience with lists is arrays in Visual Basic or (God forbid) the datastore in Powerbuilder, brace yourself for Python lists.

> **Note: Python vs. Perl: lists**
> A list in Python is like an array in Perl. In Perl, variables which store arrays always start with the `@` character; in Python, variables can be named anything, and Python keeps track of the datatype internally.

> **Note: Python vs. Java: lists**
> A list in Python is much more than an array in Java (although it can be used as one if that's really all you want out of life). A better analogy would be to the `Vector` class, which can hold arbitrary objects and can expand dynamically as new items are added.

Example 1.15. Defining a list

```
>>> li = ["a", "b", "mpilgrim", "z", "example"] (1)
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[0]                                       (2)
'a'
>>> li[4]                                       (3)
'example'
```

(1) First, we define a list of 5 elements. Note that they retain their original order. This is not an accident. A list is an ordered set of elements enclosed in square brackets.

(2) A list can be used like a zero−based array. The first element of any non−empty list is always `li[0]`.

(3) The last element of this 5−element list is `li[4]`, because lists are always zero−based.

Example 1.16. Negative list indices

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[-1] (1)
'example'
>>> li[-3] (2)
'mpilgrim'
```

(1) A negative index accesses elements from the end of the list counting backwards. The last element of any non−empty list is always `li[-1]`.

(2) If negative indices are confusing to you, think of it this way: `li[-n] == li[len(li) - n]`. So in this list, `li[-3] == li[5 - 3] == li[2]`.

Example 1.17. Slicing a list

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[1:3]  (1)
['b', 'mpilgrim']
>>> li[1:-1] (2)
['b', 'mpilgrim', 'z']
>>> li[0:3]  (3)
['a', 'b', 'mpilgrim']
```

(1) You can get a subset of a list, called a "slice", by specifying 2 indices. The return value is a new list containing all the elements of the list, in order, starting with the first slice index (in this case `li[1]`), up to but not including the second slice index (in this case `li[3]`).

(2) Slicing works if one or both of the slice indices is negative. If it helps, you can think of it this way: reading the list from left to right, the first slice index specifies the first element you want, and the second slice index specifies the first element you don't want. The return value is everything in between.

(3) Lists are zero−based, so `li[0:3]` returns the first three elements of the list, starting at `li[0]`, up to but not including `li[3]`.

Example 1.18. Slicing shorthand

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[:3] (1)
['a', 'b', 'mpilgrim']
```

```
>>> li[3:] (2) (3)
['z', 'example']
>>> li[:]  (4)
['a', 'b', 'mpilgrim', 'z', 'example']
```

(1) If the left slice index is 0, you can leave it out, and 0 is implied. So `li[:3]` is the same as `li[0:3]` from the previous example.

(2) Similarly, if the right slice index is the length of the list, you can leave it out. So `li[3:]` is the same as `li[3:5]`, because this list has 5 elements.

(3) Note the symmetry here. In this 5−element list, `li[:3]` returns the first 3 elements, and `li[3:]` returns the last 2 elements. In fact, `li[:n]` will always return the first `n` elements, and `li[n:]` will return the rest, regardless of the length of the list.

(4) If both slice indices are left out, all elements of the list are included. But this is not the same as the original `li` list; it is a new list that happens to have all the same elements. `li[:]` is a shorthand for making a complete copy of a list.

Example 1.19. Adding elements to a list

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li.append("new")              (1)
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li.insert(2, "new")           (2)
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> li.extend(["two", "elements"]) (3)
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
```

(1) `append` adds a single element to the end of the list.

(2) `insert` inserts a single element into a list. The numeric argument is the index of the first element that gets bumped out of position. Note that list elements do not have to be unique; there are now 2 separate elements with the value `'new'`, `li[2]` and `li[6]`.

(3) `extend` concatenates lists. Note that you do not call `extend` with multiple arguments; you call it with one argument, a list. In this case, that list has two elements.

Example 1.20. Searching a list

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.index("example") (1)
5
>>> li.index("new")     (2)
2
>>> li.index("c")       (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li           (4)
0
```

(1) `index` finds the first occurrence of a value in the list and returns the index.

(2) `index` finds the *first* occurrence of a value in the list. In this case, `'new'` occurs twice in the list, in `li[2]` and `li[6]`, but `index` will only return the first index, `2`.

(3) If the value is not found in the list, Python raises an exception. This is notably different from most languages, which will return some invalid index. While this may seem annoying, it is a Good Thing, because it means your program will crash at the source of the problem, rather than later on when you try to use the invalid index.

(4) To test whether a value is in the list, use `in`, which returns `1` if the value is found or `0` if it is not.

Note: What's true in Python?
Before version 2.2.1, Python had no separate boolean datatype. To compensate for this, Python accepted almost anything in a boolean context (like an `if` statement), according to the following rules: `0` is false; all other numbers are true. An empty string (`""`) is false, all other strings are true. An empty list (`[]`) is false; all other lists are true. An empty tuple (`()`) is false; all other tuples are true. An empty dictionary (`{}`) is false; all other dictionaries are true. These rules still apply in Python 2.2.1 and beyond, but now you can also use an actual boolean, which has a value of `True` or `False`. Note the capitalization; these values, like everything else in Python, are case−sensitive.

Example 1.21. Removing elements from a list

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("z")   (1)
>>> li
['a', 'b', 'new', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("new") (2)
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("c")   (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()         (4)
'elements'
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
```

(1) `remove` removes the first occurrence of a value from a list.

(2) `remove` removes *only* the first occurrence of a value. In this case, `'new'` appeared twice in the list, but `li.remove("new")` only removed the first occurrence.

(3) If the value is not found in the list, Python raises an exception. This mirrors the behavior of the `index` method.

(4) `pop` is an interesting beast. It does two things: it removes the last element of the list, and it returns the value that it removed. Note that this is different from `li[−1]`, which returns a value but does not change the list, and different from `li.remove(`*`value`*`)`, which changes the list but does not return a value.

Example 1.22. List operators

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new'] (1)
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
>>> li += ['two']                (2)
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
>>> li = [1, 2] * 3              (3)
>>> li
[1, 2, 1, 2, 1, 2]
```

(1)

Lists can also be concatenated with the + operator. *list* = *list* + *otherlist* has the same result as *list*.extend(*otherlist*). But the + operator returns a new (concatenated) list as a value, whereas extend only alters an existing list. This means that extend is faster, especially for large lists.

(2) Python supports the += operator. li += ['two'] is equivalent to li.extend(['two']). The += operator works for lists, strings, and integers, and it can be overloaded to work for user−defined classes as well. (More on classes in chapter 3.)

(3) The * operator works on lists as a repeater. li = [1, 2] * 3 is equivalent to li = [1, 2] + [1, 2] + [1, 2], which concatenates the three lists into one.

Further reading

- *How to Think Like a Computer Scientist* teaches about lists and makes an important point about passing lists as function arguments.
- *Python Tutorial* shows how to use lists as stacks and queues.
- Python Knowledge Base answers common questions about lists and has lots of example code using lists.
- *Python Library Reference* summarizes all the list methods.

1.9. Introducing tuples

A tuple is an immutable list. A tuple can not be changed in any way once it is created.

Example 1.23. Defining a tuple

```
>>> t = ("a", "b", "mpilgrim", "z", "example") (1)
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]                                       (2)
'a'
>>> t[−1]                                      (3)
'example'
>>> t[1:3]                                     (4)
('b', 'mpilgrim')
```

(1) A tuple is defined in the same way as a list, except that the whole set of elements is enclosed in parentheses instead of square brackets.

(2) The elements of a tuple have a defined order, just like a list. Tuples indices are zero−based, just like a list, so the first element of a non−empty tuple is always t[0].

(3) Negative indices count from the end of the tuple, just like a list.

(4) Slicing works too, just like a list. Note that when you slice a list, you get a new list; when you slice a tuple, you get a new tuple.

Example 1.24. Tuples have no methods

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new")    (1)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")      (2)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
```

```
>>> t.index("example") (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t           (4)
1
```

(1) You can't add elements to a tuple. Tuples have no `append` or `extend` method.

(2) You can't remove elements from a tuple. Tuples have no `remove` or `pop` method.

(3) You can't find elements in a tuple. Tuples have no `index` method.

(4) You can, however, use `in` to see if an element exists in the tuple.

So what are tuples good for?

- Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.
- Remember I said that dictionary keys can be integers, strings, and "a few other types"? Tuples are one of those types. Tuples can be used as keys in a dictionary, but lists can't.[2]
- Tuples are used in string formatting, as we'll see shortly.

Note: Tuples into lists into tuples
Tuples can be converted into lists, and vice−versa. The built−in `tuple` function takes a list and returns a tuple with the same elements, and the `list` function takes a tuple and returns a list. In effect, `tuple` freezes a list, and `list` thaws a tuple.

Further reading

- *How to Think Like a Computer Scientist* teaches about tuples and shows how to concatenate tuples.
- Python Knowledge Base shows how to sort a tuple.
- *Python Tutorial* shows how to define a tuple with one element.

1.10. Defining variables

Now that you think you know everything about dictionaries, tuples, and lists (oh my!), let's get back to our example program, `odbchelper.py`.

Python has local and global variables like most other languages, but it has no explicit variable declarations. Variables spring into existence by being assigned a value, and are automatically destroyed when they go out of scope.

Example 1.25. Defining the `myParams` variable

```
if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
```

Several points of interest here. First, note the indentation. An `if` statement is a code block and needs to be indented just like a function.

Second, the variable assignment is one command split over several lines, with a backslash ("\") serving as a line continuation marker.

> **Note: Multiline commands**
> When a command is split among several lines with the line continuation marker ("\"), the continued lines can be indented in any manner; Python's normally stringent indentation rules do not apply. If your Python IDE auto−indents the continued line, you should probably accept its default unless you have a burning reason not to.

> **Note: Implicit multiline commands**
> Strictly speaking, expressions in parentheses, straight brackets, or curly braces (like defining a dictionary) can be split into multiple lines with or without the line continuation character ("\"). I like to include the backslash even when it's not required because I think it makes the code easier to read, but that's a matter of style.

Third, you never declared the variable `myParams`, you just assigned a value to it. This is like VBScript without the `option explicit` option. Luckily, unlike VBScript, Python will not allow you to reference a variable that has never been assigned a value; trying to do so will raise an exception.

Example 1.26. Referencing an unbound variable

```
>>> x
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'x'
>>> x = 1
>>> x
1
```

You will thank Python for this one day.

Further reading

- *Python Reference Manual* shows examples of when you can skip the line continuation character and when you have to use it.

1.11. Assigning multiple values at once

One of the cooler programming shortcuts in Python is using sequences to assign multiple values at once.

Example 1.27. Assigning multiple values at once

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v (1)
>>> x
'a'
>>> y
'b'
>>> z
'e'
```

(1) `v` is a tuple of three elements, and `(x, y, z)` is a tuple of three variables. Assigning one to the other assigns each of the values of `v` to each of the variables, in order.

This has all sorts of uses. I often want to assign names to a range of values. In C, you would use `enum` and manually list each constant and its associated value, which seems especially tedious when the values are consecutive. In Python, you can use the built−in `range` function with multi−variable assignment to quickly assign consecutive values.

Example 1.28. Assigning consecutive values

```
>>> range(7)                                                                    (1)
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7) (2)
>>> MONDAY                                                                      (3)
0
>>> TUESDAY
1
>>> SUNDAY
6
```

(1) The built−in `range` function returns a list of integers. In its simplest form, it takes an upper limit and returns a `0`−based list counting up to but not including the upper limit. (If you like, you can pass other parameters to specify a base other than `0` and a step other than `1`. You can `print range.__doc__` for details.)

(2) `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY` are the variables we're defining. (This example came from the `calendar` module, a fun little module which prints calendars, like the UNIX program `cal`. The `calendar` module defines integer constants for days of the week.)

(3) Now each variable has its value: `MONDAY` is `0`, `TUESDAY` is `1`, and so forth.

You can also use multi−variable assignment to build functions that return multiple values, simply by returning a tuple of all the values. The caller can treat it as a tuple, or assign the values to individual variables. Many standard Python libraries do this, including the `os` module, which we'll discuss in chapter 3.

Further reading

- *How to Think Like a Computer Scientist* shows how to use multi−variable assignment to swap the values of two variables.

1.12. Formatting strings

Python supports formatting values into strings. Although this can include very complicated expressions, the most basic usage is to insert values into a string with the `%s` placeholder.

> **Note: Python vs. C: string formatting**
> String formatting in Python uses the same syntax as the `sprintf` function in C.

Example 1.29. Introducing string formatting

```
>>> k = "uid"
>>> v = "sa"
>>> "%s=%s" % (k, v) (1)
'uid=sa'
```

(1) The whole expression evaluates to a string. The first `%s` is replaced by the value of `k`; the second `%s` is replaced by the value of `v`. All other characters in the string (in this case, the equals sign) stay as they are.

Note that `(k, v)` is a tuple. I told you they were good for something.

You might be thinking that this is a lot of work just to do simple string concatentation, and you'd be right, except that string formatting isn't just concatenation. It's not even just formatting. It's also type coercion.

Example 1.30. String formatting vs. concatenating

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " is not a good password for " + uid      (1)
secret is not a good password for sa
>>> print "%s is not a good password for %s" % (pwd, uid) (2)
secret is not a good password for sa
>>> userCount = 6
>>> print "Users connected: %d" % (userCount, )           (3) (4)
Users connected: 6
>>> print "Users connected: " + userCount                 (5)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot add type "int" to string
```

(1) `+` is the string concatenation operator.

(2) In this trivial case, string formatting accomplishes the same result as concatentation.

(3) `(userCount, )` is a tuple with one element. Yes, the syntax is a little strange, but there's a good reason for it: it's unambiguously a tuple. In fact, you can always include a comma after the last element when defining a list, tuple, or dictionary, but the comma is required when defining a tuple with one element. If the comma weren't required, Python wouldn't know whether `(userCount)` was a tuple with one element or just the value of `userCount`.

(4) String formatting works with integers by specifying `%d` instead of `%s`.

(5) Trying to concatenate a string with a non−string raises an exception. Unlike string formatting, string concatenation only works when everything is already a string.

Further reading

- *Python Library Reference* summarizes all the string formatting format characters.
- *Effective AWK Programming* discusses all the format characters and advanced string formatting techniques like specifying width, precision, and zero−padding.

1.13. Mapping lists

One of the most powerful features of Python is the list comprehension, which provides a compact way of mapping a list into another list by applying a function to each of the elements of the list.

Example 1.31. Introducing list comprehensions

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]      (1)
[2, 18, 16, 8]
>>> li                           (2)
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li] (3)
>>> li
[2, 18, 16, 8]
```

(1)

To make sense of this, look at it from right to left. `li` is the list you're mapping. Python loops through `li` one element at a time, temporarily assigning the value of each element to the variable `elem`. Python then applies the function `elem*2` and appends that result to the returned list.

(2) Note that list comprehensions do not change the original list.

(3) It is safe to assign the result of a list comprehension to the variable that you're mapping. There are no racing conditions or any weirdness to worry about; Python constructs the new list in memory, and when the list comprehension is complete, it assigns the result to the variable.

Example 1.32. List comprehensions in `buildConnectionString`

```
["%s=%s" % (k, v) for k, v in params.items()]
```

First, notice that you're calling the `items` function of the `params` dictionary. This function returns a list of tuples of all the data in the dictionary.

Example 1.33. `keys`, `values`, and `items`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.keys()   (1)
['server', 'uid', 'database', 'pwd']
>>> params.values() (2)
['mpilgrim', 'sa', 'master', 'secret']
>>> params.items()  (3)
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
```

(1) The `keys` method of a dictionary returns a list of all the keys. The list is not in the order in which the dictionary was defined (remember, elements in a dictionary are unordered), but it is a list.

(2) The `values` method returns a list of all the values. The list is in the same order as the list returned by `keys`, so `params.values()[n] == params[params.keys()[n]]` for all values of `n`.

(3) The `items` method returns a list of tuples of the form `(`*`key`*`, `*`value`*`)`. The list contains all the data in the dictionary.

Now let's see what `buildConnectionString` does. It takes a list, `params.items()`, and maps it to a new list by applying string formatting to each element. The new list will have the same number of elements as `params.items()`, but each element in the new list will be a string that contains both a key and its associated value from the `params` dictionary.

Example 1.34. List comprehensions in `buildConnectionString`, step by step

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.items()
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
>>> [k for k, v in params.items()]                (1)
['server', 'uid', 'database', 'pwd']
>>> [v for k, v in params.items()]                (2)
['mpilgrim', 'sa', 'master', 'secret']
>>> ["%s=%s" % (k, v) for k, v in params.items()] (3)
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

(1) Note that we're using two variables to iterate through the `params.items()` list. This is another use of multi−variable assignment. The first element of `params.items()` is `('server', 'mpilgrim')`, so in

the first iteration of the list comprehension, `k` will get `'server'` and `v` will get `'mpilgrim'`. In this case we're ignoring the value of `v` and only including the value of `k` in the returned list, so this list comprehension ends up being equivalent to `params.keys()`. (You wouldn't really use a list comprehension like this in real code; this is an overly simplistic example so you can get your head around what's going on here.)

(2) Here we're doing the same thing, but ignoring the value of `k`, so this list comprehension ends up being equivalent to `params.values()`.

(3) Combining the previous two examples with some simple string formatting, we get a list of strings that include both the key and value of each element of the dictionary. This looks suspiciously like the output of the program; all that remains is to join the elements in this list into a single string.

Further reading

- *Python Tutorial* discusses another way to map lists using the built−in `map` function.
- *Python Tutorial* shows how to do nested list comprehensions.

1.14. Joining lists and splitting strings

You have a list of key−value pairs in the form *`key=value`*, and you want to join them into a single string. To join any list of strings into a single string, use the `join` method of a string object.

Example 1.35. Joining a list in `buildConnectionString`

```
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

One interesting note before we continue. I keep repeating that functions are objects, strings are objects, everything is an object. You might have thought I meant that string *variables* are objects. But no, look closely at this example and you'll see that the string `";"` itself is an object, and you are calling its `join` method.

Anyway, the `join` method joins the elements of the list into a single string, with each element separated by a semi−colon. The delimiter doesn't have to be a semi−colon; it doesn't even have to be a single character. It can be any string.

> **Important: You can't join non−strings**
> `join` only works on lists of strings; it does not do any type coercion. `join`ing a list that has one or more non−string elements will raise an exception.

Example 1.36. Output of `odbchelper.py`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> ";".join(["%s=%s" % (k, v) for k, v in params.items()])
'server=mpilgrim;uid=sa;database=master;pwd=secret'
```

This string is then returned from the `help` function and printed by the calling block, which gives you the output that you marveled at when you started reading this chapter.

Historical note. When I first learned Python, I expected `join` to be a method of a list, which would take the delimiter as an argument. Lots of people feel the same way, and there's a story behind the `join` method. Prior to Python 1.6, strings didn't have all these useful methods. There was a separate `string` module which contained all the string functions; each function took a string as its first argument. The functions were deemed important enough to put onto the strings themselves, which made sense for functions like `lower`, `upper`, and `split`. But many hard−core Python programmers objected to the new `join` method, arguing that it should be a method of the list instead, or that it shouldn't move at all but simply stay a part of the old `string` module (which still has lots of useful stuff in it). I use the new `join` method exclusively, but you will see code written either way, and if it really bothers you, you can use the old `string.join` function instead.

You're probably wondering if there's an analogous method to split a string into a list. And of course there is, and it's called `split`.

Example 1.37. Splitting a string

```
>>> li = ['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s = ";".join(li)
>>> s
'server=mpilgrim;uid=sa;database=master;pwd=secret'
>>> s.split(";")    (1)
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s.split(";", 1) (2)
['server=mpilgrim', 'uid=sa;database=master;pwd=secret']
```

(1) `split` reverses `join` by splitting a string into a multi−element list. Note that the delimiter (";") is stripped out completely; it does not appear in any of the elements of the returned list.

(2) `split` takes an optional second argument, which is the number of times to split. (""Oooooh, optional arguments..." You'll learn how to do this in your own functions in the next chapter.)

Note: Searching with split
`anystring``.split(`*`delimiter`*`, 1)` is a useful technique when you want to search a string for a substring and then work with everything before the substring (which ends up in the first element of the returned list) and everything after it (which ends up in the second element).

Further reading

- Python Knowledge Base answers common questions about strings and has lots of example code using strings.
- *Python Library Reference* summarizes all the string methods.
- *Python Library Reference* documents the `string` module.
- *The Whole Python FAQ* explains why `join` is a string method instead of a list method.

1.15. Summary

The `odbchelper.py` program and its output should now make perfect sense.

Example 1.38. `odbchelper.py`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
```

```
            }
print buildConnectionString(myParams)
```

Example 1.39. Output of `odbchelper.py`

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

Before diving into the next chapter, make sure you're comfortable doing all of these things:

- Using the Python IDE to test expressions interactively
- Writing Python modules so they can also be run as standalone programs, at least for testing purposes
- Importing modules and calling their functions
- Declaring functions and using `doc strings`, local variables, and proper indentation
- Defining dictionaries, tuples, and lists
- Accessing attributes and methods of any object, including strings, lists, dictionaries, functions, and modules
- Concatenating values through string formatting
- Mapping lists into other lists using list comprehensions
- Splitting strings into lists and joining lists into strings

---

[1] Different programming languages define "object" in different ways. In some, it means that *all* objects *must* have attributes and methods; in others, it means that all objects are subclassable. In Python, the definition is looser; some objects have neither attributes nor methods (more on this later in this chapter), and not all objects are subclassable (more on this in chapter 3). But everything is an object in the sense that it can be assigned to a variable or passed as an argument to a function (more in this in chapter 2).

[2] Actually, it's more complicated than that. Dictionary keys must be immutable. Tuples themselves are immutable, but if you have a tuple of lists, that counts as mutable and isn't safe to use as a dictionary key. Only tuples of strings, numbers, or other dictionary−safe tuples can be used as dictionary keys.

Chapter 2. The Power Of Introspection

2.1. Diving in

This chapter covers one of Python's strengths: introspection. As you know, everything in Python is an object, and introspection is code looking at other modules and functions in memory as objects, getting information about them, and manipulating them. Along the way, we'll define functions with no name, call functions with arguments out of order, and reference functions whose names we don't even know ahead of time.

Here is a complete, working Python program. You should understand a good deal about it just by looking at it. The numbered lines illustrate concepts covered in *Getting To Know Python*. Don't worry if the rest of the code looks intimidating; you'll learn all about it throughout this chapter.

Example 2.1. `apihelper.py`

If you have not already done so, you can download this and other examples used in this book.

```
def help(object, spacing=10, collapse=1): (1) (2) (3)
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])

if __name__ == "__main__":                 (4) (5)
    print help.__doc__
```

(1) This module has one function, `help`. According to its function declaration, it takes three parameters: `object`, `spacing`, and `collapse`. The last two are actually optional parameters, as we'll see shortly.

(2) The `help` function has a multi−line `doc string` that succinctly describes the function's purpose. Note that no return value is mentioned; this function will be used solely for its effects, not its value.

(3) Code within the function is indented.

(4) The `if __name__` trick allows this program do something useful when run by itself, without interfering with its use as a module for other programs. In this case, the program simply prints out the `doc string` of the `help` function.

(5) `if` statements use `==` for comparison, and parentheses are not required.

The `help` function is designed to be used by you, the programmer, while working in the Python IDE. It takes any object that has functions or methods (like a module, which has functions, or a list, which has methods) and prints out the functions and their `doc strings`.

Example 2.2. Sample usage of `apihelper.py`

```
>>> from apihelper import help
>>> li = []
>>> help(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
```

```
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

By default the output is formatted to be easily readable. Multi−line `doc string`s are collapsed into a single long line, but this option can be changed by specifying `0` for the *`collapse`* argument. If the function names are longer than 10 characters, you can specify a larger value for the *`spacing`* argument to make the output easier to read.

Example 2.3. Advanced usage of `apihelper.py`

```
>>> import odbchelper
>>> help(odbchelper)
buildConnectionString Build a connection string from a dictionary Returns string.
>>> help(odbchelper, 30)
buildConnectionString          Build a connection string from a dictionary Returns string.
>>> help(odbchelper, 30, 0)
buildConnectionString          Build a connection string from a dictionary

    Returns string.
```

2.2. Optional and named arguments

Python allows function arguments to have default values; if the function is called without the argument, the argument gets its default value. Futhermore, arguments can be specified in any order by using named arguments. Stored procedures in SQL Server Transact/SQL can do this; if you're a SQL Server scripting guru, you can skim this part.

Example 2.4. `help`, a function with two optional arguments

```
def help(object, spacing=10, collapse=1):
```

`spacing` and `collapse` are optional, because they have default values defined. `object` is required, because it has no default value. If `help` is called with only one argument, `spacing` defaults to `10` and `collapse` defaults to `1`. If `help` is called with two arguments, `collapse` still defaults to `1`.

Say you want to specify a value for `collapse` but want to accept the default value for `spacing`. In most languages, you would be out of luck, because you would have to call the function with three arguments. But in Python, arguments can be specified by name, in any order.

Example 2.5. Valid calls of `help`

```
help(odbchelper)                    (1)
help(odbchelper, 12)                (2)
help(odbchelper, collapse=0)        (3)
help(spacing=15, object=odbchelper) (4)
```

(1) With only one argument, `spacing` gets its default value of `10` and `collapse` gets its default value of `1`.

(2) With two arguments, `collapse` gets its default value of `1`.

(3) Here you are naming the `collapse` argument explicitly and specifying its value. `spacing` still gets its default value of `10`.

(4) Even required arguments (like `object`, which has no default value) can be named, and named arguments can appear in any order.

This looks totally whacked until you realize that arguments are simply a dictionary. The "normal" method of calling functions without argument names is actually just a shorthand where Python matches up the values with the argument names in the order they're specified in the function declaration. And most of the time, you'll call functions the "normal" way, but you always have the additional flexibility if you need it.

> **Note: Calling functions is flexible**
> The only thing you have to do to call a function is specify a value (somehow) for each required argument; the manner and order in which you do that is up to you.

Further reading

- *Python Tutorial* discusses exactly when and how default arguments are evaluated, which matters when the default value is a list or an expression with side effects.

2.3. `type`, `str`, `dir`, and other built−in functions

Python has a small set of extremely useful built−in functions. All other functions are partitioned off into modules. This was actually a conscious design decision, to keep the core language from getting bloated like other scripting languages (cough cough, Visual Basic).

The `type` function returns the datatype of any arbitrary object. The possible types are listed in the `types` module. This is useful for helper functions which can handle several types of data.

Example 2.6. Introducing `type`

```
>>> type(1)           (1)
<type 'int'>
>>> li = []
>>> type(li)          (2)
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)  (3)
<type 'module'>
>>> import types      (4)
>>> type(odbchelper) == types.ModuleType
1
```

(1) `type` takes anything and returns its datatype. And I mean anything. Integers, strings, lists, dictionaries, tuples, functions, classes, modules, even types.

(2) `type` can take a variable and return its datatype.

(3) `type` also works on modules.

(4) You can use the constants in the `types` module to compare types of objects. This is what the `help` function does, as we'll see shortly.

The `str` coerces data into a string. Every datatype can be coerced into a string.

Example 2.7. Introducing `str`

```
>>> str(1)          (1)
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
>>> str(horsemen)   (2)
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbchelper) (3)
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> str(None)       (4)
'None'
```

(1) For simple datatypes like integers, you would expect `str` to work, because almost every language has a function to convert an integer to a string.

(2) However, `str` works on any object of any type. Here it works on a list which we've constructed in bits and pieces.

(3) `str` also works on modules. Note that the string representation of the module includes the pathname of the module on disk, so yours will be different.

(4) A subtle but important behavior of `str` is that it works on `None`, the Python null value. It returns the string `'None'`. We will use this to our advantage in the `help` function, as we'll see shortly.

At the heart of our `help` function is the powerful `dir` function. `dir` returns a list of the attributes and methods of any object: modules, functions, strings, lists, dictionaries... pretty much anything.

Example 2.8. Introducing `dir`

```
>>> li = []
>>> dir(li)           (1)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> d = {}
>>> dir(d)            (2)
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'setdefault', 'update', 'values']
>>> import odbchelper
>>> dir(odbchelper)   (3)
['__builtins__', '__doc__', '__file__', '__name__', 'buildConnectionString']
```

(1) `li` is a list, so `dir(li)` returns a list of all the methods of a list. Note that the returned list contains the names of the methods as strings, not the methods themselves.

(2) `d` is a dictionary, so `dir(d)` returns a list of the names of dictionary methods. At least one of these, `keys`, should look familiar.

(3) This is where it really gets interesting. `odbchelper` is a module, so `dir(odbchelper)` returns a list of all kinds of stuff defined in the module, including built−in attributes, like `__name__` and `__doc__`, and whatever other attributes and methods you define. In this case, `odbchelper` has only one user−defined method, the `buildConnectionString` function we studied in *Getting To Know Python*.

Finally, the `callable` function takes any object and returns `1` if the object can be called, or `0` otherwise. Callable objects include functions, class methods, even classes themselves. (More on classes in chapter 3.)

Example 2.9. Introducing `callable`

```
>>> import string
>>> string.punctuation           (1)
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join                  (2)
<function join at 00C55A7C>
>>> callable(string.punctuation) (3)
```

```
0
>>> callable(string.join)       (4)
1
>>> print string.join.__doc__    (5)
join(list [,sep]) -> string

    Return a string composed of the words in list, with
    intervening occurrences of sep.  The default separator is a
    single space.

    (joinfields and join are synonymous)
```

(1) The functions in the `string` module are deprecated (although lots of people still use the `join` function), but the module contains lots of useful constants like this `string.punctuation`, which contains all the standard punctuation characters.

(2) `string.join` is a function that joins a list of strings.

(3) `string.punctuation` is not callable; it is a string. (A string does have callable methods, but the string itself is not callable.)

(4) `string.join` is callable; it's a function that takes two arguments.

(5) Any callable object may have a `doc string`. Using the `callable` function on each of an object's attributes, we can determine which attributes we care about (methods, functions, classes) and which we want to ignore (constants, *etc.*) without knowing anything about the object ahead of time.

`type`, `str`, `dir`, and all the rest of Python's built−in functions are grouped into a special module called `__builtin__`. (That's two underscores before and after.) If it helps, you can think of Python automatically executing `from __builtin__ import *` on startup, which imports all the "built−in" functions into the namespace so you can use them directly.

The advantage of thinking like this is that you can access all the built−in functions and attributes as a group by getting information about the `__builtin__` module. And guess what, we have a function for that; it's called `help`. Try it yourself and skim through the list now; we'll dive into some of the more important functions later. (Some of the built−in error classes, like `AttributeError`, should already look familiar.)

Example 2.10. Built−in attributes and functions

```
>>> from apihelper import help
>>> import __builtin__
>>> help(__builtin__, 20)
ArithmeticError      Base class for arithmetic errors.
AssertionError       Assertion failed.
AttributeError       Attribute not found.
EOFError             Read beyond end of file.
EnvironmentError     Base class for I/O related errors.
Exception            Common base class for all exceptions.
FloatingPointError   Floating point operation failed.
IOError              I/O operation failed.

[...snip...]
```

> **Note: Python is self−documenting**
> Python comes with excellent reference manuals, which you should peruse thoroughly to learn all the modules Python has to offer. But whereas in most languages you would find yourself referring back to the manuals (or man pages, or, God help you, MSDN) to remind yourself how to use these modules, Python is largely self−documenting.

Further reading

- *Python Library Reference* documents all the built−in functions and all the built−in exceptions.

2.4. Getting object references with `getattr`

You already know that Python functions are objects. What you don't know is that you can get a reference to a function without knowing its name until run−time, using the `getattr` function.

Example 2.11. Introducing `getattr`

```
>>> li = ["Larry", "Curly"]
>>> li.pop                       (1)
<built-in method pop of list object at 010DF884>
>>> getattr(li, "pop")           (2)
<built-in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe") (3)
>>> li
["Larry", "Curly", "Moe"]
>>> getattr({}, "clear")         (4)
<built-in method clear of dictionary object at 00F113D4>
>>> getattr((), "pop")           (5)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

(1) This gets a reference to the `pop` method of the list. Note that this is not calling the `pop` method; that would be `li.pop()`. This is the method itself.

(2) This also returns a reference to the `pop` method, but this time, the method name is specified as a string argument to the `getattr` function. `getattr` is an incredibly useful built−in function which returns any attribute of any object. In this case, the object is a list, and the attribute is the `pop` method.

(3) In case it hasn't sunk in just how incredibly useful this is, try this: the return value of `getattr` *is* the method, which you can then call just as if you had said `li.append("Moe")` directly. But you didn't call the function directly; you specified the function name as a string instead.

(4) `getattr` also works on dictionaries.

(5) In theory, `getattr` would work on tuples, except that tuples have no methods, so `getattr` will raise an exception no matter what attribute name you give.

`getattr` isn't just for built−in datatypes. It also works on modules.

Example 2.12. `getattr` in `apihelper.py`

```
>>> import odbchelper
>>> odbchelper.buildConnectionString             (1)
<function buildConnectionString at 00D18DD4>
>>> getattr(odbchelper, "buildConnectionString") (2)
<function buildConnectionString at 00D18DD4>
>>> object = odbchelper
>>> method = "buildConnectionString"
>>> getattr(object, method)                      (3)
<function buildConnectionString at 00D18DD4>
>>> type(getattr(object, method))                (4)
<type 'function'>
>>> import types
```

```
>>> type(getattr(object, method)) == types.FunctionType
1
>>> callable(getattr(object, method))           (5)
1
```

(1) This returns a reference to the `buildConnectionString` function in the `odbchelper` module, which we studied in *Getting To Know Python*. (The hex address you see is specific to my machine; your output will be different.)

(2) Using `getattr`, we can get the same reference to the same function. In general, `getattr(`*`object`*`, "`*`attribute`*`")` is equivalent to *`object.attribute`*. If *`object`* is a module, then *`attribute`* can be anything defined in the module: a function, class, or global variable.

(3) And this is what we actually use in the `help` function. `object` is passed into the function as an argument; `method` is a string which is the name of a method or function.

(4) In this case, `method` is the name of a function, which we can prove by getting its `type`.

(5) Since `method` is a function, it is callable.

2.5. Filtering lists

As you know, Python has powerful capabilities for mapping lists into other lists, via list comprehensions. This can be combined with a filtering mechanism, where some elements in the list are mapped while others are skipped entirely.

Example 2.13. List filtering syntax

```
[mapping-expression for element in source-list if filter-expression]
```

This is an extension of the list comprehensions that you know and love. The first two thirds are the same; the last part, starting with the `if`, is the filter expression. A filter expression can be any expression that evaluates true or false (which in Python can be almost anything). Any element for which the filter expression evaluates true will be included in the mapping. All other elements are ignored, so they are never put through the mapping expression and are not included in the output list.

Example 2.14. Introducing list filtering

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]       (1)
['mpilgrim', 'foo']
>>> [elem for elem in li if elem != "b"]         (2)
['a', 'mpilgrim', 'foo', 'c', 'd', 'd']
>>> [elem for elem in li if li.count(elem) == 1] (3)
['a', 'mpilgrim', 'foo', 'c']
```

(1) The mapping expression here is simple (it just returns the value of each element), so concentrate on the filter expression. As Python loops through the list, it runs each element through the filter expression; if the filter expression is true, the element is mapped and the result of the mapping expression is included in the returned list. Here you are filtering out all the one−character strings, so you're left with a list of all the longer strings.

(2) Here you are filtering out a specific value, `b`. Note that this filters all occurrences of `b`, since each time it comes up, the filter expression will be false.

(3) `count` is a list method that returns the number of times a value occurs in a list. You might think that this filter would eliminate duplicates from a list, returning a list containing only one copy of each value in the original list. But it doesn't, because values that appear twice in the original list (in this case, `b` and

d) are excluded completely. There are ways of eliminating duplicates from a list, but filtering is not the solution.

Example 2.15. Filtering a list in `apihelper.py`

```
methodList = [method for method in dir(object) if callable(getattr(object, method))]
```

This looks complicated, and it is complicated, but the basic structure is the same. The whole filter expression returns a list, which is assigned to the `methodList` variable. The first half of the expression is the list mapping part. The mapping expression is an identity expression; it returns the value of each element. `dir(object)` returns a list of `object`'s attributes and methods; that's the list you're mapping. So the only new part is the filter expression after the `if`.

The filter expression looks scary, but it's not. You already know about `callable`, `getattr`, and `in`. As you saw in the previous section, the expression `getattr(object, method)` returns a function object if `object` is a module and `method` is the name of a function in that module.

So this expression takes an object, named `object`, getting a list of the names of its attributes, methods, functions, and a few other things, and then filtering that list to weed out all the stuff that we don't care about. We do the weeding out by taking the name of each attribute/method/function and getting a reference to the real thing, via the `getattr` function. Then we check to see if that object is callable, which will be any methods and functions, both built−in (like the `pop` method of a list) and user−defined (like the `buildConnectionString` function of the `odbchelper` module). We don't care about other attributes, like the `__name__` attribute that's built in to every module.

Further reading

- *Python Tutorial* discusses another way to filter lists using the built−in `filter` function.

2.6. The peculiar nature of and and or

In Python, `and` and `or` perform boolean logic as you would expect, but they do not return boolean values; they return one of the actual values they are comparing.

Example 2.16. Introducing `and`

```
>>> 'a' and 'b'         (1)
'b'
>>> '' and 'b'          (2)
''
>>> 'a' and 'b' and 'c' (3)
'c'
```

(1) When using `and`, values are evaluated in a boolean context from left to right. `0`, `''`, `[]`, `()`, `{}`, and `None` are false in a boolean context; everything else is true.[3] If all values are true in a boolean context, `and` returns the last value. In this case, `and` evaluates `'a'`, which is true, then `'b'`, which is true, and returns `'b'`.

(2) If any value is false in a boolean context, `and` returns the first false value. In this case, `''` is the first false value.

(3) All values are true, so `and` returns the last value, `'c'`.

Example 2.17. Introducing `or`

```
>>> 'a' or 'b'          (1)
'a'
>>> '' or 'b'           (2)
'b'
>>> '' or [] or {}      (3)
{}
>>> def sidefx():
...     print "in sidefx()"
...     return 1
>>> 'a' or sidefx()     (4)
'a'
```

(1) When using `or`, values are evaluated in a boolean context from left to right, just like `and`. If any value is true, `or` returns that value immediately. In this case, `'a'` is the first true value.

(2) `or` evaluates `''`, which is false, then `'b'`, which is true, and returns `'b'`.

(3) If all values are false, `or` returns the last value. `or` evaluates `''`, which is false, then `[]`, which is false, then `{}`, which is false, and returns `{}`.

(4) Note that `or` only evaluates values until it finds one that is true in a boolean context, and then it ignores the rest. This distinction is important if some values can have side effects. Here, the function `sidefx` is never called, because `or` evaluates `'a'`, which is true, and returns `'a'` immediately.

If you're a C hacker, you are certainly familiar with the `bool ? a : b` expression, which evaluates to `a` if `bool` is true, and `b` otherwise. Because of the way `and` and `or` work in Python, you can accomplish the same thing.

Example 2.18. Introducing the `and−or` trick

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b (1)
'first'
>>> 0 and a or b (2)
'second'
```

(1) This syntax looks similar to the `bool ? a : b` expression in C. The entire expression is evaluated from left to right, so the `and` is evaluated first. `1 and 'first'` evalutes to `'first'`, then `'first' or 'second'` evalutes to `'first'`.

(2) `0 and 'first'` evalutes to `0`, then `0 or 'second'` evaluates to `'second'`.

However, since this Python expression is simply boolean logic, and not a special construct of the language, there is one very, very, very important difference between this `and−or` trick in Python and the `bool ? a : b` syntax in C. If the value of `a` is false, the expression will not work as you would expect it to. (Can you tell I was bitten by this? More than once?)

Example 2.19. When the `and−or` trick fails

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b (1)
'second'
```

(1) Since `a` is an empty string, which Python considers false in a boolean context, `1 and ''` evalutes to `''`, then `'' or 'second'` evalutes to `'second'`. Oops! That's not what we wanted.

Important: Using and−or effectively

The `and−or` trick, `bool and a or b`, will not work like the C expression `bool ? a : b` when `a` is false in a boolean context.

The real trick behind the `and−or` trick, then, is to make sure that the value of `a` is never false. One common way of doing this is to turn `a` into `[a]` and `b` into `[b]`, then taking the first element of the returned list, which will be either `a` or `b`.

Example 2.20. Using the `and−or` trick safely

```
>>> a = ""
>>> b = "second"
>>> (1 and [a] or [b])[0] (1)
''
```

(1) Since `[a]` is a non−empty list, it is never false. Even if `a` is `0` or `''` or some other false value, the list `[a]` is true because it has one element.

By now, this trick may seem like more trouble than it's worth. You could, after all, accomplish the same thing with an `if` statement, so why go through all this fuss? Well, in many cases, you are choosing between two constant values, so you can use the simpler syntax and not worry, because you know that the `a` value will always be true. And even if you have to use the more complicated safe form, there are good reasons to do so; there are some cases in Python where `if` statements are not allowed, like `lambda` functions.

Further reading

- Python Cookbook discusses alternatives to the `and−or` trick.

2.7. Using `lambda` functions

Python supports an interesting syntax that lets you define one−line mini−functions on the fly. Borrowed from Lisp, these so−called `lambda` functions can be used anywhere a function is required.

Example 2.21. Introducing `lambda` functions

```
>>> def f(x):
...     return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2  (1)
>>> g(3)
6
>>> (lambda x: x*2)(3) (2)
6
```

(1) This is a `lambda` function that accomplishes the same thing as the normal function above it. Note the abbreviated syntax here: there are no parentheses around the argument list, and the `return` keyword is missing (it is implied, since the entire function can only be one expression). Also, the function has no name, but it can be called through the variable it is assigned to.

(2) You can use a `lambda` function without even assigning it to a variable. Not the most useful thing in the world, but it just goes to show that a lambda is just an in−line function.

To generalize, a `lambda` function is a function that takes any number of arguments (including optional arguments) and returns the value of a single expression. `lambda` functions can not contain commands, and they can not contain more than one expression. Don't try to squeeze too much into a `lambda` function; if you need something more complex, define a normal function instead and make it as long as you want.

> **Note: lambda is optional**
> `lambda` functions are a matter of style. Using them is never required; anywhere you could use them, you could define a separate normal function and use that instead. I use them in places where I want to encapsulate specific, non−reusable code without littering my code with a lot of little one−line functions.

Example 2.22. `lambda` functions in `apihelper.py`

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

Several things to note here in passing. First, we're using the simple form of the `and−or` trick, which is OK, because a `lambda` function is always true in a boolean context. (That doesn't mean that a `lambda` function can't return a false value. The function is always true; its return value could be anything.)

Second, we're using the `split` function with no arguments. You've already seen it used with 1 or 2 arguments, but with no arguments it splits on whitespace.

Example 2.23. `split` with no arguments

```
>>> s = "this   is\na\ttest"  (1)
>>> print s
this   is
a       test
>>> print s.split()           (2)
['this', 'is', 'a', 'test']
>>> print " ".join(s.split()) (3)
'this is a test'
```

(1) This is a multiline string, defined by escape characters instead of triple quotes. `\n` is a carriage return; `\t` is a tab character.

(2) `split` with no arguments splits on whitespace. So three spaces, a carriage return, and a tab character are all the same.

(3) You can normalize whitespace by `split`ting a string and then re`join`ing it with a single space as a delimiter. This is what the `help` function does to collapse multi−line `doc string`s into a single line.

So what is the `help` function actually doing with these `lambda` functions, `split`s, and `and−or` tricks?

Example 2.24. Assigning a function to a variable

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

`processFunc` is now a function, but which function it is depends on the value of the `collapse` variable. If `collapse` is true, `processFunc(`*`string`*`)` will collapse whitespace; otherwise, `processFunc(`*`string`*`)` will return its argument unchanged.

To do this in a less robust language, like Visual Basic, you would probably create a function that took a string and a *`collapse`* argument and used an `if` statement to decide whether to collapse the whitespace or not, then returned the appropriate value. This would be inefficient, because the function would have to handle every possible case; every time you called it, it would have to decide whether to collapse whitespace before it could give you what you wanted.

In Python, you can take that decision logic out of the function and define a `lambda` function that is custom−tailored to give you exactly (and only) what you want. This is more efficient, more elegant, and less prone to those nasty oh−I−thought−those−arguments−were−reversed kinds of errors.

Further reading

- Python Knowledge Base discusses using `lambda` to call functions indirectly.
- *Python Tutorial* shows how to access outside variables from inside a `lambda` function. (PEP 227 explains how this will change in future versions of Python.)
- *The Whole Python FAQ* has examples of obfuscated one−liners using `lambda`.

2.8. Putting it all together

The last line of code, the only one we haven't deconstructed yet, is the one that does all the work. But by now the work is easy, because everything we need is already set up just the way we need it. All the dominoes are in place; it's time to knock them down.

Example 2.25. The meat of `apihelper.py`

```
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])
```

Note that this is one command, split over multiple lines, but it doesn't use the line continuation character ("\"). Remember when I said that some expressions can be split into multiple lines without using a backslash? A list comprehension is one of those expressions, since the entire expression is contained in square brackets.

Now, let's take it from the end and work backwards. The

```
for method in methodList
```

shows us that this is a list comprehension. As you know, `methodList` is a list of all the methods we care about in `object`. So we're looping through that list with `method`.

Example 2.26. Getting a `doc string` dynamically

```
>>> import odbchelper
>>> object = odbchelper                   (1)
>>> method = 'buildConnectionString'      (2)
>>> getattr(object, method)               (3)
<function buildConnectionString at 010D6D74>
>>> print getattr(object, method).__doc__ (4)
Build a connection string from a dictionary of parameters.

    Returns string.
```

(1) In the `help` function, `object` is the object we're getting help on, passed in as an argument.

(2) As we're looping through `methodList`, `method` is the name of the current method.

(3) Using the `getattr` function, we're getting a reference to the *`method`* function in the *`object`* module.

(4) Now, printing the actual `doc string` of the method is easy.

The next piece of the puzzle is the use of `str` around the `doc string`. As you may recall, `str` is a built−in function that coerces data into a string. But a `doc string` is always a string, so why bother with the `str` function? The answer is that not every function has a `doc string`, and if it doesn't, its `__doc__` attribute is `None`.

Example 2.27. Why use `str` on a `doc string`?

```
>>> >>> def foo(): print 2
>>> >>> foo()
2
>>> >>> foo.__doc__     (1)
>>> foo.__doc__ == None (2)
1
>>> str(foo.__doc__)    (3)
'None'
```

(1) We can easily define a function that has no `doc string`, so its `__doc__` attribute is `None`. Confusingly, if you evaluate the `__doc__` attribute directly, the Python IDE prints nothing at all, which makes sense if you think about it, but is still unhelpful.

(2) You can verify that the value of the `__doc__` attribute is actually `None` by comparing it directly.

(3) Using the `str` function takes the null value and returns a string representation of it, `'None'`.

Note: Python vs. SQL: comparing null values
In SQL, you must use `IS NULL` instead of `= NULL` to compare a null value. In Python, you can use either `== None` or `is None`, but `is None` is faster.

Now that we are guaranteed to have a string, we can pass the string to `processFunc`, which we have already defined as a function that either does or doesn't collapse whitespace. Now you see why it was important to use `str` to convert a `None` value into a string representation. `processFunc` is assuming a string argument and calling its `split` method, which would crash if we passed it `None` because `None` doesn't have a `split` method.

Stepping back even further, we see that we're using string formatting again to concatenate the return value of `processFunc` with the return value of `method`'s `ljust` method. This is a new string method that we haven't seen before.

Example 2.28. Introducing the `ljust` method

```
>>> s = 'buildConnectionString'
>>> s.ljust(30) (1)
'buildConnectionString         '
>>> s.ljust(20) (2)
'buildConnectionString'
```

(1) `ljust` pads the string with spaces to the given length. This is what the `help` function uses to make two columns of output and line up all the `doc string`s in the second column.

(2) If the given length is smaller than the length of the string, `ljust` will simply return the string unchanged. It never truncates the string.

We're almost done. Given the padded method name from the `ljust` method and the (possibly collapsed) `doc string` from the call to `processFunc`, we concatenate the two and get a single string. Since we're mapping `methodList`, we end up with a list of strings. Using the `join` method of the string `"\n"`, we join this list into a single string, with each element of the list on a separate line, and print the result.

Example 2.29. Printing a list

```
>>> li = ['a', 'b', 'c']
>>> print "\n".join(li) (1)
a
b
c
```

(1) This is also a useful debugging trick when you're working with lists. And in Python, you're always working with lists.

That's the last piece of the puzzle. This code should now make perfect sense.

Example 2.30. The meat of `apihelper.py`, revisited

```
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])
```

2.9. Summary

The `apihelper.py` program and its output should now make perfect sense.

Example 2.31. `apihelper.py`

```
def help(object, spacing=10, collapse=1):
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])

if __name__ == "__main__":
    print help.__doc__
```

Example 2.32. Output of `apihelper.py`

```
>>> from apihelper import help
>>> li = []
>>> help(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

Before diving into the next chapter, make sure you're comfortable doing all of these things:

- Defining and calling functions with optional and named arguments
- Using `str` to coerce any arbitrary value into a string representation
- Using `getattr` to get references to functions and other attributes dynamically
- Extending the list comprehension syntax to do list filtering
- Recognizing the `and−or` trick and using it safely
- Defining `lambda` functions
- Assigning functions to variables and calling the function by referencing the variable. I can't emphasize this enough: this mode of thought is vital to advancing your understanding of Python. You'll see more complex applications of this concept throughout this book.

---

[3] Well, almost everything. By default, instances of classes are true in a boolean context, but you can define special methods in your class to make an instance evaluate to false. You'll learn all about classes and special methods in chapter 3.

Chapter 3. An Object−Oriented Framework

3.1. Diving in

This chapter, and pretty much every chapter after this, deals with object−oriented Python programming. Remember when I said you should know an object−oriented language to read this book? Well, I wasn't kidding.

Here is a complete, working Python program. Read the `doc string`s of the module, the classes, and the functions to get an overview of what this program does and how it works. As usual, don't worry about the stuff you don't understand; that's what the rest of the chapter is for.

Example 3.1. `fileinfo.py`

If you have not already done so, you can download this and other examples used in this book.

```
"""Framework for getting filetype-specific metadata.

Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key-value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
        try:
```

```
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]): (1)
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

(1) This program's output depends on the files on your hard drive. To get meaningful output, you'll have to change the directory path to point to a directory of MP3 files on your own machine.

Example 3.2. Output of `fileinfo.py`

This was the output I got on my machine. Your output will be different, unless, by some startling coincidence, you share my exact taste in music.

```
album=
artist=Ghost in the Machine
title=A Time Long Forgotten (Concept
genre=31
name=/music/_singles/a_time_long_forgotten_con.mp3
year=1999
comment=http://mp3.com/ghostmachine

album=Rave Mix
artist=***DJ MARY-JANE***
title=HELLRAISER****Trance from Hell
genre=31
name=/music/_singles/hellraiser.mp3
year=2000
comment=http://mp3.com/DJMARYJANE

album=Rave Mix
artist=***DJ MARY-JANE***
title=KAIRO****THE BEST GOA
genre=31
```

```
name=/music/_singles/kairo.mp3
year=2000
comment=http://mp3.com/DJMARYJANE

album=Journeys
artist=Masters of Balance
title=Long Way Home
genre=31
name=/music/_singles/long_way_home1.mp3
year=2000
comment=http://mp3.com/MastersofBalan

album=
artist=The Cynic Project
title=Sidewinder
genre=18
name=/music/_singles/sidewinder.mp3
year=2000
comment=http://mp3.com/cynicproject

album=Digitosis@128k
artist=VXpanded
title=Spinning
genre=255
name=/music/_singles/spinning.mp3
year=2000
comment=http://mp3.com/artists/95/vxp
```

3.2. Importing modules using `from` *`module`* `import`

Python has two ways of importing modules. Both are useful, and you should know when to use each. One way, `import` *`module`*, you've already seen in chapter 1. The other way accomplishes the same thing but works in subtlely and importantly different ways.

Example 3.3. Basic `from` *`module`* `import` syntax

```
from UserDict import UserDict
```

This is similar to the `import` *`module`* syntax that you know and love, but with an important difference: the attributes and methods of the imported module `types` are imported directly into the local namespace, so they are available directly, without qualification by module name. You can import individual items or use `from` *`module`* `import *` to import everything.

> **Note: Python vs. Perl: from module import**
> `from` *`module`* `import *` in Python is like `use` *`module`* in Perl; `import` *`module`* in Python is like `require` *`module`* in Perl.

> **Note: Python vs. Java: from module import**
> `from` *`module`* `import *` in Python is like `import` *`module`*`.*` in Java; `import` *`module`* in Python is like `import` *`module`* in Java.

Example 3.4. `import` *`module`* vs. `from` *`module`* `import`

```
>>> import types
>>> types.FunctionType             (1)
```

```
<type 'function'>
>>> FunctionType                  (2)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'FunctionType'
>>> from types import FunctionType (3)
>>> FunctionType                  (4)
<type 'function'>
```

(1) The `types` module contains no methods, just attributes for each Python object type. Note that the attribute, `FunctionType`, must be qualified by the module name, `types`.

(2) `FunctionType` by itself has not been defined in this namespace; it only exists in the context of `types`.

(3) This syntax imports the attribute `FunctionType` from the `types` module directly into the local namespace.

(4) Now `FunctionType` can be accessed directly, without reference to `types`.

When should you use `from` *`module`* `import`?

- If you will be accessing attributes and methods often and don't want to type the module name over and over, use `from` *`module`* `import`.
- If you want to selectively import some attributes and methods but not others, use `from` *`module`* `import`.
- If the module contains attributes or functions with the same name as ones in your module, you must use `import` *`module`* to avoid name conflicts.

Other than that, it's just a matter of style, and you will see Python code written both ways.

Further reading

- eff−bot has more to say on `import` *`module`* *vs.* `from` *`module`* `import`.
- *Python Tutorial* discusses advanced import techniques, including `from` *`module`* `import *`.

3.3. Defining classes

Python is fully object−oriented: you can define your own classes, inherit from your own or built−in classes, and instantiate the classes you've defined.

Defining a class in Python is simple; like functions, there is no separate interface definition. Just define the class and start coding. A Python class starts with the reserved word `class`, followed by the class name. Technically, that's all that's required, since a class doesn't have to inherit from any other class.

Example 3.5. The simplest Python class

```
class foo: (1)
    pass   (2) (3)
```

(1) The name of this class is `foo`, and it doesn't inherit from any other class.

(2) This class doesn't define any methods or attributes, but syntactically, there needs to be something in the definition, so we use `pass`. This is a Python reserved word that just means "move along, nothing to see here". It's a statement that does nothing, and it's a good placeholder when you're stubbing out functions or classes.

(3) You probably guessed this, but everything in a class is indented, just like the code within a function, `if` statement, `for` loop, and so forth. The first thing not indented is not in the class.

Note: Python vs. Java: pass
The `pass` statement in Python is like an empty set of braces (`{}`) in Java or C.

Of course, realistically, most classes will be inherited from other classes, and they will define their own class methods and attributes. But as you've just seen, there is nothing that a class absolutely must have, other than a name. In particular, C++ programmers may find it odd that Python classes don't have explicit constructors and destructors. Python classes do have something similar to a constructor: the `__init__` method.

Example 3.6. Defining the `FileInfo` class

```
from UserDict import UserDict

class FileInfo(UserDict): (1)
```

(1) In Python, the ancestor of a class is simply listed in parentheses immediately after the class name. So the `FileInfo` class is inherited from the `UserDict` class (which was imported from the `UserDict` module). `UserDict` is a class that acts like a dictionary, allowing you to essentially subclass the dictionary datatype and add your own behavior. (There are similar classes `UserList` and `UserString` which allow you to subclass lists and strings.) There is a bit of black magic behind this, which we will demystify later in this chapter when we explore the `UserDict` class in more depth.

Note: Python vs. Java: ancestors
In Python, the ancestor of a class is simply listed in parentheses immediately after the class name. There is no special keyword like `extends` in Java.

Note: Multiple inheritance
Although I won't discuss it in depth in this book, Python supports multiple inheritance. In the parentheses following the class name, you can list as many ancestor classes as you like, separated by commas.

Example 3.7. Initializing the `FileInfo` class

```
class FileInfo(UserDict):
    "store file metadata"              (1)
    def __init__(self, filename=None): (2) (3) (4)
```

(1) Classes can (and should) have `doc strings` too, just like modules and functions.

(2) `__init__` is called immediately after an instance of the class is created. It would be tempting but incorrect to call this the constructor of the class. Tempting, because it looks like a constructor (by convention, `__init__` is the first method defined for the class), acts like one (it's the first piece of code executed in a newly created instance of the class), and even sounds like one ("init" certainly suggests a constructor−ish nature). Incorrect, because the object has already been constructed by the time `__init__` is called, and you already have a valid reference to the new instance of the class. But `__init__` is the closest thing you're going to get in Python to a constructor, and it fills much the same role.

(3) The first argument of every class method, including `__init__`, is always a reference to the current instance of the class. By convention, this argument is always named `self`. In the `__init__` method, `self` refers to the newly created object; in other class methods, it refers to the instance whose method was called. Although you need to specify `self` explicitly when defining the method, you do *not* specify it when calling the method; Python will add it for you automatically.

(4) `__init__` methods can take any number of arguments, and just like functions, the arguments can be defined with default values, making them optional to the caller. In this case, `filename` has a default value of `None`,

which is the Python null value.

Note: Python vs. Java: self
By convention, the first argument of any class method (the reference to the current instance) is called `self`. This argument fills the role of the reserved word `this` in C++ or Java, but `self` is not a reserved word in Python, merely a naming convention. Nonetheless, please don't call it anything but `self`; this is a very strong convention.

Example 3.8. Coding the `FileInfo` class

```
class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)        (1)
        self["name"] = filename        (2)
                                       (3)
```

(1) Some pseudo−object−oriented languages like Powerbuilder have a concept of "extending" constructors and other events, where the ancestor's method is called automatically before the descendant's method is executed. Python does not do this; you must always explicitly call the appropriate method in the ancestor class.

(2) I told you that this class acts like a dictionary, and here is the first sign of it. We're assigning the argument `filename` as the value of this object's `name` key.

(3) Note that the `__init__` method never returns a value.

Note: When to use self
When defining your class methods, you *must* explicitly list `self` as the first argument for each method, including `__init__`. When you call a method of an ancestor class from within your class, you *must* include the `self` argument. But when you call your class method from outside, you do not specify anything for the `self` argument; you skip it entirely, and Python automatically adds the instance reference for you. I am aware that this is confusing at first; it's not really inconsistent, but it may appear inconsistent because it relies on a distinction (between bound and unbound methods) that you don't know about yet.

Whew. I realize that's a lot to absorb, but you'll get the hang of it. All Python classes work the same way, so once you learn one, you've learned them all. If you forget everything else, remember this one thing, because I promise it will trip you up:

Note: __init__ methods
`__init__` methods are optional, but when you define one, you must remember to explicitly call the ancestor's `__init__` method. This is more generally true: whenever a descendant wants to extend the behavior of the ancestor, the descendant method must explicitly call the ancestor method at the proper time, with the proper arguments.

Further reading

- *Learning to Program* has a gentler introduction to classes.
- *How to Think Like a Computer Scientist* shows how to use classes to model compound datatypes.
- *Python Tutorial* has an in−depth look at classes, namespaces, and inheritance.
- Python Knowledge Base answers common questions about classes.

3.4. Instantiating classes

Instantiating classes in Python is straightforward. To instantiate a class, simply call the class as if it were a function, passing the arguments that the `__init__` method defines. The return value will be the newly created object.

Example 3.9. Creating a `FileInfo` instance

```
>>> import fileinfo
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3") (1)
>>> f.__class__                                        (2)
<class fileinfo.FileInfo at 010EC204>
>>> f.__doc__                                          (3)
'base class for file info'
>>> f                                                  (4)
{'name': '/music/_singles/kairo.mp3'}
```

(1) We are creating an instance of the `FileInfo` class (defined in the `fileinfo` module) and assigning the newly created instance to the variable `f`. We are passing one parameter, `/music/_singles/kairo.mp3`, which will end up as the `filename` argument in `FileInfo`'s `__init__` method.

(2) Every class instance has a built−in attribute, `__class__`, which is the object's class. (Note that the representation of this includes the physical address of the instance on my machine; your representation will be different.) Java programmers may be familiar with the `Class` class, which contains methods like `getName` and `getSuperclass` to get metadata information about an object. In Python, this kind of metadata is available directly on the object itself through attributes like `__class__`, `__name__`, and `__bases__`.

(3) You can access the instance's `doc string` just like a function or a module. All instances of a class share the same `doc string`.

(4) Remember when the `__init__` method assigned its `filename` argument to `self["name"]`? Well, here's the result. The arguments we pass when we create the class instance get sent right along to the `__init__` method (along with the object reference, `self`, which Python adds for free).

Note: Python vs. Java: instantiating classes
In Python, simply call a class as if it were a function to create a new instance of the class. There is no explicit `new` operator like C++ or Java.

If creating new instances is easy, destroying them is even easier. In general, there is no need to explicitly free instances, because they are freed automatically when the variables assigned to them go out of scope. Memory leaks are rare in Python.

Example 3.10. Trying to implement a memory leak

```
>>> def leakmem():
...     f = fileinfo.FileInfo('/music/_singles/kairo.mp3') (1)
...
>>> for i in range(100):
...     leakmem()                                          (2)
```

(1) Every time the `leakmem` function is called, we are creating an instance of `FileInfo` and assigning it to the variable `f`, which is a local variable within the function. Then the function ends without ever freeing `f`, so you would expect a memory leak, but you would be wrong. When the function ends, the local variable `f` goes out of scope. At this point, there are no longer any references to the newly created instance of `FileInfo` (since we never assigned it to anything other than `f`), so Python destroys the instance for us.

(2) No matter how many times we call the `leakmem` function, it will never leak memory, because every time, Python will destroy the newly created `FileInfo` class before returning from `leakmem`.

The technical term for this form of garbage collection is "reference counting". Python keeps a list of references to every instance created. In the above example, there was only one reference to the `FileInfo` instance: the local variable `f`. When the function ends, the variable `f` goes out of scope, so the reference count drops to `0`, and Python destroys the instance automatically.

In previous versions of Python, there were situations where reference counting failed, and Python couldn't clean up after you. If you created two instances that referenced each other (for instance, a doubly−linked list, where each node has a pointer to the previous and next node in the list), neither instance would ever be destroyed automatically because Python (correctly) believed that there is always a reference to each instance. Python 2.0 has an additional form of garbage collection called "mark−and−sweep" which is smart enough to notice this virtual gridlock and clean up circular references correctly.

As a former philosophy major, it disturbs me to think that things disappear when no one is looking at them, but that's exactly what happens in Python. In general, you can simply forget about memory management and let Python clean up after you.

Further reading

- *Python Library Reference* summarizes built−in attributes like `__class__`.
- *Python Library Reference* documents the `gc` module, which gives you low−level control over Python's garbage collection.

3.5. `UserDict`: a wrapper class

As you've seen, `FileInfo` is a class that acts like a dictionary. To explore this further, let's look at the `UserDict` class in the `UserDict` module, which is the ancestor of our `FileInfo` class. This is nothing special; the class is written in Python and stored in a `.py` file, just like our code. In particular, it's stored in the `lib` directory in your Python installation.

> **Tip: Open modules quickly**
> In the Python IDE on Windows, you can quickly open any module in your library path with File−>Locate... (**Ctrl−L**).

Historical note. In versions of Python prior to 2.2, you could not directly subclass built−in datatypes like strings, lists, and dictionaries. To compensate for this, Python comes with wrapper classes that mimic the behavior of these built−in datatypes: `UserString`, `UserList`, and `UserDict`. Using a combination of normal and special methods, the `UserDict` class does an excellent imitation of a dictionary, but it's just a class like any other, so you can subclass it to provide custom dictionary−like classes like `FileInfo`. In Python 2.2 and later, you could rewrite this chapter's example so that `FileInfo` inherited directly from `dict` instead of `UserDict`. However, you should still read about how `UserDict` works, in case you need to implement this kind of wrapper object yourself, or in case you need to support versions of Python prior to 2.2.

Example 3.11. Defining the `UserDict` class

```
class UserDict:                                (1)
    def __init__(self, dict=None):             (2)
        self.data = {}                         (3)
        if dict is not None: self.update(dict) (4) (5)
```

(1) Note that `UserDict` is a base class, not inherited from any other class.

(2) This is the `__init__` method that we overrode in the `FileInfo` class. Note that the argument list in this ancestor class is different than the descendant. That's okay; each subclass can have its own set of arguments, as long as it calls the ancestor with the correct arguments. Here the ancestor class has a way to define initial values (by passing a dictionary in the `dict` argument) which our `FileInfo` does not take advantage of.

(3) Python supports data attributes (called "instance variables" in Java and Powerbuilder, "member variables" in C++), which is data held by a specific instance of a class. In this case, each instance of `UserDict` will have a data attribute `data`. To reference this attribute from code outside the class, you would qualify it with the instance name, *`instance`*`.data`, in the same way that you qualify a function with its module name. To reference a data attribute from within the class, we use `self` as the qualifier. By convention, all data attributes are initialized to reasonable values in the `__init__` method. However, this is not required, since data attributes, like local variables, spring into existence when they are first assigned a value.

(4) The `update` method is a dictionary duplicator: it copies all the keys and values from one dictionary to another. This does *not* clear the target dictionary first; if the target dictionary already has some keys, the ones from the source dictionary will be overwritten, but others will be left untouched. Think of `update` has a merge function, not a copy function.

(5) Also, this is a syntax you may not have seen before (I haven't used it in the examples in this book). This is an `if` statement, but instead of having an indented block starting on the next line, there is just a single statement on the same line, after the colon. This is perfectly legal syntax, and is just a shortcut when you have only one statement in a block. (It's like specifying a single statement without braces in C++.) You can use this syntax, or you can have indented code on subsequent lines, but you can't do both for the same block.

Note: Python vs. Java: function overloading
Java and Powerbuilder support function overloading by argument list, *i.e.* one class can have multiple methods with the same name but a different number of arguments, or arguments of different types. Other languages (most notably PL/SQL) even support function overloading by argument name; *i.e.* one class can have multiple methods with the same name and the same number of arguments of the same type but different argument names. Python supports neither of these; it has no form of function overloading whatsoever. Methods are defined solely by their name, and there can be only one method per class with a given name. So if a descendant class has an `__init__` method, it *always* overrides the ancestor `__init__` method, even if the descendant defines it with a different argument list. And the same rule applies to any other method.

Note: Guido on derived classes
Guido, the original author of Python, explains method overriding this way: "Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class, may in fact end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)" If that doesn't make sense to you (it confuses the hell out of me), feel free to ignore it. I just thought I'd pass it along.

Note: Always initialize data attributes
Always assign an initial value to all of an instance's data attributes in the `__init__` method. It will save you hours of debugging later, tracking down `AttributeError` exceptions because you're referencing uninitialized (and therefore non−existent) attributes.

Example 3.12. `UserDict` normal methods

```
    def clear(self): self.data.clear()          (1)
    def copy(self):                             (2)
        if self.__class__ is UserDict:          (3)
            return UserDict(self.data)
```

```
        import copy                            (4)
        return copy.copy(self)
    def keys(self): return self.data.keys()     (5)
    def items(self): return self.data.items()
    def values(self): return self.data.values()
```

(1) `clear` is a normal class method; it is publicly available to be called by anyone at any time. Note that `clear`, like all class methods, has `self` as its first argument. (Remember, you don't include `self` when you call the method; it's something that Python adds for you.) Also note the basic technique of this wrapper class: store a real dictionary (`data`) as a data attribute, define all the methods that a real dictionary has, and have each class method redirect to the corresponding method on the real dictionary. (In case you'd forgotten, a dictionary's `clear` method deletes all of its keys and their associated values.)

(2) The `copy` method of a real dictionary returns a new dictionary that is an exact duplicate of the original (all the same key−value pairs). But `UserDict` can't simply redirect to `self.data.copy`, because that method returns a real dictionary, and what we want is to return a new instance that is the same class as `self`.

(3) We use the `__class__` attribute to see if `self` is a `UserDict`; if so, we're golden, because we know how to copy a `UserDict`: just create a new `UserDict` and give it the real dictionary that we've squirreled away in `self.data`.

(4) If `self.__class__` is not `UserDict`, then `self` must be some subclass of `UserDict` (like maybe `FileInfo`), in which case life gets trickier. `UserDict` doesn't know how to make an exact copy of one of its descendants; there could, for instance, be other data attributes defined in the subclass, so we would have to iterate through them and make sure to copy all of them. Luckily, Python comes with a module to do exactly this, and it's called `copy`. I won't go into the details here (though it's a wicked cool module, if you're ever inclined to dive into it on your own). Suffice to say that `copy` can copy arbitrary Python objects, and that's how we're using it here.

(5) The rest of the methods are straightforward, redirecting the calls to the built−in methods on `self.data`.

Further reading

- *Python Library Reference* documents the `UserDict` module and the `copy` module.

3.6. Special class methods

In addition to normal class methods, there are a number of special methods which Python classes can define. Instead of being called directly by your code (like normal methods), special methods are called for you by Python in particular circumstances or when specific syntax is used.

As you saw in the previous section, normal methods went a long way towards wrapping a dictionary in a class. But normal methods alone are not enough, because there are lots of things you can do with dictionaries besides call methods on them. For starters, you can get and set items with a syntax that doesn't include explicitly invoking methods. This is where special class methods come in: they provide a way to map non−method−calling syntax into method calls.

Example 3.13. The `__getitem__` special method

```
    def __getitem__(self, key): return self.data[key]
```

```
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3")
>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__getitem__("name") (1)
'/music/_singles/kairo.mp3'
>>> f["name"]             (2)
```

```
'/music/_singles/kairo.mp3'
```

(1) The `__getitem__` special method looks simple enough. Like the normal methods `clear`, `keys`, and `values`, it just redirects to the dictionary to return its value. But how does it get called? Well, you can call `__getitem__` directly, but in practice you wouldn't actually do that; I'm just doing it here to show you how it works. The right way to use `__getitem__` is to get Python to call it for you.

(2) This looks just like the syntax you would use to get a dictionary value, and in fact it returns the value you would expect. But here's the missing link: under the covers, Python has converted this syntax to the method call `f.__getitem__("name")`. That's why `__getitem__` is a special class method; not only can you call it yourself, you can get Python to call it for you by using the right syntax.

Example 3.14. The `__setitem__` special method

```
    def __setitem__(self, key, item): self.data[key] = item
```

```
>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__setitem__("genre", 31) (1)
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':31}
>>> f["genre"] = 32            (2)
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':32}
```

(1) Like the `__getitem__` method, `__setitem__` simply redirects to the real dictionary `self.data` to do its work. And like `__getitem__`, you wouldn't ordinarily call it directly like this; Python calls `__setitem__` for you when you use the right syntax.

(2) This looks like regular dictionary syntax, except of course that `f` is really a class that's trying very hard to masquerade as a dictionary, and `__setitem__` is an essential part of that masquerade. This line of code actually calls `f.__setitem__("genre", 32)` under the covers.

`__setitem__` is a special class method because it gets called for you, but it's still a class method. Just as easily as the `__setitem__` method was defined in `UserDict`, we can redefine it in our descendant class to override the ancestor method. This allows us to define classes that act like dictionaries in some ways but define their own behavior above and beyond the built−in dictionary.

This concept is the basis of the entire framework we're studying in this chapter. Each file type can have a handler class which knows how to get metadata from a particular type of file. Once some attributes (like the file's name and location) are known, the handler class knows how to derive other attributes automatically. This is done by overriding the `__setitem__` method, checking for particular keys, and adding additional processing when they are found.

For example, `MP3FileInfo` is a descendant of `FileInfo`. When an `MP3FileInfo`'s `name` is set, it doesn't just set the `name` key (like the ancestor `FileInfo` does); it also looks in the file itself for MP3 tags and populates a whole set of keys.

Example 3.15. Overriding `__setitem__` in `MP3FileInfo`

```
    def __setitem__(self, key, item):         (1)
        if key == "name" and item:            (2)
            self.__parse(item)                (3)
        FileInfo.__setitem__(self, key, item) (4)
```

(1)

Note that our `__setitem__` method is defined exactly the same way as the ancestor method. This is important, since Python will be calling the method for us, and it expects it to be defined with a certain number of arguments. (Technically speaking, the names of the arguments don't matter, just the number.)

(2) Here's the crux of the entire `MP3FileInfo` class: if we're assigning a value to the `name` key, we want to do something extra.

(3) The extra processing we do for `name`s is encapsulated in the `__parse` method. This is another class method defined in `MP3FileInfo`, and when we call it, we qualify it with `self`. Just calling `__parse` would look for a normal function defined outside the class, which is not what we want; calling `self.__parse` will look for a class method defined within the class. This isn't anything new; you reference data attributes the same way.

(4) After doing our extra processing, we want to call the ancestor method. Remember, this is never done for you in Python; you have to do it manually. Note that we're calling the immediate ancestor, `FileInfo`, even though it doesn't have a `__setitem__` method. That's okay, because Python will walk up the ancestor tree until it finds a class with the method we're calling, so this line of code will eventually find and call the `__setitem__` defined in `UserDict`.

Note: Calling other class methods

When accessing data attributes within a class, you need to qualify the attribute name: `self.`*`attribute`*. When calling other methods within a class, you need to qualify the method name: `self.`*`method`*.

Example 3.16. Setting an `MP3FileInfo`'s `name`

```
>>> import fileinfo
>>> mp3file = fileinfo.MP3FileInfo()                   (1)
>>> mp3file
{'name':None}
>>> mp3file["name"] = "/music/_singles/kairo.mp3"      (2)
>>> mp3file
{'album': 'Rave Mix', 'artist': '***DJ MARY-JANE***', 'genre': 31,
'title': 'KAIRO****THE BEST GOA', 'name': '/music/_singles/kairo.mp3',
'year': '2000', 'comment': 'http://mp3.com/DJMARYJANE'}
>>> mp3file["name"] = "/music/_singles/sidewinder.mp3" (3)
>>> mp3file
{'album': '', 'artist': 'The Cynic Project', 'genre': 18, 'title': 'Sidewinder',
'name': '/music/_singles/sidewinder.mp3', 'year': '2000',
'comment': 'http://mp3.com/cynicproject'}
```

(1) First, we create an instance of `MP3FileInfo`, without passing it a filename. (We can get away with this because the `filename` argument of the `__init__` method is optional.) Since `MP3FileInfo` has no `__init__` method of its own, Python walks up the ancestor tree and finds the `__init__` method of `FileInfo`. This `__init__` method manually calls the `__init__` method of `UserDict` and then sets the `name` key to `filename`, which is `None`, since we didn't pass a filename. Thus, `mp3file` initially looks like a dictionary with one key, `name`, whose value is `None`.

(2) Now the real fun begins. Setting the `name` key of `mp3file` triggers the `__setitem__` method on `MP3FileInfo` (not `UserDict`), which notices that we're setting the `name` key with a real value and calls `self.__parse`. Although we haven't traced through the `__parse` method yet, you can see from the output that it sets several other keys: `album`, `artist`, `genre`, `title`, `year`, and `comment`.

(3) Modifying the `name` key will go through the same process again: Python calls `__setitem__`, which calls `self.__parse`, which sets all the other keys.

3.7. Advanced special class methods

There are more special methods than just `__getitem__` and `__setitem__`. Some of them let you emulate functionality that you may not even know about.

Example 3.17. More special methods in `UserDict`

```
    def __repr__(self): return repr(self.data)     (1)
    def __cmp__(self, dict):                       (2)
        if isinstance(dict, UserDict):
            return cmp(self.data, dict.data)
        else:
            return cmp(self.data, dict)
    def __len__(self): return len(self.data)       (3)
    def __delitem__(self, key): del self.data[key] (4)
```

(1) `__repr__` is a special method which is called when you call `repr(`*`instance`*`)`. The `repr` function is a built−in function that returns a string representation of an object. It works on any object, not just class instances. You're already intimately familiar with `repr` and you don't even know it. In the interactive window, when you type just a variable name and hit **ENTER**, Python uses `repr` to display the variable's value. Go create a dictionary `d` with some data and then `print repr(d)` to see for yourself.

(2) `__cmp__` is called when you compare class instances. In general, you can compare any two Python objects, not just class instances, by using `==`. There are rules that define when built−in datatypes are considered equal; for instance, dictionaries are equal when they have all the same keys and values, and strings are equal when they are the same length and contain the same sequence of characters. For class instances, you can define the `__cmp__` method and code the comparison logic yourself, and then you can use `==` to compare instances of your class and Python will call your `__cmp__` special method for you.

(3) `__len__` is called when you call `len(`*`instance`*`)`. The `len` function is a built−in function that returns the length of an object. It works on any object that could reasonably be thought of as having a length. The `len` of a string is its number of characters; the `len` of a dictionary is its number of keys; the `len` of a list or tuple is its number of elements. For class instances, define the `__len__` method and code the length calculation yourself, then call `len(`*`instance`*`)` and Python will call your `__len__` special method for you.

(4) `__delitem__` is called when you call `del `*`instance`*`[`*`key`*`]`, which you may remember as the way to delete individual items from a dictionary. When you use `del` on a class instance, Python calls the `__delitem__` special method for you.

Note: Python vs. Java: equality and identity
In Java, you determine whether two string variables reference the same physical memory location by using `str1 == str2`. This is called *object identity*, and it is written in Python as `str1 is str2`. To compare string values in Java, you would use `str1.equals(str2)`; in Python, you would use `str1 == str2`. Java programmers who have been taught to believe that the world is a better place because `==` in Java compares by identity instead of by value may have a difficult time adjusting to Python's lack of such "gotchas".

At this point, you may be thinking, "all this work just to do something in a class that I can do with a built−in datatype". And it's true that life would be easier (and the entire `UserDict` class would be unnecessary) if you could inherit from built−in datatypes like a dictionary. But even if you could, special methods would still be useful, because they can be used in any class, not just wrapper classes like `UserDict`.

Special methods mean that *any class* can store key−value pairs like a dictionary, just by defining the `__setitem__` method. *Any class* can act like a sequence, just by defining the `__getitem__` method. Any class that defines the `__cmp__` method can be compared with `==`. And if your class represents something that has a length, don't define a `GetLength` method; define the `__len__` method and use `len(instance)`.

> **Note: Physical vs. logical models**
> While other object−oriented languages only let you define the physical model of an object ("this object has a `GetLength` method"), Python's special class methods like `__len__` allow you to define the logical model of an object ("this object has a length").

There are lots of other special methods. There's a whole set of them that let classes act like numbers, allowing you to add, subtract, and do other arithmetic operations on class instances. (The canonical example of this is a class that represents complex numbers, numbers with both real and imaginary components.) The `__call__` method lets a class act like a function, allowing you to call a class instance directly. And there are other special methods that allow classes to have read−only and write−only data attributes; we'll talk more about those in later chapters.

Further reading

- *Python Reference Manual* documents all the special class methods.

3.8. Class attributes

You already know about data attributes, which are variables owned by a specific instance of a class. Python also supports class attributes, which are variables owned by the class itself.

Example 3.18. Introducing class attributes

```
class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}
```

```
>>> import fileinfo
>>> fileinfo.MP3FileInfo            (1)
<class fileinfo.MP3FileInfo at 01257FDC>
>>> fileinfo.MP3FileInfo.tagDataMap (2)
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
>>> m = fileinfo.MP3FileInfo()      (3)
>>> m.tagDataMap
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
```

(1) MP3FileInfo is the class itself, not any particular instance of the class.

(2) tagDataMap is a class attribute: literally, an attribute of the class. It is available before creating any instances of the class.

(3) Class attributes are available both through direct reference to the class and through any instance of the class.

Note: Class attributes in Java
In Java, both static variables (called class attributes in Python) and instance variables (called data attributes in Python) are defined immediately after the class definition (one with the static keyword, one without). In Python, only class attributes can be defined here; data attributes are defined in the __init__ method.

Class attributes can be used as class−level constants (which is how we use them in MP3FileInfo), but they are not really constants.[4] You can also change them.

Example 3.19. Modifying class attributes

```
>>> class counter:
...     count = 0                     (1)
...     def __init__(self):
...         self.__class__.count += 1 (2)
...
>>> counter
<class __main__.counter at 010EAECC>
>>> counter.count                     (3)
0
>>> c = counter()
>>> c.count                           (4)
1
>>> counter.count
1
>>> d = counter()                     (5)
>>> d.count
2
>>> c.count
2
>>> counter.count
2
```

(1) count is a class attribute of the counter class.

(2) __class__ is a built−in attribute of every class instance (of every class). It is a reference to the class that self is an instance of (in this case, the counter class).

(3) Because count is a class attribute, it is available through direct reference to the class, before we have created any instances of the class.

(4) Creating an instance of the class calls the __init__ method, which increments the class attribute count by 1. This affects the class itself, not just the newly created instance.

(5) Creating a second instance will increment the class attribute count again. Notice how the class attribute is shared by the class and all instances of the class.

3.9. Private functions

Like most languages, Python has the concept of private functions, which can not be called from outside their module; private class methods, which can not be called from outside their class; and private attributes, which can not be accessed from outside their class. Unlike most languages, whether a Python function, method, or attribute is private or public is determined entirely by its name.

In MP3FileInfo, there are two methods: __parse and __setitem__. As we have already discussed, __setitem__ is a special method; normally, you would call it indirectly by using the dictionary syntax on a class instance, but it is public, and you could call it directly (even from outside the fileinfo module) if you had a really good reason. However, __parse is private, because it has two underscores at the beginning of its name.

Note: What's private in Python?
If the name of a Python function, class method, or attribute starts with (but doesn't end with) two underscores, it's private; everything else is public.

Note: Method naming conventions
In Python, all special methods (like __setitem__) and built−in attributes (like __doc__) follow a standard naming convention: they both start with and end with two underscores. Don't name your own methods and attributes this way; it will only confuse you (and others) later.

Note: No protected methods
Python has no concept of protected class methods (accessible only in their own class and descendant classes). Class methods are either private (accessible only in their own class) or public (accessible from anywhere).

Example 3.20. Trying to call a private method

```
>>> import fileinfo
>>> m = fileinfo.MP3FileInfo()
>>> m.__parse("/music/_singles/kairo.mp3") (1)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'MP3FileInfo' instance has no attribute '__parse'
```

(1) If you try to call a private method, Python will raise a slightly misleading exception, saying that the method does not exist. Of course it does exist, but it's private, so it's not accessible outside the class.[5]

Further reading

- *Python Tutorial* discusses the inner workings of private variables.

3.10. Handling exceptions

Like many object−oriented languages, Python has exception handling via try...except blocks.

Note: Python vs. Java: exception handling
Python uses try...except to handle exceptions and raise to generate them. Java and C++ use try...catch to handle exceptions, and throw to generate them.

If you already know all about exceptions, you can skim this section. If you've been stuck programming in a lesser language that doesn't have exception handling, or you've been using a real language but not using exceptions, this section is very important.

Exceptions are everywhere in Python; virtually every module in the standard Python library uses them, and Python itself will raise them in lots of different circumstances. You've already seen them repeatedly throughout this book.

- Accessing a non−existent dictionary key will raise a KeyError exception.
- Searching a list for a non−existent value will raise a ValueError exception.
- Calling a non−existent method will raise an AttributeError exception.

* Referencing a non−existent variable will raise a `NameError` exception.
* Mixing datatypes without coercion will raise a `TypeError` exception.

In each of these cases, we were simply playing around in the Python IDE: an error occurred, the exception was printed (depending on your IDE, in an intentionally jarring shade of red), and that was that. This is called an *unhandled* exception; when the exception was raised, there was no code to explicitly notice it and deal with it, so it bubbled its way back to the default behavior built in to Python, which is to spit out some debugging information and give up. In the IDE, that's no big deal, but if that happened while your actual Python program was running, the entire program would come to a screeching halt.[6]

An exception doesn't have to be a complete program crash, though. Exceptions, when raised, can be *handled*. Sometimes an exception is really because you have a bug in your code (like accessing a variable that doesn't exist), but many times, an exception is something you can plan for. If you're opening a file, it might not exist; if you're connecting to a database, it might be unavailable, or you might not have the correct security credentials to access it. If you know a line of code may raise an exception, you should handle the exception using a `try...except` block.

Example 3.21. Opening a non−existent file

```
>>> fsock = open("/notthere", "r")      (1)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: '/notthere'
>>> try:
...     fsock = open("/notthere")       (2)
... except IOError:                     (3)
...     print "The file does not exist, exiting gracefully"
... print "This line will always print" (4)
The file does not exist, exiting gracefully
This line will always print
```

(1) Using the built−in `open` function, we can try to open a file for reading (more on `open` in the next section). But the file doesn't exist, so this raises the `IOError` exception. Since we haven't provided any explicit check for an `IOError` exception, Python just prints out some debugging information about what happened and then gives up.

(2) We're trying to open the same non−existent file, but this time we're doing it within a `try...except` block.

(3) When the `open` method raises an `IOError` exception, we're ready for it. The `except IOError:` line catches the exception and executes our own block of code, which in this case just prints a more pleasant error message.

(4) Once an exception has been handled, processing continues normally on the first line after the `try...except` block. Note that this line will always print, whether or not an exception occurs. If you really did have a file called `notthere` in your root directory, the call to `open` would succeed, the `except` clause would be ignored, and this line would still be executed.

Exceptions may seem unfriendly (after all, if you don't catch the exception, your entire program will crash), but consider the alternative. Would you rather get back an unusable file object to a non−existent file? You'd have to check its validity somehow anyway, and if you forgot, your program would give you strange errors somewhere down the line that you would have to trace back to the source. I'm sure you've done this; it's not fun. With exceptions, errors occur immediately, and you can handle them in a standard way at the source of the problem.

There are lots of other uses for exceptions besides handling actual error conditions. A common use in the standard Python library is to try to import a module, then check whether it worked. Importing a module that does not exist will raise an `ImportError` exception. You can use this to define multiple levels of functionality based on which modules are available at run−time, or to support multiple platforms (where platform−specific code is separated into different modules).

You can also define your own exceptions by creating a class that inherits from the built−in `Exception` class, and then raise your exceptions with the `raise` command. This is beyond the scope of this section, but see the further reading section if you're interested.

Example 3.22. Supporting platform−specific functionality

This code comes from the `getpass` module, a wrapper module for getting a password from the user. Getting a password is accomplished differently on UNIX, Windows, and Mac OS platforms, but this code encapsulates all of those differences.

```
# Bind the name getpass to the appropriate function
try:
    import termios, TERMIOS                     (1)
except ImportError:
    try:
        import msvcrt                           (2)
    except ImportError:
        try:
            from EasyDialogs import AskPassword (3)
        except ImportError:
            getpass = default_getpass           (4)
        else:                                   (5)
            getpass = AskPassword
    else:
        getpass = win_getpass
else:
    getpass = unix_getpass
```

(1) `termios` is a UNIX−specific module that provides low−level control over the input terminal. If this module is not available (because it's not on your system, or your system doesn't support it), the import fails and Python raises an `ImportError`, which we catch.

(2) OK, we didn't have `termios`, so let's try `msvcrt`, which is a Windows−specific module that provides an API to lots of useful functions in the Microsoft Visual C++ runtime services. If this import fails, Python will raise an `ImportError`, which we catch.

(3) If the first two didn't work, we try to import a function from `EasyDialogs`, which is a Mac OS−specific module that provides functions to pop up dialogs of various types. Once again, if this import fails, Python will raise an `ImportError`, which we catch.

(4) None of these platform−specific modules is available (which is possible, since Python has been ported to lots of different platforms), so we have to fall back on a default password input function (which is defined elsewhere in the `getpass` module). Notice what we're doing here: we're assigning the function `default_getpass` to the variable `getpass`. If you read the official `getpass` documentation, it tells you that the `getpass` module defines a `getpass` function. This is how it does it: by binding `getpass` to the right function for your platform. Then when you call the `getpass` function, you're really calling a platform−specific function that this code has set up for you. You don't have to know or care what platform your code is running on; just call `getpass`, and it will always do the right thing.

(5) A `try...except` block can have an `else` clause, like an `if` statement. If no exception is raised during the `try` block, the `else` clause is executed afterwards. In this case, that means that the `from EasyDialogs import AskPassword` import worked, so we should bind `getpass` to the `AskPassword` function. Each of the other `try...except` blocks have similar `else` clauses to bind `getpass` to the appropriate function when we find an `import` that works.

Further reading

* *Python Tutorial* discusses defining and raising your own exceptions, and handling multiple exceptions at once.
* *Python Library Reference* summarizes all the built−in exceptions.
* *Python Library Reference* documents the getpass module.
* *Python Library Reference* documents the `traceback` module, which provides low−level access to exception attributes after an exception is raised.
* *Python Reference Manual* discusses the inner workings of the `try...except` block.

3.11. File objects

Python has a built−in function, `open`, for opening a file on disk. `open` returns a file object, which has methods and attributes for getting information about and manipulating the opened file.

Example 3.23. Opening a file

```
>>> f = open("/music/_singles/kairo.mp3", "rb") (1)
>>> f                                           (2)
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.mode                                      (3)
'rb'
>>> f.name                                      (4)
'/music/_singles/kairo.mp3'
```

(1) The `open` method can take up to three parameters: a filename, a mode, and a buffering parameter. Only the first one, the filename, is required; the other two are optional. If not specified, the file is opened for reading in text mode. Here we are opening the file for reading in binary mode. (`print open.__doc__` displays a great explanation of all the possible modes.)

(2) The `open` function returns an object (by now, this should not surprise you). A file object has several useful attributes.

(3) The `mode` attribute of a file object tells you what mode the file was opened in.

(4) The `name` attribute of a file object tells you the name of the file that the file object has open.

Example 3.24. Reading a file

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.tell()              (1)
0
>>> f.seek(−128, 2)       (2)
>>> f.tell()              (3)
7542909
>>> tagData = f.read(128) (4)
>>> tagData
'TAGKAIRO****THE BEST GOA         ***DJ MARY−JANE***            Rave Mix                      2000htt
>>> f.tell()              (5)
7543037
```

(1) A file object maintains state about the file it has open. The `tell` method of a file object tells you your current position in the open file. Since we haven't done anything with this file yet, the current position is `0`, which is the beginning of the file.

(2) The `seek` method of a file object moves to another position in the open file. The second parameter specifies what the first one means; `0` means move to an absolute position (counting from the start of the file), `1` means move to a relative position (counting from the current position), and `2` means move to a

position relative to the end of the file. Since the MP3 tags we're looking for are stored at the end of the file, we use `2` and tell the file object to move to a position `128` bytes from the end of the file.

(3) The `tell` method confirms that the current file position has moved.

(4) The `read` method reads a specified number of bytes from the open file and returns a string with the data which was read. The optional parameter specifies the maximum number of bytes to read. If no parameter is specified, `read` will read until the end of the file. (We could have simply said `read()` here, since we know exactly where we are in the file and we are, in fact, reading the last 128 bytes.) The read data is assigned to the `tagData` variable, and the current position is updated based on how many bytes were read.

(5) The `tell` method confirms that the current position has moved. If you do the math, you'll see that after reading 128 bytes, the position has been incremented by 128.

Example 3.25. Closing a file

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed  (1)
0
>>> f.close() (2)
>>> f
<closed file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed
1
>>> f.seek(0) (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.tell()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.read()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.close() (4)
```

(1) The `closed` attribute of a file object indicates whether the object has a file open or not. In this case, the file is still open (`closed` is `0`). Open files consume system resources, and depending on the file mode, other programs may not be able to access them. It's important to close files as soon as you're done with them.

(2) To close a file, call the `close` method of the file object. This frees the lock (if any) that you were holding on the file, flushes buffered writes (if any) that the system hadn't gotten around to actually writing yet, and releases the system resources. The `closed` attribute confirms that the file is closed.

(3) Just because a file is closed doesn't mean that the file object ceases to exist. The variable `f` will continue to exist until it goes out of scope or gets manually deleted. However, none of the methods that manipulate an open file will work once the file has been closed; they all raise an exception.

(4) Calling `close` on a file object whose file is already closed does *not* raise an exception; it fails silently.

Example 3.26. File objects in `MP3FileInfo`

```
        try:                                (1)
            fsock = open(filename, "rb", 0) (2)
            try:
                fsock.seek(-128, 2)         (3)
                tagdata = fsock.read(128)   (4)
```

```
            finally:                            (5)
                fsock.close()
            .
            .
            .
        except IOError:                         (6)
            pass
```

(1) Because opening and reading files is risky and may raise an exception, all of this code is wrapped in a `try...except` block. (Hey, isn't standardized indentation great? This is where you start to appreciate it.)

(2) The `open` function may raise an `IOError`. (Maybe the file doesn't exist.)

(3) The `seek` method may raise an `IOError`. (Maybe the file is smaller than 128 bytes.)

(4) The `read` method may raise an `IOError`. (Maybe the disk has a bad sector, or it's on a network drive and the network just went down.)

(5) This is new: a `try...finally` block. Once the file has been opened successfully by the `open` function, we want to make absolutely sure that we close it, even if an exception is raised by the `seek` or `read` methods. That's what a `try...finally` block is for: code in the `finally` block will *always* be executed, even if something in the `try` block raises an exception. Think of it as code that gets executed "on the way out", regardless of what happened on the way.

(6) At last, we handle our `IOError` exception. This could be the `IOError` exception raised by the call to `open`, `seek`, or `read`. Here, we really don't care, because all we're going to do is ignore it silently and continue. (Remember, `pass` is a Python statement that does nothing.) That's perfectly legal; "handling" an exception can mean explicitly doing nothing. It still counts as handled, and processing will continue normally on the next line of code after the `try...except` block.

Further reading

- *Python Tutorial* discusses reading and writing files, including how to read a file one line at a time into a list.
- eff−bot discusses efficiency and performance of various ways of reading a file.
- Python Knowledge Base answers common questions about files.
- *Python Library Reference* summarizes all the file object methods.

3.12. `for` loops

Like most other languages, Python has `for` loops. The only reason you haven't seen them until now is that Python is good at so many other things that you don't need them as often.

Most other languages don't have a powerful list datatype like Python, so you end up doing a lot of manual work, specifying a start, end, and step to define a range of integers or characters or other iteratable entities. But in Python, a `for` loop simply iterates over a list, the same way list comprehensions work.

Example 3.27. Introducing the `for` loop

```
>>> li = ['a', 'b', 'e']
>>> for s in li:         (1)
...     print s          (2)
a
b
e
>>> print "\n".join(li)  (3)
a
b
e
```

(1) The syntax for a `for` loop is similar to list comprehensions. `li` is a list, and `s` will take the value of each element in turn, starting from the first element.

(2) Like an `if` statement or any other indented block, a `for` loop can have any number of lines of code in it.

(3) This is the reason you haven't seen the `for` loop yet: we haven't needed it yet. It's amazing how often you use `for` loops in other languages when all you really want is a `join` or a list comprehension.

Example 3.28. Simple counters

```
>>> for i in range(5):       (1)
...     print i
0
1
2
3
4
>>> li = ['a', 'b', 'c', 'd', 'e']
>>> for i in range(len(li)): (2)
...     print li[i]
a
b
c
d
e
```

(1) Doing a "normal" (by Visual Basic standards) counter `for` loop is also simple. As we saw in Example 1.28, `range` produces a list of integers, which we then loop through. I know it looks a bit odd, but it is occasionally (and I stress *occasionally*) useful to have a counter loop.

(2) Don't ever do this. This is Visual Basic−style thinking. Break out of it. Just iterate through the list, as shown in the previous example.

Example 3.29. Iterating through a dictionary

```
>>> for k, v in os.environ.items(): (1) (2)
...     print "%s=%s" % (k, v)
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...snip...]
>>> print "\n".join(["%s=%s" % (k, v) for k, v in os.environ.items()]) (3)
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...snip...]
```

(1) `os.environ` is a dictionary of the environment variables defined on your system. In Windows, these are your user and system variables accessible from MS−DOS. In UNIX, they are the variables exported in your shell's startup scripts. In Mac OS, there is no concept of environment variables, so this dictionary is empty.

(2) `os.environ.items()` returns a list of tuples: `[(`*`key1`*`, `*`value1`*`), (`*`key2`*`, `*`value2`*`), ...]`. The `for` loop iterates through this list. The first round, it assigns *`key1`* to `k` and *`value1`* to `v`, so `k` = `USERPROFILE` and `v` = `C:\Documents and`

Settings\mpilgrim. The second round, k gets the second key, OS, and v gets the corresponding value, Windows_NT.

(3) With multi−variable assignment and list comprehensions, you can replace the entire for loop with a single statement. Whether you actually do this in real code is a matter of personal coding style; I like it because it makes it clear that what we're doing is mapping a dictionary into a list, then joining the list into a single string. Other programmers prefer to write this out as a for loop. Note that the output is the same in either case, although this version is slightly faster, because there is only one print statement instead of many.

Example 3.30. for loop in MP3FileInfo

```
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)} (1)
    .
    .
    .
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items(): (2)
                    self[tag] = parseFunc(tagdata[start:end])                (3)
```

(1) tagDataMap is a class attribute that defines the tags we're looking for in an MP3 file. Tags are stored in fixed−length fields; once we read the last 128 bytes of the file, bytes 3 through 32 of those are always the song title, 33−62 are always the artist name, 63−92 the album name, and so forth. Note that tagDataMap is a dictionary of tuples, and each tuple contains two integers and a function reference.

(2) This looks complicated, but it's not. The structure of the for variables matches the structure of the elements of the list returned by items. Remember, items returns a list of tuples of the form (*key*, *value*). The first element of that list is ("title", (3, 33, <function stripnulls>)), so the first time around the loop, tag gets "title", start gets 3, end gets 33, and parseFunc gets the function stripnulls.

(3) Now that we've extracted all the parameters for a single MP3 tag, saving the tag data is easy. We slice tagdata from start to end to get the actual data for this tag, call parseFunc to post−process the data, and assign this as the value for the key tag in the pseudo−dictionary self. After iterating through all the elements in tagDataMap, self has the values for all the tags, and you know what that looks like.

3.13. More on modules

Modules, like everything else in Python, are objects. Once imported, you can always get a reference to a module through the global dictionary sys.modules.

Example 3.31. Introducing sys.modules

```
>>> import sys                          (1)
>>> print '\n'.join(sys.modules.keys()) (2)
win32api
os.path
os
exceptions
__main__
ntpath
nt
sys
__builtin__
```

```
site
signal
UserDict
stat
```

(1) The `sys` module contains system−level information, like the version of Python you're running (`sys.version` or `sys.version_info`), and system−level options like the maximum allowed recursion depth (`sys.getrecursionlimit()` and `sys.setrecursionlimit()`).

(2) `sys.modules` is a dictionary containing all the modules that have ever been imported since Python was started; the key is the module name, the value is the module object. Note that this is more than just the modules *your* program has imported. Python preloads some modules on startup, and if you're in a Python IDE, `sys.modules` contains all the modules imported by all the programs you've run within the IDE.

Example 3.32. Using `sys.modules`

```
>>> import fileinfo         (1)
>>> print '\n'.join(sys.modules.keys())
win32api
os.path
os
fileinfo
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
>>> fileinfo
<module 'fileinfo' from 'fileinfo.pyc'>
>>> sys.modules["fileinfo"] (2)
<module 'fileinfo' from 'fileinfo.pyc'>
```

(1) As new modules are imported, they are added to `sys.modules`. This explains why importing the same module twice is very fast: Python has already loaded and cached the module in `sys.modules`, so importing the second time is simply a dictionary lookup.

(2) Given the name (as a string) of any previously−imported module, you can get a reference to the module itself through the `sys.modules` dictionary.

Example 3.33. The `__module__` class attribute

```
>>> from fileinfo import MP3FileInfo
>>> MP3FileInfo.__module__              (1)
'fileinfo'
>>> sys.modules[MP3FileInfo.__module__] (2)
<module 'fileinfo' from 'fileinfo.pyc'>
```

(1) Every Python class has a built−in class attribute `__module__`, which is the name of the module in which the class is defined.

(2) Combining this with the `sys.modules` dictionary, you can get a reference to the module in which a class is defined.

Example 3.34. `sys.modules` in `fileinfo.py`

```
def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):       (1)
    "get file info class from filename extension"
    subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]        (2)
    return hasattr(module, subclass) and getattr(module, subclass) or FileInfo (3)
```

(1) This is a function with two arguments; `filename` is required, but `module` is optional and defaults to the module which contains the `FileInfo` class. This looks inefficient, because you might expect Python to evaluate the `sys.modules` expression every time the function is called. In fact, Python only evaluates default expressions once, the first time the module is imported. As we'll see later, we never call this function with a `module` argument, so `module` serves as a function−level constant.

(2) We'll plough through this line later, after we dive into the `os` module. For now, take it on faith that `subclass` ends up as the name of a class, like `MP3FileInfo`.

(3) You already know about `getattr`, which gets a reference to an object by name. `hasattr` is a complementary function that checks whether an object has a particular attribute; in this case, whether a module has a particular class (although it works for any object and any attribute, just like `getattr`). In English, this line of code says "if this module has the class named by `subclass` then return it, otherwise return the base class `FileInfo`".

Further reading

- *Python Tutorial* discusses exactly when and how default arguments are evaluated.
- *Python Library Reference* documents the `sys` module.

3.14. The os module

The `os` module has lots of useful functions for manipulating files and processes, and `os.path` has functions for manipulating file and directory paths.

Example 3.35. Constructing pathnames

```
>>> import os
>>> os.path.join("c:\\music\\ap\\", "mahadeva.mp3") (1) (2)
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.join("c:\\music\\ap", "mahadeva.mp3")   (3)
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.expanduser("~")                         (4)
'c:\\Documents and Settings\\mpilgrim\\My Documents'
>>> os.path.join(os.path.expanduser("~"), "Python") (5)
'c:\\Documents and Settings\\mpilgrim\\My Documents\\Python'
```

(1) `os.path` is a reference to a module; which module it is depends on what platform you're running on. Just like `getpass` encapsulates differences between platforms by setting `getpass` to a platform−specific function, `os` encapsulates differences between platforms by setting `path` to a platform−specific module.

(2) The `join` function of `os.path` constructs a pathname out of one or more partial pathnames. In this simple case, it simply concatenates strings. (Note that dealing with pathnames on Windows is annoying because the backslash character must be escaped.)

(3) In this slightly less trivial case, `join` will add an extra backslash to the pathname before joining it to the filename. I was overjoyed when I discovered this, since `addSlashIfNecessary` is always one of the stupid little functions I have to write when building up my toolbox in a new language. *Do not* write this stupid little function in Python; smart people have already taken care of it for you.

(4) `expanduser` will expand a pathname that uses ~ to represent the current user's home directory. This works on any platform where users have a home directory, like Windows, UNIX, and Mac OS X; it has no effect on Mac

OS.

(5) Combining these techniques, you can easily construct pathnames for directories and files under the user's home directory.

Example 3.36. Splitting pathnames

```
>>> os.path.split("c:\\music\\ap\\mahadeva.mp3")                        (1)
('c:\\music\\ap', 'mahadeva.mp3')
>>> (filepath, filename) = os.path.split("c:\\music\\ap\\mahadeva.mp3") (2)
>>> filepath                                                            (3)
'c:\\music\\ap'
>>> filename                                                            (4)
'mahadeva.mp3'
>>> (shortname, extension) = os.path.splitext(filename)                 (5)
>>> shortname
'mahadeva'
>>> extension
'.mp3'
```

(1) The `split` function splits a full pathname and returns a tuple containing the path and filename. Remember when I said you could use multi−variable assignment to return multiple values from a function? Well, `split` is such a function.

(2) We assign the return value of the `split` function into a tuple of two variables. Each variable receives the value of the corresponding element of the returned tuple.

(3) The first variable, `filepath`, receives the value of the first element of the tuple returned from `split`, the file path.

(4) The second variable, `filename`, receives the value of the second element of the tuple returned from `split`, the filename.

(5) `os.path` also contains a function `splitext`, which splits a filename and returns a tuple containing the filename and the file extension. We use the same technique to assign each of them to separate variables.

Example 3.37. Listing directories

```
>>> os.listdir("c:\\music\\_singles\\")                                          (1)
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3', 'kairo.mp3',
'long_way_home1.mp3', 'sidewinder.mp3', 'spinning.mp3']
>>> dirname = "c:\\"
>>> os.listdir(dirname)                                                          (2)
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'cygwin', 'docbook',
'Documents and Settings', 'Incoming', 'Inetpub', 'IO.SYS', 'MSDOS.SYS', 'Music',
'NTDETECT.COM', 'ntldr', 'pagefile.sys', 'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
>>> [f for f in os.listdir(dirname) if os.path.isfile(os.path.join(dirname, f))] (3)
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'IO.SYS', 'MSDOS.SYS',
'NTDETECT.COM', 'ntldr', 'pagefile.sys']
>>> [f for f in os.listdir(dirname) if os.path.isdir(os.path.join(dirname, f))]  (4)
['cygwin', 'docbook', 'Documents and Settings', 'Incoming',
'Inetpub', 'Music', 'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
```

(1) The `listdir` function takes a pathname and returns a list of the contents of the directory.

(2) `listdir` returns both files and folders, with no indication of which is which.

(3) You can use list filtering and the `isfile` function of the `os.path` module to separate the files from the folders. `isfile` takes a pathname and returns 1 if the path represents a file, and 0 otherwise. Here we're using `os.path.join` to ensure a full pathname, but `isfile` also works with a partial path,

relative to the current working directory. You can use `os.getcwd()` to get the current working directory.

(4) `os.path` also has a `isdir` function which returns 1 if the path represents a directory, and 0 otherwise. You can use this to get a list of the subdirectories within a directory.

Example 3.38. Listing directories in `fileinfo.py`

```
def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]
```

These two lines of code combine everything we've learned so far about the `os` module, and then some.

1. `os.listdir(directory)` returns a list of all the files and folders in `directory`.
2. Iterating through the list with `f`, we use `os.path.normcase(f)` to normalize the case according to operating system defaults. `normcase` is a useful little function that compensates for case−insensitive operating systems that think that `mahadeva.mp3` and `mahadeva.MP3` are the same file. For instance, on Windows and Mac OS, `normcase` will convert the entire filename to lowercase; on UNIX−compatible systems, it will return the filename unchanged.
3. Iterating through the normalized list with `f` again, we use `os.path.splitext(f)` to split each filename into name and extension.
4. For each file, we see if the extension is in the list of file extensions we care about (`fileExtList`, which was passed to the `listDirectory` function).
5. For each file we care about, we use `os.path.join(directory, f)` to construct the full pathname of the file, and return a list of the full pathnames.

Note: When to use the os module
Whenever possible, you should use the functions in `os` and `os.path` for file, directory, and path manipulations. These modules are wrappers for platform−specific modules, so functions like `os.path.split` work on UNIX, Windows, Mac OS, and any other supported Python platform.

Further reading

- Python Knowledge Base answers questions about the `os` module.
- *Python Library Reference* documents the `os` module and the `os.path` module.

3.15. Putting it all together

Once again, all the dominoes are in place. We've seen how each line of code works. Now let's step back and see how it all fits together.

Example 3.39. `listDirectory`

```
def listDirectory(directory, fileExtList):                                 (1)
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]                  (2)
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):   (3)
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]    (4)
```

```
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo (5)
    return [getFileInfoClass(f)(f) for f in fileList]                            (6)
```

(1) `listDirectory` is the main attraction of this entire module. It takes a directory (like `c:\music\_singles\` in my case) and a list of interesting file extensions (like `['.mp3']`), and it returns a list of class instances that act like dictionaries that contain metadata about each interesting file in that directory. And it does it in just a few straightforward lines of code.

(2) As we saw in the previous section, this line of code gets a list of the full pathnames of all the files in `directory` that have an interesting file extension (as specified by `fileExtList`).

(3) Old−school Pascal programmers may be familiar with them, but most people give me a blank stare when I tell them that Python supports *nested functions* −− literally, a function within a function. The nested function `getFileInfoClass` can only be called from the function in which it is defined, `listDirectory`. As with any other function, you don't need an interface declaration or anything fancy; just define the function and code it.

(4) Now that you've seen the `os` module, this line should make more sense. It gets the extension of the file (`os.path.splitext(filename)[1]`), forces it to uppercase (`.upper()`), slices off the dot (`[1:]`), and constructs a class name out of it with string formatting. So `c:\music\ap\mahadeva.mp3` becomes `.mp3` becomes `.MP3` becomes `MP3` becomes `MP3FileInfo`.

(5) Having constructed the name of the handler class that would handle this file, we check to see if that handler class actually exists in this module. If it does, we return the class, otherwise we return the base class `FileInfo`. This is a very important point: *this function returns a class*. Not an instance of a class, but the class itself.

(6) For each file in our "interesting files" list (`fileList`), we call `getFileInfoClass` with the filename (`f`). Calling `getFileInfoClass(f)` returns a class; we don't know exactly which class, but we don't care. We then create an instance of this class (whatever it is) and pass the filename (`f` again), to the `__init__` method. As we saw earlier in this chapter, the `__init__` method of `FileInfo` sets `self["name"]`, which triggers `__setitem__`, which is overridden in the descendant (`MP3FileInfo`) to parse the file appropriately to pull out the file's metadata. We do all that for each interesting file and return a list of the resulting instances.

Note that `listDirectory` is completely generic. It doesn't know ahead of time which types of files it will be getting, or which classes are defined that could potentially handle those files. It inspects the directory for the files to process, then introspects its own module to see what special handler classes (like `MP3FileInfo`) are defined. You can extend this program to handle other types of files simply by defining an appropriately−named class: `HTMLFileInfo` for HTML files, `DOCFileInfo` for Word `.doc` files, and so forth. `listDirectory` will handle them all, without modification, by handing the real work off to the appropriate classes and collating the results.

3.16. Summary

The `fileinfo.py` program should now make perfect sense.

Example 3.40. `fileinfo.py`

```
"""Framework for getting filetype−specific metadata.

Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key−value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...
```

```
Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
        try:
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]):
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
```

```
print
```

Before diving into the next chapter, make sure you're comfortable doing all of these things:

- Importing modules using either `import module` or `from module import`
- Defining and instantiating classes
- Defining `__init__` methods and other special class methods, and understanding when they are called
- Subclassing `UserDict` to define classes that act like dictionaries
- Defining data attributes and class attributes, and understanding the differences between them
- Defining private methods
- Catching exceptions with `try...except`
- Protecting external resources with `try...finally`
- Reading from files
- Assigning multiple values at once in a `for` loop
- Using the `os` module for all your cross−platform file manipulation needs
- Dynamically instantiating classes of unknown type by treating classes as objects and passing them around

---

[4] There are no constants in Python. Everything can be changed if you try hard enough. This fits with one of the core principles of Python: bad behavior should be discouraged but not banned. If you really want to change the value of `None`, you can do it, but don't come running to me when your code is impossible to debug.

[5] Strictly speaking, private methods are accessible outside their class, just not *easily* accessible. Nothing in Python is truly private; internally, the names of private methods and attributes are mangled and unmangled on the fly to make them seem inaccessible by their given names. You can access the `__parse` method of the `MP3FileInfo` class by the name `_MP3FileInfo__parse`. Acknowledge that this is interesting, then promise to never, ever do it in real code. Private methods are private for a reason, but like many other things in Python, their privateness is ultimately a matter of convention, not force.

[6] Or, as some marketroids would put it, your program would perform an illegal action. Whatever.

Chapter 4. HTML Processing

4.1. Diving in

I often see questions on comp.lang.python like "How can I list all the [headers|images|links] in my HTML document?" "How do I [parse|translate|munge] the text of my HTML document but leave the tags alone?" "How can I [add|remove|quote] attributes of all my HTML tags at once?" This chapter will answer all of these questions.

Here is a complete, working Python program in two parts. The first part, `BaseHTMLProcessor.py`, is a generic tool to help you process HTML files by walking through the tags and text blocks. The second part, `dialect.py`, is an example of how to use `BaseHTMLProcessor.py` to translate the text of an HTML document but leave the tags alone. Read the `doc string`s and comments to get an overview of what's going on. Most of it will seem like black magic, because it's not obvious how any of these class methods ever get called. Don't worry, all will be revealed in due time.

Example 4.1. `BaseHTMLProcessor.py`

If you have not already done so, you can download this and other examples used in this book.

```
from sgmllib import SGMLParser
import htmlentitydefs

class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        # extend (called by SGMLParser.__init__)
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs):
        # called for each start tag
        # attrs is a list of (attr, value) tuples
        # e.g. for <pre class="screen">, tag="pre", attrs=[("class", "screen")]
        # Ideally we would like to reconstruct original tag and attributes, but
        # we may end up quoting attribute values that weren't quoted in the source
        # document, or we may change the type of quotes around the attribute value
        # (single to double quotes).
        # Note that improperly embedded non-HTML code (like client-side Javascript)
        # may be parsed incorrectly by the ancestor, causing runtime script errors.
        # All non-HTML code must be enclosed in HTML comment tags (<!-- code -->)
        # to ensure that it will pass through this parser unaltered (in handle_comment).
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())

    def unknown_endtag(self, tag):
        # called for each end tag, e.g. for </pre>, tag will be "pre"
        # Reconstruct the original end tag.
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):
        # called for each character reference, e.g. for " ", ref will be "160"
        # Reconstruct the original character reference.
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):
        # called for each entity reference, e.g. for "&copy;", ref will be "copy"
        # Reconstruct the original entity reference.
        self.pieces.append("&%(ref)s" % locals())
```

```
        # standard HTML entities are closed with a semicolon; other entities are not
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):
        # called for each block of plain text, i.e. outside of any tag and
        # not containing any character or entity references
        # Store the original text verbatim.
        self.pieces.append(text)

    def handle_comment(self, text):
        # called for each HTML comment, e.g. <!-- insert Javascript code here -->
        # Reconstruct the original comment.
        # It is especially important that the source document enclose client-side
        # code (like Javascript) within comments so it can pass through this
        # processor undisturbed; see comments in unknown_starttag for details.
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):
        # called for each processing instruction, e.g. <?instruction>
        # Reconstruct original processing instruction.
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        # called for the DOCTYPE, if present, e.g.
        # <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        #     "http://www.w3.org/TR/html4/loose.dtd">
        # Reconstruct original DOCTYPE
        self.pieces.append("<!%(text)s>" % locals())

    def output(self):
        """Return processed HTML as a single string"""
        return "".join(self.pieces)
```

Example 4.2. `dialect.py`

```
import re
from BaseHTMLProcessor import BaseHTMLProcessor

class Dialectizer(BaseHTMLProcessor):
    subs = ()

    def reset(self):
        # extend (called from __init__ in ancestor)
        # Reset all data attributes
        self.verbatim = 0
        BaseHTMLProcessor.reset(self)

    def start_pre(self, attrs):
        # called for every <pre> tag in HTML source
        # Increment verbatim mode count, then handle tag like normal
        self.verbatim += 1
        self.unknown_starttag("pre", attrs)

    def end_pre(self):
        # called for every </pre> tag in HTML source
        # Decrement verbatim mode count
        self.unknown_endtag("pre")
        self.verbatim -= 1

    def handle_data(self, text):
```

```
        # override
        # called for every block of text in HTML source
        # If in verbatim mode, save text unaltered;
        # otherwise process the text with a series of substitutions
        self.pieces.append(self.verbatim and text or self.process(text))

    def process(self, text):
        # called from handle_data
        # Process text block by performing series of regular expression
        # substitutions (actual substitions are defined in descendant)
        for fromPattern, toPattern in self.subs:
            text = re.sub(fromPattern, toPattern, text)
        return text

class ChefDialectizer(Dialectizer):
    """convert HTML to Swedish Chef-speak

    based on the classic chef.x, copyright (c) 1992, 1993 John Hagerman
    """
    subs = ((r'a([nu])', r'u\1'),
            (r'A([nu])', r'U\1'),
            (r'a\B', r'e'),
            (r'A\B', r'E'),
            (r'en\b', r'ee'),
            (r'\Bew', r'oo'),
            (r'\Be\b', r'e-a'),
            (r'\be', r'i'),
            (r'\bE', r'I'),
            (r'\Bf', r'ff'),
            (r'\Bir', r'ur'),
            (r'(\w*?)i(\w*?)$', r'\1ee\2'),
            (r'\bow', r'oo'),
            (r'\bo', r'oo'),
            (r'\bO', r'Oo'),
            (r'the', r'zee'),
            (r'The', r'Zee'),
            (r'th\b', r't'),
            (r'\Btion', r'shun'),
            (r'\Bu', r'oo'),
            (r'\BU', r'Oo'),
            (r'v', r'f'),
            (r'V', r'F'),
            (r'w', r'w'),
            (r'W', r'W'),
            (r'([a-z])[.]', r'\1.  Bork Bork Bork!'))

class FuddDialectizer(Dialectizer):
    """convert HTML to Elmer Fudd-speak"""
    subs = ((r'[rl]', r'w'),
            (r'qu', r'qw'),
            (r'th\b', r'f'),
            (r'th', r'd'),
            (r'n[.]', r'n, uh-hah-hah-hah.'))

class OldeDialectizer(Dialectizer):
    """convert HTML to mock Middle English"""
    subs = ((r'i([bcdfghjklmnpqrstvwxyz])e\b', r'y\1'),
            (r'i([bcdfghjklmnpqrstvwxyz])e', r'y\1\1e'),
            (r'ick\b', r'yk'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'e[ea]([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'([bcdfghjklmnpqrstvwxyz])y', r'\1ee'),
            (r'([bcdfghjklmnpqrstvwxyz])er', r'\1re'),
```

```
            (r'([aeiou])re\b', r'\1r'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'i\1e'),
            (r'tion\b', r'cioun'),
            (r'ion\b', r'ioun'),
            (r'aid', r'ayde'),
            (r'ai', r'ey'),
            (r'ay\b', r'y'),
            (r'ay', r'ey'),
            (r'ant', r'aunt'),
            (r'ea', r'ee'),
            (r'oa', r'oo'),
            (r'ue', r'e'),
            (r'oe', r'o'),
            (r'ou', r'ow'),
            (r'ow', r'ou'),
            (r'\bhe', r'hi'),
            (r've\b', r'veth'),
            (r'se\b', r'e'),
            (r"'s\b", r'es'),
            (r'ic\b', r'ick'),
            (r'ics\b', r'icc'),
            (r'ical\b', r'ick'),
            (r'tle\b', r'til'),
            (r'll\b', r'l'),
            (r'ould\b', r'olde'),
            (r'own\b', r'oune'),
            (r'un\b', r'onne'),
            (r'rry\b', r'rye'),
            (r'est\b', r'este'),
            (r'pt\b', r'pte'),
            (r'th\b', r'the'),
            (r'ch\b', r'che'),
            (r'ss\b', r'sse'),
            (r'([wybdp])\b', r'\1e'),
            (r'([rnt])\b', r'\1\1e'),
            (r'from', r'fro'),
            (r'when', r'whan'))

def translate(url, dialectName="chef"):
    """fetch URL and translate using dialect

    dialect in ("chef", "fudd", "olde")"""
    import urllib
    sock = urllib.urlopen(url)
    htmlSource = sock.read()
    sock.close()
    parserName = "%sDialectizer" % dialectName.capitalize()
    parserClass = globals()[parserName]
    parser = parserClass()
    parser.feed(htmlSource)
    parser.close()
    return parser.output()

def test(url):
    """test all dialects against URL"""
    for dialect in ("chef", "fudd", "olde"):
        outfile = "%s.html" % dialect
        fsock = open(outfile, "wb")
        fsock.write(translate(url, dialect))
        fsock.close()
        import webbrowser
        webbrowser.open_new(outfile)
```

```
if __name__ == "__main__":
    test("http://diveintopython.org/odbchelper_list.html")
```

Example 4.3. Output of `dialect.py`

Running this script will translate *Introducing lists* into mock Swedish Chef−speak (from The Muppets), mock Elmer Fudd−speak (from Bugs Bunny cartoons), and mock Middle English (loosely based on Chaucer's *The Canterbury Tales*). If you look at the HTML source of the output pages, you'll see that all the HTML tags and attributes are untouched, but the text between the tags has been "translated" into the mock language. If you look closer, you'll see that, in fact, only the titles and paragraphs were translated; the code listings and screen examples were left untouched.

4.2. Introducing `sgmllib.py`

HTML processing is broken into three steps: breaking down the HTML into its constituent pieces, fiddling with the pieces, and reconstructing the pieces into HTML again. The first step is done by `sgmllib.py`, a part of the standard Python library.

The key to understanding this chapter is to realize that HTML is not just text, it is structured text. The structure is derived from the more−or−less−hierarchical sequence of start tags and end tags. Usually you don't work with HTML this way; you work with it *textually* in a text editor, or *visually* in a web browser or web authoring tool. `sgmllib.py` presents HTML *structurally*.

`sgmllib.py` contains one important class: `SGMLParser`. `SGMLParser` parses HTML into useful pieces, like start tags and end tags. As soon as it succeeds in breaking down some data into a useful piece, it calls a method on itself based on what it found. In order to use the parser, you subclass the `SGMLParser` class and override these methods. This is what I meant when I said that it presents HTML *structurally*: the structure of the HTML determines the sequence of method calls and the arguments passed to each method.

`SGMLParser` parses HTML into 8 kinds of data, and calls a separate method for each of them:

Start tag
: An HTML tag that starts a block, like `<html>`, `<head>`, `<body>`, or `<pre>`, or a standalone tag like `<br>` or `<img>`. When it finds a start tag *`tagname`*, `SGMLParser` will look for a method called `start_`*`tagname`* or `do_`*`tagname`*. For instance, when it finds a `<pre>` tag, it will look for a `start_pre` or `do_pre` method. If found, `SGMLParser` calls this method with a list of the tag's attributes; otherwise, it calls `unknown_starttag` with the tag name and list of attributes.

End tag
: An HTML tag that ends a block, like `</html>`, `</head>`, `</body>`, or `</pre>`. When it finds an end tag, `SGMLParser` will look for a method called `end_`*`tagname`*. If found, `SGMLParser` calls this method, otherwise it calls `unknown_endtag` with the tag name.

Character reference
: An escaped character referenced by its decimal or hexadecimal equivalent, like ` `. When found, `SGMLParser` calls `handle_charref` with the text of the decimal or hexidecimal character equivalent.

Entity reference
: An HTML entity, like `©`. When found, `SGMLParser` calls `handle_entityref` with the name of the HTML entity.

Comment
: An HTML comment, enclosed in `<!-- ... -->`. When found, `SGMLParser` calls `handle_comment` with the body of the comment.

Processing instruction
: An HTML processing instruction, enclosed in `<? ... >`. When found, `SGMLParser` calls `handle_pi`

with the body of the processing instruction.

Declaration

An HTML declaration, such as a `DOCTYPE`, enclosed in `<! ... >`. When found, `SGMLParser` calls `handle_decl` with the body of the declaration.

Text data

A block of text. Anything that doesn't fit into the other 7 categories. When found, `SGMLParser` calls `handle_data` with the text.

Important: Language evolution: DOCTYPE

Python 2.0 had a bug where `SGMLParser` would not recognize declarations at all (`handle_decl` would never be called), which meant that `DOCTYPE`s were silently ignored. This is fixed in Python 2.1.

`sgmllib.py` comes with a test suite to illustrate this. You can run `sgmllib.py`, passing the name of an HTML file on the command line, and it will print out the tags and other elements as it parses them. It does this by subclassing the `SGMLParser` class and defining `unknown_starttag`, `unknown_endtag`, `handle_data` and other methods which simply print their arguments.

Tip: Specifying command line arguments in Windows

In the Python IDE on Windows, you can specify command line arguments in the "Run script" dialog. Separate multiple arguments with spaces.

Example 4.4. Sample test of `sgmllib.py`

Here is a snippet from the table of contents of the HTML version of this book, `toc.html`.

```
<h1>
  <a name='c40a'></a>
  Dive Into Python
</h1>
<p class='pubdate'>
  28 Feb 2001
</p>
<p class='copyright'>
  Copyright copy 2000, 2001 by
  <a href='mailto:f8dy@diveintopython.org' title='send e-mail to the author'>
    Mark Pilgrim
  </a>
</p>
<p>
  <a name='c40ab2b4'></a>
  <b></b>
</p>
<p>
  This book lives at
  <a href='http://diveintopython.org/'>
    http://diveintopython.org/
  </a>
  .
  If you're reading it somewhere else, you may not have the latest version.
</p>
```

Running this through the test suite of `sgmllib.py` yields this output:

```
start tag: <h1>
start tag: <a name="c40a" >
```

```
end tag: </a>
data: 'Dive Into Python'
end tag: </h1>
start tag: <p class="pubdate" >
data: '28 Feb 2001'
end tag: </p>
start tag: <p class="copyright" >
data: 'Copyright '
*** unknown entity ref: &copy;
data: ' 2000, 2001 by '
start tag: <a href="mailto:f8dy@diveintopython.org" title="send e-mail to the author" >
data: 'Mark Pilgrim'
end tag: </a>
end tag: </p>
start tag: <p>
start tag: <a name="c40ab2b4" >
end tag: </a>
start tag: <b>
end tag: </b>
end tag: </p>
start tag: <p>
data: 'This book lives at '
start tag: <a href="http://diveintopython.org/" >
data: 'http://diveintopython.org/'
end tag: </a>
data: ".\012If you're reading it somewhere else, you may not have the lates"
data: 't version.\012'
end tag: </p>
```

Here's the roadmap for the rest of the chapter:

- Subclass `SGMLParser` to create classes that extract interesting data out of HTML documents.
- Subclass `SGMLParser` to create `BaseHTMLProcessor`, which overrides all 8 handler methods and uses them to reconstruct the original HTML from the pieces.
- Subclass `BaseHTMLProcessor` to create `Dialectizer`, which adds some methods to process specific HTML tags specially, and overrides the `handle_data` method to provide a framework for processing the text blocks between the HTML tags.
- Subclass `Dialectizer` to create classes that define text processing rules used by `Dialectizer.handle_data`.
- Write a test suite that grabs a real web page from `http://diveintopython.org/` and processes it.

4.3. Extracting data from HTML documents

To extract data from HTML documents, subclass the `SGMLParser` class and define methods for each tag or entity you want to capture.

The first step to extracting data from an HTML document is getting some HTML. If you have some HTML lying around on your hard drive, you can use file functions to read it, but the real fun begins when you get HTML from live web pages.

Example 4.5. Introducing `urllib`

```
>>> import urllib                                         (1)
>>> sock = urllib.urlopen("http://diveintopython.org/") (2)
>>> htmlSource = sock.read()                              (3)
>>> sock.close()                                          (4)
>>> print htmlSource                                      (5)
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
      <meta http-equiv='Content-Type' content='text/html; charset=ISO-8859-1'>
   <title>Dive Into Python</title>
<link rel='stylesheet' href='diveintopython.css' type='text/css'>
<link rev='made' href='mailto:f8dy@diveintopython.org'>
<meta name='keywords' content='Python, Dive Into Python, tutorial, object-oriented, programming, docu
<meta name='description' content='a free Python tutorial for experienced programmers'>
</head>
<body bgcolor='white' text='black' link='#0000FF' vlink='#840084' alink='#0000FF'>
<table cellpadding='0' cellspacing='0' border='0' width='100%'>
<tr><td class='header' width='1%' valign='top'>diveintopython.org</td>
<td width='99%' align='right'><hr size='1' noshade></td></tr>
<tr><td class='tagline' colspan='2'>Python for experienced programmers</td></tr>

[...snip...]
```

(1) The `urllib` module is part of the standard Python library. It contains functions for getting information about and actually retrieving data from Internet−based URLs (mainly web pages).

(2) The simplest use of `urllib` is to retrieve the entire text of a web page using the `urlopen` function. Opening a URL is similar to opening a file. The return value of `urlopen` is a file−like object, which has some of the same methods as a file object.

(3) The simplest thing to do with the file−like object returned by `urlopen` is `read`, which reads the entire HTML of the web page into a single string. The object also supports `readlines`, which reads the text line by line into a list.

(4) When you're done with the object, make sure to `close` it, just like a normal file object.

(5) We now have the complete HTML of the home page of `http://diveintopython.org/` in a string, and we're ready to parse it.

Example 4.6. Introducing `urllister.py`

If you have not already done so, you can download this and other examples used in this book.

```
from sgmllib import SGMLParser

class URLLister(SGMLParser):
    def reset(self):                              (1)
        SGMLParser.reset(self)
        self.urls = []

    def start_a(self, attrs):                     (2)
        href = [v for k, v in attrs if k=='href'] (3) (4)
        if href:
            self.urls.extend(href)
```

(1) `reset` is called by the `__init__` method of `SGMLParser`, and it can also be called manually once an instance of the parser has been created. So if you need to do any initialization, do it in `reset`, not in `__init__`, so that it will be re−initialized properly when someone re−uses a parser instance.

(2) `start_a` is called by `SGMLParser` whenever it finds an `<a>` tag. The tag may contain an `href` attribute, and/or other attributes, like `name` or `title`. The `attrs` parameter is a list of tuples, `[(attribute, value), (attribute, value), ...]`. Or it may be just an `<a>`, a valid (if useless) HTML tag, in which case `attrs` would be an empty list.

(3) We can find out whether this `<a>` tag has an `href` attribute with a simple multi−variable list comprehension.

(4) String comparisons like `k=='href'` are always case−sensitive, but that's safe in this case, because `SGMLParser` converts attribute names to lowercase while building `attrs`.

Example 4.7. Using `urllister.py`

```
>>> import urllib, urllister
>>> usock = urllib.urlopen("http://diveintopython.org/")
>>> parser = urllister.URLLister()
>>> parser.feed(usock.read())        (1)
>>> usock.close()                    (2)
>>> parser.close()                   (3)
>>> for url in parser.urls: print url (4)
toc.html
#download
toc.html
history.html
download/dip_pdf.zip
download/dip_pdf.tgz
download/dip_pdf.hqx
download/diveintopython.pdf
download/diveintopython.zip
download/diveintopython.tgz
download/diveintopython.hqx

[...snip...]
```

(1) Call the `feed` method, defined in `SGMLParser`, to get HTML into the parser.[7] It takes a string, which is what `usock.read()` returns.

(2) Like files, you should `close` your URL objects as soon as you're done with them.

(3) You should `close` your parser object, too, but for a different reason. The `feed` method isn't guaranteed to process all the HTML you give it; it may buffer it, waiting for more. Once there isn't any more, call `close` to flush the buffer and force everything to be fully parsed.

(4) Once the parser is `close`d, the parsing is complete, and `parser.urls` contains a list of all the linked URLs in the HTML document.

4.4. Introducing `BaseHTMLProcessor.py`

`SGMLParser` doesn't produce anything by itself. It parses and parses and parses, and it calls a method for each interesting thing it finds, but the methods don't do anything. `SGMLParser` is an HTML *consumer*: it takes HTML and breaks it down into small, structured pieces. As you saw in the previous section, you can subclass `SGMLParser` to define classes that catch specific tags and produce useful things, like a list of all the links on a web page. Now we'll take this one step further by defining a class that catches everything `SGMLParser` throws at it and reconstructs the complete HTML document. In technical terms, this class will be an HTML *producer*.

`BaseHTMLProcessor` subclasses `SGMLParser` and provides all 8 essential handler methods: `unknown_starttag`, `unknown_endtag`, `handle_charref`, `handle_entityref`, `handle_comment`, `handle_pi`, `handle_decl`, and `handle_data`.

Example 4.8. Introducing `BaseHTMLProcessor`

```
class BaseHTMLProcessor(SGMLParser):
    def reset(self):                       (1)
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs): (2)
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
```

```
    def unknown_endtag(self, tag):          (3)
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):          (4)
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):        (5)
        self.pieces.append("&%(ref)s" % locals())
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):            (6)
        self.pieces.append(text)

    def handle_comment(self, text):         (7)
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):              (8)
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        self.pieces.append("<!%(text)s>" % locals())
```

(1) `reset`, called by `SGMLParser.__init__`, initializes `self.pieces` as an empty list before calling the ancestor method. `self.pieces` is a data attribute which will hold the pieces of the HTML document we're constructing. Each handler method will reconstruct the HTML that `SGMLParser` parsed, and each method will append that string to `self.pieces`. Note that `self.pieces` is a list. You might be tempted to define it as a string and just keep appending each piece to it. That would work, but Python is much more efficient at dealing with lists.[8]

(2) Since `BaseHTMLProcessor` does not define any methods for specific tags (like the `start_a` method in `URLLister`), `SGMLParser` will call `unknown_starttag` for every start tag. This method takes the tag (`tag`) and the list of attribute name/value pairs (`attrs`), reconstructs the original HTML, and appends it to `self.pieces`. The string formatting here is a little strange; we'll untangle that in the next section.

(3) Reconstructing end tags is much simpler; just take the tag name and wrap it in the `</...>` brackets.

(4) When `SGMLParser` finds a character reference, it calls `handle_charref` with the bare reference. If the HTML document contains the reference ` `, `ref` will be `160`. Reconstructing the original complete character reference just involves wrapping `ref` in `&#...;` characters.

(5) Entity references are similar to character references, but without the hash mark. Reconstructing the original entity reference requires wrapping `ref` in `&...;` characters. (Actually, as an erudite reader pointed out to me, it's slightly more complicated than this. Only certain standard HTML entites end in a semicolon; other similar−looking entities do not. Luckily for us, the set of standard HTML entities is defined in a dictionary in a Python module called `htmlentitydefs`. Hence the extra `if` statement.)

(6) Blocks of text are simply appended to `self.pieces` unaltered.

(7) HTML comments are wrapped in `<!--...-->` characters.

(8) Processing instructions are wrapped in `<?...>` characters.

Important: Processing HTML with embedded script
The HTML specification requires that all non−HTML (like client−side JavaScript) must be enclosed in HTML comments, but not all web pages do this properly (and all modern web browsers are forgiving if they don't). `BaseHTMLProcessor` is not forgiving; if script is improperly embedded, it will be parsed as if it were HTML. For instance, if the script contains less−than and equals signs, `SGMLParser` may incorrectly think that it has found tags and attributes. `SGMLParser` always converts tags and attribute names to lowercase, which may break the script, and `BaseHTMLProcessor` always encloses attribute values in double quotes (even if the original HTML document used single quotes or no quotes), which will certainly break the script. Always

protect your client−side script within HTML comments.

Example 4.9. `BaseHTMLProcessor` output

```
    def output(self):               (1)
        """Return processed HTML as a single string"""
        return "".join(self.pieces) (2)
```

(1) This is the one method in `BaseHTMLProcessor` that is never called by the ancestor `SGMLParser`. Since the other handler methods store their reconstructed HTML in `self.pieces`, this function is needed to join all those pieces into one string. As noted before, Python is great at lists and mediocre at strings, so we only create the complete string when somebody explicitly asks for it.

(2) If you prefer, you could use the `join` method of the `string` module instead: `string.join(self.pieces, "")`

Further reading

- W3C discusses character and entity references.
- *Python Library Reference* confirms your suspicions that the `htmlentitydefs` module is exactly what it sounds like.

4.5. `locals` and `globals`

Python has two built−in functions, `locals` and `globals`, which provide dictionary−based access to local and global variables.

First, a word on namespaces. This is dry stuff, but it's important, so pay attention. Python uses what are called namespaces to keep track of variables. A namespace is just like a dictionary where the keys are names of variables and the dictionary values are the values of those variables. In fact, you can access a namespace as a Python dictionary, as we'll see in a minute.

At any particular point in a Python program, there are several namespaces available. Each function has its own namespace, called the local namespace, which keeps track of the function's variables, including function arguments and locally defined variables. Each module has its own namespace, called the global namespace, which keeps track of the module's variables, including functions, classes, any other imported modules, and module−level variables and constants. And there is the built−in namespace, accessible from any module, which holds built−in functions and exceptions.

When a line of code asks for the value of a variable `x`, Python will search for that variable in all the available namespaces, in order:

1. local namespace − specific to the current function or class method. If the function defines a local variable `x`, or has an argument `x`, Python will use this and stop searching.
2. global namespace − specific to the current module. If the module has defined a variable, function, or class called `x`, Python will use that and stop searching.
3. built−in namespace − global to all modules. As a last resort, Python will assume that `x` is the name of built−in function or variable.

If Python doesn't find `x` in any of these namespaces, it gives up and raises a `NameError` with the message `There is no variable named 'x'`, which you saw all the way back in chapter 1, but you didn't appreciate how much work Python was doing before giving you that error.

Important: Language evolution: nested scopes
Python 2.2 introduced a subtle but important change that affects the namespace search order: nested scopes. In versions of Python prior to 2.2, when you reference a variable within a nested function or `lambda` function, Python will search for that variable in the current (nested or `lambda`) function's namespace, then in the module's namespace. Python 2.2 will search for the variable in the current (nested or `lambda`) function's namespace, *then in the parent function's namespace*, then in the module's namespace. Python 2.1 can work either way; by default, it works like Python 2.0, but you can add the following line of code at the top of your module to make your module work like Python 2.2:

```
from __future__ import nested_scopes
```

Like many things in Python, namespaces are directly accessible at run−time. Specifically, the local namespace is accessible via the built−in `locals` function, and the global (module level) namespace is accessible via the built−in `globals` function.

Example 4.10. Introducing `locals`

```
>>> def foo(arg): (1)
...     x = 1
...     print locals()
...
>>> foo(7)        (2)
{'arg': 7, 'x': 1}
>>> foo('bar')    (3)
{'arg': 'bar', 'x': 1}
```

(1) The function `foo` has two variables in its local namespace: `arg`, whose value is passed in to the function, and `x`, which is defined within the function.

(2) `locals` returns a dictionary of name/value pairs. The keys of this dictionary are the names of the variables as strings; the values of the dictionary are the actual values of the variables. So calling `foo` with `7` prints the dictionary containing the function's two local variables: `arg` (`7`) and `x` (`1`).

(3) Remember, Python has dynamic typing, so you could just as easily pass a string in for `arg`; the function (and the call to `locals`) would still work just as well. `locals` works with all variables of all datatypes.

What `locals` does for the local (function) namespace, `globals` does for the global (module) namespace. `globals` is more exciting, though, because a module's namespace is more exciting.[9] Not only does the module's namespace include module−level variables and constants, it includes all the functions and classes defined in the module. Plus, it includes anything that was imported into the module.

Remember the difference between `from` *`module`* `import` and `import` *`module`*? With `import` *`module`*, the module itself is imported, but it retains its own namespace, which is why you have to use the module name to access any of its functions or attributes: *`module.function`*. But with `from` *`module`* `import`, you're actually importing specific functions and attributes from another module into your own namespace, which is why you access them directly without referencing the original module they came from. With the `globals` function, you can actually see this happen.

Example 4.11. Introducing `globals`

Add the following block to `BaseHTMLProcessor.py`:

```
if __name__ == "__main__":
    for k, v in globals().items():             (1)
```

```
        print k, "=", v
```

(1) Just so you don't get intimidated, remember that you've seen all this before. The `globals` function returns a dictionary, and we're iterating through the dictionary using the `items` method and multi−variable assignment. The only thing new here is the `globals` function.

Now running the script from the command line gives this output:

```
c:\docbook\dip\py>python BaseHTMLProcessor.py

SGMLParser = sgmllib.SGMLParser                (1)
htmlentitydefs = <module 'htmlentitydefs' from 'C:\Python21\lib\htmlentitydefs.py'> (2)
BaseHTMLProcessor = __main__.BaseHTMLProcessor (3)
__name__ = __main__                            (4)
[...snip...]
```

(1) `SGMLParser` was imported from `sgmllib`, using `from *module* import`. That means that it was imported directly into our module's namespace, and here it is.

(2) Contrast this with `htmlentitydefs`, which was imported using `import`. That means that the `htmlentitydefs` module itself is in our namespace, but the `entitydefs` variable defined within `htmlentitydefs` is not.

(3) This module only defines one class, `BaseHTMLProcessor`, and here it is. Note that the value here is the class itself, not a specific instance of the class.

(4) Remember the `if __name__` trick? When running a module (as opposed to importing it from another module), the built−in `__name__` attribute is a special value, `__main__`. Since we ran this module as a script from the command line, `__name__` is `__main__`, which is why our little test code to print the `globals` got executed.

> **Note: Accessing variables dynamically**
> Using the `locals` and `globals` functions, you can get the value of arbitrary variables dynamically, providing the variable name as a string. This mirrors the functionality of the `getattr` function, which allows you to access arbitrary functions dynamically by providing the function name as a string.

There is one other important difference between `locals` and `globals`, which you should learn now before it bites you. It will bite you anyway, but at least then you'll remember learning it.

Example 4.12. `locals` is read−only, `globals` is not

```
def foo(arg):
    x = 1
    print locals()    (1)
    locals()["x"] = 2 (2)
    print "x=",x      (3)

z = 7
print "z=",z
foo(3)
globals()["z"] = 8    (4)
print "z=",z          (5)
```

(1) Since `foo` is called with `3`, this will print `{'arg': 3, 'x': 1}`. This should not be a surprise.

(2) You might think that this would change the value of the local variable `x` to `2`, but it doesn't. `locals` does not actually return the local namespace, it returns a copy. So changing it does nothing to the value of the variables in the local namespace.

(3) This prints `x= 1`, not `x= 2`.

(4) After being burned by `locals`, you might think that this *wouldn't* change the value of `z`, but it does. Due to internal differences in how Python is implemented (which I'd rather not go into, since I don't fully understand them myself), `globals` returns the actual global namespace, not a copy: the exact opposite behavior of `locals`. So any changes to the dictionary returned by `globals` directly affect your global variables.

(5) This prints `z= 8`, not `z= 7`.

4.6. Dictionary−based string formatting

String formatting provides an easy way to insert values into strings. Values are listed in a tuple and inserted in order into the string in place of each formatting marker. While this is efficient, it is not always the easiest code to read, especially when multiple values are being inserted. You can't simply scan through the string in one pass and understand what the result will be; you're constantly switching between reading the string and reading the tuple of values.

There is an alternative form of string formatting that uses dictionaries instead of tuples of values.

Example 4.13. Introducing dictionary−based string formatting

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> "%(pwd)s" % params                                    (1)
'secret'
>>> "%(pwd)s is not a good password for %(uid)s" % params (2)
'secret is not a good password for sa'
>>> "%(database)s of mind, %(database)s of body" % params (3)
'master of mind, master of body'
```

(1) Instead of a tuple of explicit values, this form of string formatting uses a dictionary, `params`. And instead of a simple `%s` marker in the string, the marker contains a name in parentheses. This name is used as a key in the `params` dictionary and subsitutes the corresponding value, `secret`, in place of the `%(pwd)s` marker.

(2) Dictionary−based string formatting works with any number of named keys. Each key must exist in the given dictionary, or the formatting will fail with a `KeyError`.

(3) You can even specify the same key twice; each occurrence will be replaced with the same value.

So why would you use dictionary−based string formatting? Well, it does seem like overkill to set up a dictionary of keys and values simply to do string formatting in the next line; it's really most useful when you happen to have a dictionary of meaningful keys and values already. Like `locals`.

Example 4.14. Dictionary−based string formatting in `BaseHTMLProcessor.py`

```
    def handle_comment(self, text):
        self.pieces.append("<!--%(text)s-->" % locals()) (1)
```

(1) Using the built−in `locals` function is the most common use of dictionary−based string formatting. It means that you can use the names of local variables within your string (in this case, `text`, which was passed to the class method as an argument) and each named variable will be replaced by its value. If `text` is `'Begin page footer'`, the string formatting `"<!--%(text)s-->" % locals()` will resolve to the string `'<!--Begin page footer-->'`.

```
    def unknown_starttag(self, tag, attrs):
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs]) (1)
```

```
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())                                   (2)
```

(1) When this method is called, `attrs` is a list of key/value tuples, just like the `items` of a dictionary, which means we can use multi−variable assignment to iterate through it. This should be a familiar pattern by now, but there's a lot going on here, so let's break it down:

1. Suppose `attrs` is `[('href', 'index.html'), ('title', 'Go to home page')]`.
2. In the first round of the list comprehension, `key` will get `'href'`, and `value` will get `'index.html'`.
3. The string formatting `' %s="%s"' % (key, value)` will resolve to `' href="index.html"'`. This string becomes the first element of the list comprehension's return value.
4. In the second round, `key` will get `'title'`, and `value` will get `'Go to home page'`.
5. The string formatting will resolve to `' title="Go to home page"'`.
6. The list comprehension returns a list of these two resolved strings, and `strattrs` will join both elements of this list together to form `' href="index.html" title="Go to home page"'`.

(2) Now, using dictionary−based string formatting, we insert the value of `tag` and `strattrs` into a string. So if `tag` is `'a'`, the final result would be `'<a href="index.html" title="Go to home page">'`, and that is what gets appended to `self.pieces`.

Important: Performance issues with locals
Using dictionary−based string formatting with `locals` is a convenient way of making complex string formatting expressions more readable, but it comes with a price. There is a slight performance hit in making the call to `locals`, since `locals` builds a copy of the local namespace.

4.7. Quoting attribute values

A common question on comp.lang.python is "I have a bunch of HTML documents with unquoted attribute values, and I want to properly quote them all. How can I do this?"[10] (This is generally precipitated by a project manager who has found the HTML−is−a−standard religion joining a large project and proclaiming that all pages must validate against an HTML validator. Unquoted attribute values are a common violation of the HTML standard.) Whatever the reason, unquoted attribute values are easy to fix by feeding HTML through `BaseHTMLProcessor`.

`BaseHTMLProcessor` consumes HTML (since it's descended from `SGMLParser`) and produces equivalent HTML, but the HTML output is not identical to the input. Tags and attribute names will end up in lowercase, even if they started in uppercase or mixed case, and attribute values will be enclosed in double quotes, even if they started in single quotes or with no quotes at all. It is this last side effect that we can take advantage of.

Example 4.15. Quoting attribute values

```
>>> htmlSource = """        (1)
...     <html>
...     <head>
...     <title>Test page</title>
...     </head>
...     <body>
...     <ul>
...     <li><a href=index.html>Home</a></li>
...     <li><a href=toc.html>Table of contents</a></li>
...     <li><a href=history.html>Revision history</a></li>
...     </body>
...     </html>
...     """
>>> from BaseHTMLProcessor import BaseHTMLProcessor
```

```
>>> parser = BaseHTMLProcessor()
>>> parser.feed(htmlSource) (2)
>>> print parser.output()   (3)
<html>
<head>
<title>Test page</title>
</head>
<body>
<ul>
<li><a href="index.html">Home</a></li>
<li><a href="toc.html">Table of contents</a></li>
<li><a href="history.html">Revision history</a></li>
</body>
</html>
```

(1) Note that the attribute values of the `href` attributes in the `<a>` tags are not properly quoted. (Also note that we're using triple quotes for something other than a `doc string`. And directly in the IDE, no less. They're very useful.)

(2) Feed the parser.

(3) Using the `output` function defined in `BaseHTMLProcessor`, we get the output as a single string, complete with quoted attribute values. While this may seem anti−climactic, think about how much has actually happened here: `SGMLParser` parsed the entire HTML document, breaking it down into tags, refs, data, and so forth; `BaseHTMLProcessor` used those elements to reconstruct pieces of HTML (which are still stored in `parser.pieces`, if you want to see them); finally, we called `parser.output`, which joined all the pieces of HTML into one string.

4.8. Introducing `dialect.py`

`Dialectizer` is a simple (and silly) descendant of `BaseHTMLProcessor`. It runs blocks of text through a series of substitutions, but it makes sure that anything within a `<pre>...</pre>` block passes through unaltered.

To handle the `<pre>` blocks, we define two methods in `Dialectizer`: `start_pre` and `end_pre`.

Example 4.16. Handling specific tags

```
    def start_pre(self, attrs):             (1)
        self.verbatim += 1                  (2)
        self.unknown_starttag("pre", attrs) (3)

    def end_pre(self):                      (4)
        self.unknown_endtag("pre")          (5)
        self.verbatim -= 1                  (6)
```

(1) `start_pre` is called every time `SGMLParser` finds a `<pre>` tag in the HTML source. (In a minute, we'll see exactly how this happens.) The method takes a single parameter, `attrs`, which contains the attributes of the tag (if any). `attrs` is a list of key/value tuples, just like `unknown_starttag` takes.

(2) In the `reset` method, we initialize a data attribute that serves as a counter for `<pre>` tags. Every time we hit a `<pre>` tag, we increment the counter; every time we hit a `</pre>` tag, we'll decrement the counter. (We could just use this as a flag and set it to `1` and reset it to `0`, but it's just as easy to do it this way, and this handles the odd (but possible) case of nested `<pre>` tags.) In a minute, we'll see how this counter is put to good use.

(3) That's it, that's the only special processing we do for `<pre>` tags. Now we pass the list of attributes along to `unknown_starttag` so it can do the default processing.

(4) `end_pre` is called every time `SGMLParser` finds a `</pre>` tag. Since end tags can not contain attributes, the method takes no parameters.

(5) First, we want to do the default processing, just like any other end tag.

(6) Second, we decrement our counter to signal that this `<pre>` block has been closed.

At this point, it's worth digging a little further into `SGMLParser`. I've claimed repeatedly (and you've taken it on faith so far) that `SGMLParser` looks for and calls specific methods for each tag, if they exist. For instance, we just saw the definition of `start_pre` and `end_pre` to handle `<pre>` and `</pre>`. But how does this happen? Well, it's not magic, it's just good Python coding.

Example 4.17. `SGMLParser`

```
    def finish_starttag(self, tag, attrs):               (1)
        try:
            method = getattr(self, 'start_' + tag)       (2)
        except AttributeError:                           (3)
            try:
                method = getattr(self, 'do_' + tag)      (4)
            except AttributeError:
                self.unknown_starttag(tag, attrs)        (5)
                return -1
            else:
                self.handle_starttag(tag, method, attrs) (6)
                return 0
        else:
            self.stack.append(tag)
            self.handle_starttag(tag, method, attrs)
            return 1                                     (7)

    def handle_starttag(self, tag, method, attrs):
        method(attrs)                                    (8)
```

(1) At this point, `SGMLParser` has already found a start tag and parsed the attribute list. The only thing left to do is figure out whether there is a specific handler method for this tag, or whether we should fall back on the default method (`unknown_starttag`).

(2) The "magic" of `SGMLParser` is nothing more than our old friend, `getattr`. What you may not have realized before is that `getattr` will find methods defined in descendants of an object as well as the object itself. Here the object is `self`, the current instance. So if `tag` is `'pre'`, this call to `getattr` will look for a `start_pre` method on the current instance, which is an instance of the `Dialectizer` class.

(3) `getattr` raises an `AttributeError` if the method it's looking for doesn't exist in the object (or any of its descendants), but that's okay, because we wrapped the call to `getattr` inside a `try...except` block and explicitly caught the `AttributeError`.

(4) Since we didn't find a `start_xxx` method, we'll also look for a `do_xxx` method before giving up. This alternate naming scheme is generally used for standalone tags, like `<br>`, which have no corresponding end tag. But you can use either naming scheme; as you can see, `SGMLParser` tries both for every tag. (You shouldn't define both a `start_xxx` and `do_xxx` handler method for the same tag, though; only the `start_xxx` method will get called.)

(5) Another `AttributeError`, which means that the call to `getattr` failed with `do_xxx`. Since we found neither a `start_xxx` nor a `do_xxx` method for this tag, we catch the exception and fall back on the default method, `unknown_starttag`.

(6) Remember, `try...except` blocks can have an `else` clause, which is called if no exception is raised during the `try...except` block. Logically, that means that we *did* find a `do_xxx` method for this tag, so we're going to call it.

(7) By the way, don't worry about these different return values; in theory they mean something, but they're never actually used. Don't worry about the `self.stack.append(tag)` either; `SGMLParser` keeps track internally of whether your start tags are balanced by appropriate end tags, but it doesn't do anything with this information either. In theory, you could use this module to validate that your tags were fully balanced, but it's probably not worth it, and it's beyond the scope of this chapter. We have better things to worry about right now.

(8) `start_xxx` and `do_xxx` methods are not called directly; the tag, method, and attributes are passed to this function, `handle_starttag`, so that descendants can override it and change the way *all* start tags are dispatched. We don't do need that level of control, so we just let this method do its thing, which is to call the method (`start_xxx` or `do_xxx`) with the list of attributes. Remember, `method` is a function, returned from `getattr`, and functions are objects. (I know you're getting tired of hearing it, and I promise I'll stop saying it as soon as we stop finding new ways of using it to our advantage.) Here, the function object is passed into this dispatch method as an argument, and this method turns around and calls the function. At this point, we don't have to know what the function is, what it's named, or where it's defined; the only thing we have to know about the function is that it is called with one argument, `attrs`.

Now back to our regularly scheduled program: `Dialectizer`. When we left, we were in the process of defining specific handler methods for `<pre>` and `</pre>` tags. There's only one thing left to do, and that is to process text blocks with our pre−defined substitutions. For that, we need to override the `handle_data` method.

Example 4.18. Overriding the `handle_data` method

```
    def handle_data(self, text):                                         (1)
        self.pieces.append(self.verbatim and text or self.process(text)) (2)
```

(1) `handle_data` is called with only one argument, the text to process.

(2) In the ancestor `BaseHTMLProcessor`, the `handle_data` method simply appended the text to the output buffer, `self.pieces`. Here the logic is only slightly more complicated. If we're in the middle of a `<pre>...</pre>` block, `self.verbatim` will be some value greater than `0`, and we want to put the text in the output buffer unaltered. Otherwise, we will call a separate method to process the substitutions, then put the result of that into the output buffer. In Python, this is a one−liner, using the `and−or` trick.

We're close to completely understanding `Dialectizer`. The only missing link is the nature of the text substitutions themselves. If you know any Perl, you know that when complex text substitutions are required, the only real solution is regular expressions.

4.9. Regular expressions 101

Regular expressions are a powerful (and fairly standardized) way of searching, replacing, and parsing text with complex patterns of characters. If you've used regular expressions in other languages (like Perl), you should skip this section and just read the summary of the `re` module to get an overview of the available functions and their arguments.

Strings have methods for searching (`index`, `find`, and `count`), replacing (`replace`), and parsing (`split`), but they are limited to the simplest of cases. The search methods look for a single, hard−coded substring, and they are always case−sensitive; to do case−insensitive searches of a string `s`, you must call `s.lower()` or `s.upper()` and make sure your search strings are the appropriate case to match. The `replace` and `split` methods have the same limitations. You should use them if you can (they're fast and easy to read), but for anything more complex, you'll have to move up to regular expressions.

Example 4.19. Matching at the end of a string

This series of examples was inspired by a real−life problem I had in my day job, scrubbing and standardizing street addresses exported from a legacy system before importing them into a newer system. (See, I don't just make this stuff up; it's actually useful.)

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')               (1)
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')               (2)
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.') (3)
'100 NORTH BROAD RD.'
>>> import re                              (4)
>>> re.sub('ROAD$', 'RD.', s)              (5) (6)
'100 NORTH BROAD RD.'
```

(1) My goal is to standardize a street address so that `'ROAD'` is always abbreviated as `'RD.'`. At first glance, I thought this was simple enough that I could just use the string method `replace`. After all, all the data was already uppercase, so case mismatches would not be a problem. And the search string, `'ROAD'`, was a constant. And in this deceptively simple example, `s.replace` does indeed work.

(2) Life, unfortunately, is full of counterexamples, and I quickly discovered this one. The problem here is that `'ROAD'` appears twice in the address, once as part of the street name `'BROAD'` and once as its own word. The `replace` method sees these two occurrences and blindly replaces both of them; meanwhile, I see my addresses getting destroyed.

(3) To solve the problem of addresses with more than one `'ROAD'` substring, we could resort to something like this: only search and replace `'ROAD'` in the last 4 characters of the address (`s[-4:]`), and leave the string along (`s[:-4]`). But you can see that this is already getting unweildy. For example, the pattern is dependent on the length of the string we're replacing (if we were replacing `'STREET'` with `'ST.'`, we would need to use `s[:-6]` and `s[-6:].replace(...)`). Would you like to come back in six months and debug this? I know I wouldn't.

(4) It's time to move up to regular expressions. In Python, all functionality related to regular expressions is contained in the `re` module.

(5) Take a look at the first parameter: `'ROAD$'`. This is a very simple regular expression which matches `'ROAD'` only when it occurs at the end of a string. The `$` means "end of the string". (There is a corresponding character, the caret `^`, which means "beginning of the string".)

(6) Using the `re.sub` function, we search the string `s` for the regular expression `'ROAD$'` and replace it with `'RD.'`. This matches the `ROAD` at the end of the string `s`, but does *not* match the `ROAD` that's part of the word `BROAD`, because that's in the middle of `s`.

Example 4.20. Matching whole words

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)     (1)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)  (2)
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)  (3)
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)  (4)
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s) (5)
'100 BROAD RD. APT 3'
```

(1) Continuing with my story of scrubbing addresses, I soon discovered that the previous example, matching `'ROAD'` at the end of the address, was not good enough, because not all addresses included a street designation at all; some just ended with the street name. Most of the time, I got away with it, but if the street name was `'BROAD'`, then the regular expression would match `'ROAD'` at the end of the string as part of the word `'BROAD'`, which is not what I wanted.

(2) What I *really* wanted was to match `'ROAD'` when it was at the end of the string *and* it was its own whole word, not a part of some larger word. To express this in a regular expression, you use `\b`, which means "a word boundary must occur right here". In Python, this is complicated by the fact that the `'\'` character in a string must itself be escaped. (This is sometimes referred to as the backslash plague, and it is one reason why regular expressions are easier in Perl than in Python. On the down side, Perl mixes regular expressions with other syntax, so if you have a bug, it may be hard to tell whether it's a bug in syntax or a bug in your regular expression.)

(3) To work around the backslash plague, you can use what is called a raw string, by prefixing the `'...'` with the letter `r`. This tells Python that nothing in this string should be escaped; `'\t'` is a tab character, but `r'\t'` is really the backslash character \ followed by the letter `t`. I recommend always using raw strings when dealing with regular expressions, otherwise things get too confusing too quickly (and regular expressions get confusing quickly enough all by themselves).

(4) **sigh** Unfortunately, I soon found more cases that contradicted my logic. In this case, the street address contained the word `'ROAD'` as a whole word by itself, but it wasn't at the end, because the address had an apartment number after the street designation. Because `'ROAD'` isn't at the very end of the string, it doesn't match, so the entire call to `re.sub` ends up replacing nothing at all, and we get the original string back, which is not what we want.

(5) To solve this problem, I removed the `$` character and added another `\b`. Now the regular expression reads "match `'ROAD'` when it's a whole word by itself anywhere in the string", whether at the end, the beginning, or somewhere in the middle.

This is just the tiniest tip of the iceberg of what regular expressions can do. They are extremely powerful, and there are entire books devoted to them. They are not the correct solution for every problem. You should learn enough about them to know when they are appropriate, and when they will simply cause more problems than they solve.

> Some people, when confronted with a problem, think "I know, I'll use regular expressions."
> Now they have two problems.
>
> −−Jamie Zawinski, in comp.lang.emacs

Further reading

- Regular Expression HOWTO teaches about regular expressions and how to use them in Python.
- *Python Library Reference* summarizes the `re` module.

4.10. Putting it all together

It's time to put everything we've learned so far to good use. I hope you were paying attention.

Example 4.21. The `translate` function, part 1

```
def translate(url, dialectName="chef"): (1)
    import urllib                       (2)
    sock = urllib.urlopen(url)          (3)
    htmlSource = sock.read()
    sock.close()
```

(1) The `translate` function has an optional argument `dialectName`, which is a string that specifies the dialect we'll be using. We'll see how this is used in a minute.

(2) Hey, wait a minute, there's an `import` statement in this function! That's perfectly legal in Python. You're used to seeing `import` statements at the top of a program, which means that the imported module is available anywhere in the program. But you can also import modules within a function, which means that the imported module is only available within the function. If you have a module that is only ever used in one function, this is an easy way to make your code more modular. (When you find that your weekend hack has turned into an 800−line work of art and decide to split it up into a dozen reusable modules, you'll appreciate this.)

(3) Now we get the source of the given URL.

Example 4.22. The `translate` function, part 2: curiouser and curiouser

```
    parserName = "%sDialectizer" % dialectName.capitalize() (1)
    parserClass = globals()[parserName]                     (2)
    parser = parserClass()                                  (3)
```

(1) `capitalize` is a string method we haven't seen before; it simply capitalizing the first letter of a string and forces everything else to lowercase. Combined with some string formatting, we've taken the name of a dialect and transformed it into the name of the corresponding Dialectizer class. If `dialectName` is the string `'chef'`, `parserName` will be the string `'ChefDialectizer'`.

(2) We have the name of a class as a string (`parserName`), and we have the global namespace as a dictionary (`globals`()). Combined, we can get a reference to the class which the string names. (Remember, classes are objects, and they can be assigned to variables just like any other object.) If `parserName` is the string `'ChefDialectizer'`, `parserClass` will be the class `ChefDialectizer`.

(3) Finally, we have a class object (`parserClass`), and we want an instance of the class. Well, we already know how to do that: call the class like a function. The fact that the class is being stored in a local variable makes absolutely no difference; we just call the local variable like a function, and out pops an instance of the class. If `parserClass` is the class `ChefDialectizer`, `parser` will be an instance of the class `ChefDialectizer`.

Why bother? After all, there are only 3 `Dialectizer` classes; why not just use a `case` statement? (Well, there's no `case` statement in Python, but why not just use a series of `if` statements?) One reason: extensibility. The `translate` function has absolutely no idea how many Dialectizer classes we've defined. Imagine if we defined a new `FooDialectizer` tomorrow; `translate` would work by passing `'foo'` as the `dialectName`.

Even better, imagine putting `FooDialectizer` in a separate module, and importing it with `from` *`module`* `import`. We've already seen that this includes it in `globals`(), so `translate` would still work without modification, even though `FooDialectizer` was in a separate file.

Now imagine that the name of the dialect is coming from somewhere outside the program, maybe from a database or from a user−inputted value on a form. You can use any number of server−side Python scripting architectures to dynamically generate web pages; this function could take a URL and a dialect name (both strings) in the query string of a web page request, and output the "translated" web page.

Finally, imagine a `Dialectizer` framework with a plug−in architecture. You could put each `Dialectizer` class in a separate file, leaving only the `translate` function in `dialect.py`. Assuming a consistent naming scheme, the `translate` function could dynamic import the appropiate class from the appropriate file, given nothing but the dialect name. (You haven't seen dynamic importing yet, but I promise to cover in a later chapter.) To add a new dialect, you would simply add an appropriately−named file in the plug−ins directory (like `foodialect.py` which contains the `FooDialectizer` class). Calling the `translate` function with the dialect name `'foo'` would find the module `foodialect.py`, import the class `FooDialectizer`, and away we go.

Example 4.23. The `translate` function, part 3

```
    parser.feed(htmlSource) (1)
    parser.close()          (2)
    return parser.output()  (3)
```

(1) After all that imagining, this is going to seem pretty boring, but the `feed` function is what does the entire transformation. We had the entire HTML source in a single string, so we only had to call `feed` once. However, you can call `feed` as often as you want, and the parser will just keep parsing. So if we were worried about memory usage (or we knew we were going to be dealing with very large HTML pages), we could set this up in a loop, where we read a few bytes of HTML and fed it to the parser. The result would be the same.

(2) Because `feed` maintains an internal buffer, you should always call the parser's `close` method when you're done (even if you fed it all at once, like we did). Otherwise you may find that your output is missing the last few bytes.

(3) Remember, `output` is the function we defined on `BaseHTMLProcessor` that joins all the pieces of output we've buffered and returns them in a single string.

And just like that, we've "translated" a web page, given nothing but a URL and the name of a dialect.

Further reading

- You thought I was kidding about the server−side scripting idea. So did I, until I found this web−based dialectizer. I have no idea if it's implemented in Python, but my company's home page is funny as hell in Pig Latin. Unfortunately, source code does not appear to be available.

4.11. Summary

Python provides you with a powerful tool, `sgmllib.py`, to manipulate HTML by turning its structure into an object model. You can use this tool in many different ways.

- parsing the HTML looking for something specific
- aggregating the results, like the URL lister
- altering the structure along the way, like the attribute quoter
- transforming the HTML into something else by manipulating the text while leaving the tags alone, like the `Dialectizer`

Along with these examples, you should be comfortable doing all of the following things:

- Using `locals`() and `globals`() to access namespaces
- Formatting strings using dictionary−based substitutions
- Using regular expressions. They are an important part of every programmer's toolkit, and they will play a larger role in future chapters.

---

[7] The technical term for a parser like `SGMLParser` is a *consumer*: it consumes HTML and breaks it down. Presumably, the name `feed` was chosen to fit into the whole "consumer" motif. Personally, it makes me think of an exhibit in the zoo where there's just a dark cage with no trees or plants or evidence of life of any kind, but if you stand perfectly still and look really closely you can make out two beady eyes staring back at you from the far left corner, but you convince yourself that that's just your mind playing tricks on you, and the only way you can tell that the whole thing isn't just an empty cage is a small innocuous sign on the railing that reads, "Do not feed the parser." But maybe that's just me. In any event, it's an interesting mental image.

[8] The reason Python is better at lists than strings is that lists are mutable but strings are immutable. This means that appending to a list just adds the element and updates the index. Since strings can not be changed after they are created, code like `s = s + newpiece` will create an entirely new string out of the concatenation of the original and the new piece, then throw away the original string. This involves a lot of expensive memory management, and the amount of effort involved increases as the string gets longer, so doing `s = s + newpiece` in a loop is deadly. In technical terms, appending `n` items to a list is $O(n)$, while appending `n` items to a string is $O(n^2)$.

[9] I don't get out much.

[10] All right, it's not that common a question. It's not up there with "What editor should I use to write Python code?" (answer: Emacs) or "Is Python better or worse than Perl?" (answer: "Perl is worse than Python because people wanted it worse." −Larry Wall, 10/14/1998) But questions about HTML processing pop up in one form or another about once a month, and among those questions, this is a popular one.

Chapter 5. XML Processing

5.1. Diving in

This chapter is about XML processing in Python. It would be helpful if you already knew what an XML document looks like, that it's made up of structured tags to form a hierarchy of elements, and so on. If this doesn't make sense to you, go read an XML tutorial first, then come back.

Being a philosophy major is not required, although if you have ever had the misfortune of being subjected to the writings of Immanuel Kant, you will appreciate the example program a lot more than if you majored in something useful, like computer science.

There are two basic ways to work with XML. One is called SAX ("Simple API for XML"), and it works by reading the XML a little bit at a time and calling a method for each element it finds. (If you read *HTML Processing*, this should sound familiar, because that's how the `sgmllib` module works.) The other is called DOM ("Document Object Model"), and it works by reading in the entire XML document at once and creating an internal representation of it using native Python classes linked in a tree structure. Python has standard modules for both kinds of parsing, but this chapter will only deal with using the DOM.

The following is a complete Python program which generates pseudo−random output based on a context−free grammar defined in an XML format. Don't worry yet if you don't understand what that means; we'll examine both the program's input and its output in more depth throughout the chapter.

Example 5.1. `kgp.py`

If you have not already done so, you can download this and other examples used in this book.

```
"""Kant Generator for Python

Generates mock philosophy based on a context−free grammar

Usage: python kgp.py [options] [source]

Options:
  −g ..., −−grammar=...   use specified grammar file or URL
  −h, −−help              show this help
  −d                      show debugging information while parsing

Examples:
  kgp.py                  generates several paragraphs of Kantian philosophy
  kgp.py −g husserl.xml   generates several paragraphs of Husserl
  kpg.py "<xref id='paragraph'/>"  generates a paragraph of Kant
  kgp.py template.xml     reads from template.xml to decide what to generate
"""
from xml.dom import minidom
import random
import toolbox
import sys
import getopt

_debug = 0

class NoSourceError(Exception): pass

class KantGenerator:
```

```
    """generates mock philosophy based on a context-free grammar"""

    def __init__(self, grammar, source=None):
        self.loadGrammar(grammar)
        self.loadSource(source and source or self.getDefaultSource())
        self.refresh()

    def _load(self, source):
        """load XML input source, return parsed XML document

        - a URL of a remote XML file ("http://diveintopython.org/kant.xml")
        - a filename of a local XML file ("~/diveintopython/common/py/kant.xml")
        - standard input ("-")
        - the actual XML document, as a string
        """
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc

    def loadGrammar(self, grammar):
        """load context-free grammar"""
        self.grammar = self._load(grammar)
        self.refs = {}
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref

    def loadSource(self, source):
        """load source"""
        self.source = self._load(source)

    def getDefaultSource(self):
        """guess default source of the current grammar

        The default source will be one of the <ref>s that is not
        cross-referenced.  This sounds complicated but it's not.
        Example: The default source for kant.xml is
        "<xref id='section'/>", because 'section' is the one <ref>
        that is not <xref>'d anywhere in the grammar.
        In most grammars, the default source will produce the
        longest (and most interesting) output.
        """
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        if not standaloneXrefs:
            raise NoSourceError, "can't guess source, and no source specified"
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)

    def reset(self):
        """reset parser"""
        self.pieces = []
        self.capitalizeNextWord = 0

    def refresh(self):
        """reset output buffer, re-parse entire source file, and return output

        Since parsing involves a good deal of randomness, this is an
        easy way to get new output without having to reload a grammar file
        each time.
        """
```

```
        self.reset()
        self.parse(self.source)
        return self.output()

    def output(self):
        """output generated text"""
        return "".join(self.pieces)

    def randomChildElement(self, node):
        """choose a random child element of a node

        This is a utility method used by do_xref and do_choice.
        """
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE]
        chosen = random.choice(choices)
        if _debug:
            sys.stderr.write('%s available choices: %s\n' % \
                (len(choices), [e.toxml() for e in choices]))
            sys.stderr.write('Chosen: %s\n' % chosen.toxml())
        return chosen

    def parse(self, node):
        """parse a single XML node

        A parsed XML document (from minidom.parse) is a tree of nodes
        of various types.  Each node is represented by an instance of the
        corresponding Python class (Element for a tag, Text for
        text data, Document for the top-level document).  The following
        statement constructs the name of a class method based on the type
        of node we're parsing ("parse_Element" for an Element node,
        "parse_Text" for a Text node, etc.) and then calls the method.
        """
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__)
        parseMethod(node)

    def parse_Document(self, node):
        """parse the document node

        The document node by itself isn't interesting (to us), but
        its only child, node.documentElement, is: it's the root node
        of the grammar.
        """
        self.parse(node.documentElement)

    def parse_Text(self, node):
        """parse a text node

        The text of a text node is usually added to the output buffer
        verbatim.  The one exception is that <p class='sentence'> sets
        a flag to capitalize the first letter of the next word.  If
        that flag is set, we capitalize the text and reset the flag.
        """
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)

    def parse_Element(self, node):
        """parse an element
```

```
        An XML element corresponds to an actual tag in the source:
        <xref id='...'>, <p chance='...'>, <choice>, etc.
        Each element type is handled in its own method.  Like we did in
        parse(), we construct a method name based on the name of the
        element ("do_xref" for an <xref> tag, etc.) and
        call the method.
        """
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)

    def parse_Comment(self, node):
        """parse a comment

        The grammar can contain XML comments, but we ignore them
        """
        pass

    def do_xref(self, node):
        """handle <xref id='...'> tag

        An <xref id='...'> tag is a cross-reference to a <ref id='...'>
        tag.  <xref id='sentence'/> evaluates to a randomly chosen child of
        <ref id='sentence'>.
        """
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))

    def do_p(self, node):
        """handle <p> tag

        The <p> tag is the core of the grammar.  It can contain almost
        anything: freeform text, <choice> tags, <xref> tags, even other
        <p> tags.  If a "class='sentence'" attribute is found, a flag
        is set and the next word will be capitalized.  If a "chance='X'"
        attribute is found, there is an X% chance that the tag will be
        evaluated (and therefore a (100-X)% chance that it will be
        completely ignored)
        """
        keys = node.attributes.keys()
        if "class" in keys:
            if node.attributes["class"].value == "sentence":
                self.capitalizeNextWord = 1
        if "chance" in keys:
            chance = int(node.attributes["chance"].value)
            doit = (chance > random.randrange(100))
        else:
            doit = 1
        if doit:
            for child in node.childNodes: self.parse(child)

    def do_choice(self, node):
        """handle <choice> tag

        A <choice> tag contains one or more <p> tags.  One <p> tag
        is chosen at random and evaluated; the rest are ignored.
        """
        self.parse(self.randomChildElement(node))

def usage():
    print __doc__

def main(argv):
```

```
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage()
            sys.exit()
        elif opt == '-d':
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):
            grammar = arg

    source = "".join(args)

    k = KantGenerator(grammar, source)
    print k.output()

if __name__ == "__main__":
    main(sys.argv[1:])
```

Example 5.2. `toolbox.py`

```
"""Miscellaneous utility functions"""

def openAnything(source):
    """URI, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner.  Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.

    Examples:
    >>> from xml.dom import minidom
    >>> sock = openAnything("http://localhost/kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("c:\\inetpub\\wwwroot\\kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("<ref id='conjunction'><text>and</text><text>or</text></ref>")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    """
    if hasattr(source, "read"):
        return source

    if source == '-':
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)
    except (IOError, OSError):
```

```
        pass

    # try to open with native open function (if source is pathname)
    try:
        return open(source)
    except (IOError, OSError):
        pass

    # treat source as string
    return StringIO.StringIO(str(source))
```

Run the program `kgp.py` by itself, and it will parse the default XML−based grammar, in `kant.xml`, and print several paragraphs worth of philosophy in the style of Immanuel Kant.

Example 5.3. Sample output of `kgp.py`

```
[f8dy@oliver kgp]$ python kgp.py
     As is shown in the writings of Hume, our a priori concepts, in
reference to ends, abstract from all content of knowledge; in the study
of space, the discipline of human reason, in accordance with the
principles of philosophy, is the clue to the discovery of the
Transcendental Deduction.  The transcendental aesthetic, in all
theoretical sciences, occupies part of the sphere of human reason
concerning the existence of our ideas in general; still, the
never-ending regress in the series of empirical conditions constitutes
the whole content for the transcendental unity of apperception.  What
we have alone been able to show is that, even as this relates to the
architectonic of human reason, the Ideal may not contradict itself, but
it is still possible that it may be in contradictions with the
employment of the pure employment of our hypothetical judgements, but
natural causes (and I assert that this is the case) prove the validity
of the discipline of pure reason.  As we have already seen, time (and
it is obvious that this is true) proves the validity of time, and the
architectonic of human reason, in the full sense of these terms,
abstracts from all content of knowledge.  I assert, in the case of the
discipline of practical reason, that the Antinomies are just as
necessary as natural causes, since knowledge of the phenomena is a
posteriori.
    The discipline of human reason, as I have elsewhere shown, is by
its very nature contradictory, but our ideas exclude the possibility of
the Antinomies.  We can deduce that, on the contrary, the pure
employment of philosophy, on the contrary, is by its very nature
contradictory, but our sense perceptions are a representation of, in
the case of space, metaphysics.  The thing in itself is a
representation of philosophy.  Applied logic is the clue to the
discovery of natural causes.  However, what we have alone been able to
show is that our ideas, in other words, should only be used as a canon
for the Ideal, because of our necessary ignorance of the conditions.

[...snip...]
```

This is, of course, complete gibberish. Well, not complete gibberish. It is syntactically and grammatically correct (although very verbose −− Kant wasn't what you would call a get−to−the−point kind of guy). Some of it may actually be true (or at least the sort of thing that Kant would have agreed with), some of it is blatantly false, and most of it is simply incoherent. But all of it is in the style of Immanuel Kant.

Let me repeat that this is much, much funnier if you are now or have ever been a philosophy major.

The interesting thing about this program is that there is nothing Kant−specific about it. All the content in the previous example was derived from the grammar file, `kant.xml`. If we tell the program to use a different grammar file (which we can specify on the command line), the output will be completely different.

Example 5.4. Simpler output from `kgp.py`

```
[f8dy@oliver kgp]$ python kgp.py −g binary.xml
00101001
[f8dy@oliver kgp]$ python kgp.py −g binary.xml
10110100
```

We will take a closer look at the structure of the grammar file later in this chapter. For now, all you have to know is that the grammar file defines the structure of the output, and the `kgp.py` program reads through the grammar and makes random decisions about which words to plug in where.

5.2. Packages

Actually parsing an XML document is very simple: one line of code. However, before we get to that line of code, we need to take a short detour to talk about packages.

Example 5.5. Loading an XML document (a sneak peek)

```
>>> from xml.dom import minidom (1)
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')
```

(1) This is a syntax we haven't seen before. It looks almost like the `from` *`module`* `import` we know and love, but the `"."` gives it away as something above and beyond a simple import. In fact, `xml` is what is known as a package, `dom` is a nested package within `xml`, and `minidom` is a module within `xml.dom`.

That sounds complicated, but it's really not. Looking at the actual implementation may help. Packages are little more than directories of modules; nested packages are subdirectories. The modules within a package (or a nested package) are still just `.py` files, like always, except that they're in a subdirectory instead of the main `lib/` directory of your Python installation.

Example 5.6. File layout of a package

```
Python21/           root Python installation (home of the executable)
|
+−−lib/             library directory (home of the standard library modules)
   |
   +−− xml/         xml package (really just a directory with other stuff in it)
       |
       +−−sax/      xml.sax package (again, just a directory)
       |
       +−−dom/      xml.dom package (contains minidom.py)
       |
       +−−parsers/  xml.parsers package (used internally)
```

So when we say `from xml.dom import minidom`, Python figures out that that means "look in the `xml` directory for a `dom` directory, and look in *that* for the `minidom` module, and import it as `minidom`". But Python is even smarter than that; not only can you import entire modules contained within a package, you can selectively import specific classes or functions from a module contained within a package. You can also import the package itself as a module. The syntax is all the same; Python figures out what you mean based on the file layout of the package, and automatically does the right thing.

Example 5.7. Packages are modules, too

```
>>> from xml.dom import minidom        (1)
>>> minidom
<module 'xml.dom.minidom' from 'C:\Python21\lib\xml\dom\minidom.pyc'>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml.dom.minidom import Element (2)
>>> Element
<class xml.dom.minidom.Element at 01095744>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml import dom                (3)
>>> dom
<module 'xml.dom' from 'C:\Python21\lib\xml\dom\__init__.pyc'>
>>> import xml                         (4)
>>> xml
<module 'xml' from 'C:\Python21\lib\xml\__init__.pyc'>
```

(1) Here we're importing a module (`minidom`) from a nested package (`xml.dom`). The result is that `minidom` is imported into our namespace, and in order to reference classes within the `minidom` module (like `Element`), we have to preface them with the module name.

(2) Here we are importing a class (`Element`) from a module (`minidom`) from a nested package (`xml.dom`). The result is that `Element` is imported directly into our namespace. Note that this does not interfere with the previous import; the `Element` class can now be referenced in two ways (but it's all still the same class).

(3) Here we are importing the `dom` package (a nested package of `xml`) as a module in and of itself. Any level of a package can be treated as a module, as we'll see in a moment. It can even have its own attributes and methods, just the modules we've seen before.

(4) Here we are importing the root level `xml` package as a module.

So how can a package (which is just a directory on disk) be imported and treated as a module (which is always a file on disk)? The answer is the magical `__init__.py` file. You see, packages are not simply directories; they are directories with a specific file, `__init__.py`, inside. This file defines the attributes and methods of the package. For instance, `xml.dom` contains a `Node` class, which is defined in `xml/dom/__init__.py`. When you import a package as a module (like `dom` from `xml`), you're really importing its `__init__.py` file.

> **Note: What makes a package**
> A package is a directory with the special `__init__.py` file in it. The `__init__.py` file defines the attributes and methods of the package. It doesn't have to define anything; it can just be an empty file, but it has to exist. But if `__init__.py` doesn't exist, the directory is just a directory, not a package, and it can't be imported or contain modules or nested packages.

So why bother with packages? Well, they provide a way to logically group related modules. Instead of having an `xml` package with `sax` and `dom` packages inside, the authors could have chosen to put all the `sax` functionality in `xmlsax.py` and all the `dom` functionality in `xmldom.py`, or even put all of it in a single module. But that would have been unwieldy (as of this writing, the XML package has over 3000 lines of code) and difficult to manage (separate source files mean multiple people can work on different areas simultaneously).

If you ever find yourself writing a large subsystem in Python (or, more likely, when you realize that your small subsystem has grown into a large one), invest some time designing a good package architecture. It's one of the many things Python is good at, so take advantage of it.

5.3. Parsing XML

As I was saying, actually parsing an XML document is very simple: one line of code. Where you go from there is up to you.

Example 5.8. Loading an XML document (for real this time)

```
>>> from xml.dom import minidom                                        (1)
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')  (2)
>>> xmldoc                                                              (3)
<xml.dom.minidom.Document instance at 010BE87C>
>>> print xmldoc.toxml()                                                (4)
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

(1) As we saw in the previous section, this imports the `minidom` module from the `xml.dom` package.

(2) Here is the one line of code that does all the work: `minidom.parse` takes one argument and returns a parsed representation of the XML document. The argument can be many things; in this case, it's simply a filename of an XML document on my local disk. (To follow along, you'll need to change the path to point to your downloaded examples directory.) But you can also pass a file object, or even a file−like object. We'll take advantage of this flexibility later in this chapter.

(3) The object returned from `minidom.parse` is a `Document` object, a descendant of the `Node` class. This `Document` object is the root level of a complex tree−like structure of interlocking Python objects that completely represent the XML document we passed to `minidom.parse`.

(4) `toxml` is a method of the `Node` class (and is therefore available on the `Document` object we got from `minidom.parse`). `toxml` prints out the XML that this `Node` represents. For the `Document` node, this prints out the entire XML document.

Now that we have an XML document in memory, we can start traversing through it.

Example 5.9. Getting child nodes

```
>>> xmldoc.childNodes    (1)
[<DOM Element: grammar at 17538908>]
>>> xmldoc.childNodes[0] (2)
<DOM Element: grammar at 17538908>
>>> xmldoc.firstChild    (3)
<DOM Element: grammar at 17538908>
```

(1) Every `Node` has a `childNodes` attribute, which is a list of the `Node` objects. A `Document` always has only one child node, the root element of the XML document (in this case, the `grammar` element).

(2) To get the first (and in this case, the only) child node, just use regular list syntax. Remember, there is nothing special going on here; this is just a regular Python list of regular Python objects.

(3)

Since getting the first child node of a node is a useful and common activity, the `Node` class has a `firstChild` attribute, which is synonymous with `childNodes[0]`. (There is also a `lastChild` attribute, which is synonymous with `childNodes[-1]`.)

Example 5.10. `toxml` works on any node

```
>>> grammarNode = xmldoc.firstChild
>>> print grammarNode.toxml() (1)
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

(1) Since the `toxml` method is defined in the `Node` class, it is available on any XML node, not just the `Document` element.

Example 5.11. Child nodes can be text

```
>>> grammarNode.childNodes                  (1)
[<DOM Text node "\n">, <DOM Element: ref at 17533332>, \
<DOM Text node "\n">, <DOM Element: ref at 17549660>, <DOM Text node "\n">]
>>> print grammarNode.firstChild.toxml()    (2)



>>> print grammarNode.childNodes[1].toxml() (3)
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print grammarNode.childNodes[3].toxml() (4)
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
>>> print grammarNode.lastChild.toxml()     (5)


```

(1) Looking at the XML in `binary.xml`, you might think that the `grammar` has only two child nodes, the two `ref` elements. But you're missing something: the carriage returns! After the `'<grammar>'` and before the first `'<ref>'` is a carriage return, and this text counts as a child node of the `grammar` element. Similarly, there is a carriage return after each `'</ref>'`; these also count as child nodes. So `grammar.childNodes` is actually a list of 5 objects: 3 `Text` objects and 2 `Element` objects.

(2) The first child is a `Text` object representing the carriage return after the `'<grammar>'` tag and before the first `'<ref>'` tag.

(3) The second child is an `Element` object representing the first `ref` element.

(4) The fourth child is an `Element` object representing the second `ref` element.

(5) The last child is a `Text` object representing the carriage return after the `'</ref>'` end tag and before the `'</grammar>'` end tag.

Example 5.12. Drilling down all the way to text

```
>>> grammarNode
<DOM Element: grammar at 19167148>
>>> refNode = grammarNode.childNodes[1] (1)
>>> refNode
<DOM Element: ref at 17987740>
>>> refNode.childNodes                  (2)
[<DOM Text node "\n">, <DOM Text node "  ">, <DOM Element: p at 19315844>, \
<DOM Text node "\n">, <DOM Text node "  ">, \
<DOM Element: p at 19462036>, <DOM Text node "\n">]
>>> pNode = refNode.childNodes[2]
>>> pNode
<DOM Element: p at 19315844>
>>> print pNode.toxml()                 (3)
<p>0</p>
>>> pNode.firstChild                    (4)
<DOM Text node "0">
>>> pNode.firstChild.data               (5)
u'0'
```

(1) As we saw in the previous example, the first `ref` element is `grammarNode.childNodes[1]`, since childNodes[0] is a `Text` node for the carriage return.

(2) The `ref` element has its own set of child nodes, one for the carriage return, a separate one for the spaces, one for the `p` element, and so forth.

(3) You can even use the `toxml` method here, deeply nested within the document.

(4) The `p` element has only one child node (you can't tell that from this example, but look at `pNode.childNodes` if you don't believe me), and it is a `Text` node for the single character `'0'`.

(5) The `.data` attribute of a `Text` node gives you the actual string that the text node represents. But what is that `'u'` in front of the string? The answer to that deserves its own section.

5.4. Unicode

Unicode is a system to represent characters from all the world's different languages. When Python parses an XML document, all data is stored in memory as unicode.

We'll get to all that in a minute, but first, some background.

Historical note. Before unicode, there were separate character encoding systems for each language, each using the same numbers (0−255) to represent that language's characters. Some languages (like Russian) had multiple conflicting standards about how to represent the same characters; other languages (like Japanese) had so many characters that they required multiple−byte character sets. Exchanging documents between systems was difficult because there was no way for a computer to tell for certain which character encoding scheme the document author had used; the computer only saw numbers, and the numbers could mean different things. Then think about trying to store these documents in the same place (like in the same database table); you would need to store the character encoding alongside each piece of text, and make sure to pass it around whenever you passed the text around. Then think about multilingual documents, with characters from multiple languages in the same document. (They typically used escape codes to switch modes; poof, we're in Russian koi8−r mode, so character 241 means this; poof, now we're in Mac Greek mode, so character 241 means something else. And so on.) These are the problems which unicode was designed to solve.

To solve these problems, unicode represents each character as a 2−byte number, from 0 to 65535.[11] Each 2−byte number represents a unique character used in at least one of the world's languages. (Characters that are used in multiple languages have the same numeric code.) There is exactly 1 number per character, and exactly 1 character per number. Unicode data is never ambiguous.

Of course, there is still the matter of all these legacy encoding systems. 7−bit ASCII, for instance, which stores English characters as numbers ranging from 0 to 127. (65 is capital "A", 97 is lowercase "a", and so forth.) English has a very simple alphabet, so it can be completely expressed in 7−bit ASCII. Western European languages like French, Spanish, and German all use an encoding system called ISO−8859−1 (also called "latin−1"), which uses the 7−bit ASCII characters for the numbers 0 through 127, but then extends into the 128−255 range for characters like n−with−a−tilde−over−it (241), and u−with−two−dots−over−it (252). And unicode uses the same characters as 7−bit ASCII for 0 through 127, and the same characters as ISO−8859−1 for 128 through 255, and then extends from there into characters for other languages with the remaining numbers, 256 through 65535.

When dealing with unicode data, you may at some point need to convert the data back into one of these other legacy encoding systems. For instance, to integrate with some other computer system which expects its data in a specific 1−byte encoding scheme, or to print it to a non−unicode−aware terminal or printer. Or to store it in an XML document which explicitly specifies the encoding scheme.

And on that note, let's get back to Python.

Python has had unicode support throughout the language since version 2.0.[12] The XML package uses unicode to store all parsed XML data, but you can use unicode anywhere.

Example 5.13. Introducing unicode

```
>>> s = u'Dive in'            (1)
>>> s
u'Dive in'
>>> print s                   (2)
Dive in
```

(1) To create a unicode string instead of a regular ASCII string, add the letter "u" before the string. Note that this particular string doesn't have any non−ASCII characters. That's fine; unicode is a superset of ASCII (a very large superset at that), so any regular ASCII string can also be stored as unicode.

(2) When printing a string, Python will attempt to convert it to your default encoding, which is usually ASCII. (More on this in a minute.) Since this unicode string is made up of characters that are also ASCII characters, printing it has the same result as printing a normal ASCII string; the conversion is seamless, and if you didn't know that s was a unicode string, you'd never notice the difference.

Example 5.14. Storing non−ASCII characters

```
>>> s = u'La Pe\xf1a'         (1)
>>> print s                   (2)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> print s.encode('latin−1') (3)
La Peña
```

(1) The real advantage of unicode, of course, is its ability to store non−ASCII characters, like the Spanish "ñ" (n with a tilde over it). The unicode character code for the tilde−n is 0xf1 in hexadecimal (241 in decimal), which you can type like this: \xf1.

(2) Remember I said that the `print` function attempts to convert a unicode string to ASCII so it can print it? Well, that's not going to work here, because our unicode string contains non−ASCII characters, so Python raises a `UnicodeError` error.

(3) Here's where the conversion−from−unicode−to−other−encoding−schemes comes in. `s` is a unicode string, but `print` can only print a regular string. To solve this problem, we call the `encode` method, available on every unicode string, to convert the unicode string to a regular string in the given encoding scheme, which we pass as a parameter. In this case, we're using `latin−1` (also known as `iso−8859−1`), which includes the tilde−n (whereas the default ASCII encoding scheme did not, since it only includes characters numbered 0 through 127).

Remember I said Python usually converted unicode to ASCII whenever it needed to make a regular string out of a unicode string? Well, this default encoding scheme is an option which you can customize.

Example 5.15. `sitecustomize.py`

```
# sitecustomize.py                   (1)
# this file can be anywhere in your Python path,
# but it usually goes in ${pythondir}/lib/site-packages/

import sys

sys.setdefaultencoding('iso-8859-1') (2)
```

(1) `sitecustomize.py` is a special script; Python will try to import it on startup, so any code in it will be run automatically. As the comment mentions, it can go anywhere (as long as `import` can find it), but it usually goes in the `site−packages` directory within your Python `lib` directory.

(2) `setdefaultencoding` function sets, well, the default encoding. This is the encoding scheme that Python will try to use whenever it needs to auto−coerce a unicode string into a regular string.

Example 5.16. Effects of setting the default encoding

```
>>> import sys
>>> sys.getdefaultencoding() (1)
'iso-8859-1'
>>> s = u'La Pe\xf1a'
>>> print s                  (2)
La Peña
```

(1) This example assumes that you have made the changes listed in the previous example to your `sitecustomize.py` file, and restarted Python. If your default encoding still says `'ascii'`, you didn't set up your `sitecustomize.py` properly, or you didn't restart Python. The default encoding can only be changed during Python startup; you can't change it later. (Due to some wacky programming tricks that I won't get into right now, you can't even call `sys.setdefaultencoding` after Python has started up. Dig into `site.py` and search for "`setdefaultencoding`" to find out how.)

(2) Now that the default encoding scheme includes all the characters we use in our string, Python has no problem auto−coercing the string and printing it.

Now, what about XML? Well, every XML document is in a specific encoding. Again, ISO−8859−1 is a popular encoding for data in Western European languages. KOI8−R is popular for Russian texts. The encoding, if specified, is in the header of the XML document.

Example 5.17. `russiansample.xml`

```
<?xml version="1.0" encoding="koi8-r"?>       (1)
<preface>
<title> @548A;>285</title>                    (2)
</preface>
```

(1) This is a sample extract from a real Russian XML document; it's part of the translation of the *Preface* of this book. Note the encoding, `koi8-r`, specified in the header.

(2) These are Cyrillic characters which, as far as I know, spell the Russian word for "Preface". If you open this file in a regular text editor, the characters will most likely like gibberish, because they're encoded using the `koi8-r` encoding scheme, but they're being displayed in `iso-8859-1`.

Example 5.18. Parsing `russiansample.xml`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('russiansample.xml') (1)
>>> title = xmldoc.getElementsByTagName('title')[0].firstChild.data
>>> title                                        (2)
u'\u041f\u0440\u0435\u0434\u0438\u0441\u043b\u043e\u0432\u0438\u0435'
>>> print title                                  (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> convertedtitle = title.encode('koi8-r')     (4)
>>> convertedtitle
'\xf0\xd2\xc5\xc4\xc9\xd3\xcc\xcf\xd7\xc9\xc5'
>>> print convertedtitle                         (5)
 @548A;>285
```

(1) I'm assuming here that you saved the previous example as `russiansample.xml` in the current directory. I am also, for the sake of completeness, assuming that you've changed your default encoding back to `'ascii'` by removing your `sitecustomize.py` file, or at least commenting out the `setdefaultencoding` line.

(2) Note that the text data of the `title` tag (now in the `title` variable, thanks to that long concatenation of Python functions which I hastily skipped over and, annoyingly, won't explain until the next section) −− the text data inside the XML document's `title` element is stored in unicode.

(3) Printing the title is not possible, because this unicode string contains non−ASCII characters, so Python can't convert it to ASCII because that doesn't make sense.

(4) We can, however, explicitly convert it to `koi8-r`, in which case we get a (regular, not unicode) string of single−byte characters (`f0`, `d2`, `c5`, and so forth) that are the `koi8-r`−encoded versions of the characters in the original unicode string.

(5) Printing the `koi8-r`−encoded string will probably show gibberish on your screen, because your Python IDE is interpreting those characters as `iso-8859-1`, not `koi8-r`. But at least they do print. (And, if you look carefully, it's the same gibberish that you saw when you opened the original XML document in a non−unicode−aware text editor. Python converted it from `koi8-r` into unicode when it parsed the XML document, and we've just converted it back.)

To sum up, unicode itself is a bit intimidating if you've never seen it before, but unicode data is really very easy to handle in Python. If your XML documents are all 7−bit ASCII (like the examples in this chapter), you will literally never think about unicode. Python will convert the ASCII data in the XML documents into unicode while parsing, and auto−coerce it back to ASCII whenever necessary, and you'll never even notice. But if you need to deal with dat in other languages, Python is ready.

Further reading

* Unicode.org is the home page of the unicode standard, including a brief technical introduction.
* Unicode Tutorial has some more examples of how to use Python's unicode functions, including how to force Python to coerce unicode into ASCII even when it doesn't really want to.
* Unicode Proposal is the original technical specification for Python's unicode functionality. For advanced unicode hackers only.

5.5. Searching for elements

Traversing XML documents by stepping through each node can be tedious. If you're looking for something in particular, buried deep within your XML document, there is a shortcut you can use to find it quickly: `getElementsByTagName`.

For this section, we'll be using the `binary.xml` grammar file, which looks like this:

Example 5.19. `binary.xml`

```
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

It has two `ref`s, `'bit'` and `'byte'`. A `bit` is either a `'0'` or `'1'`, and a `byte` is 8 `bit`s.

Example 5.20. Introducing `getElementsByTagName`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref') (1)
>>> reflist
[<DOM Element: ref at 136138108>, <DOM Element: ref at 136144292>]
>>> print reflist[0].toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print reflist[1].toxml()
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
```

(1) `getElementsByTagName` takes one argument, the name of the element you wish to find. It returns a list of `Element` objects, corresponding to the XML elements that have that name. In this case, we find two `ref` elements.

Example 5.21. Every element is searchable

```
>>> firstref = reflist[0]                      (1)
>>> print firstref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> plist = firstref.getElementsByTagName("p") (2)
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>]
>>> print plist[0].toxml()                     (3)
<p>0</p>
>>> print plist[1].toxml()
<p>1</p>
```

(1) Continuing from the previous example, the first object in our `reflist` is the `'bit'` `ref` element.

(2) We can use the same `getElementsByTagName` method on this `Element` to find all the `<p>` elements within the `'bit'` `ref` element.

(3) Just as before, the `getElementsByTagName` method returns a list of all the elements it found. In this case, we have two, one for each bit.

Example 5.22. Searching is actually recursive

```
>>> plist = xmldoc.getElementsByTagName("p") (1)
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>, <DOM Element: p at 136146124>]
>>> plist[0].toxml()                         (2)
'<p>0</p>'
>>> plist[1].toxml()
'<p>1</p>'
>>> plist[2].toxml()                         (3)
'<p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>'
```

(1) Note carefully the difference between this and the previous example. Previously, we were searching for `p` elements within `firstref`, but here we are searching for `p` elements within `xmldoc`, the root−level object that represents the entire XML document. This *does* find the `p` elements nested within the `ref` elements within the root `grammar` element.

(2) The first two `p` elements are within the first `ref` (the `'bit'` `ref`).

(3) The last `p` element is the one within the second `ref` (the `'byte'` `ref`).

5.6. Accessing element attributes

XML elements can have one or more attributes, and it is incredibly simple to access them once you have parsed an XML document.

For this section, we'll be using the `binary.xml` grammar file that we saw in the previous section.

> **Note: XML attributes and Python attributes**
> This section may be a little confusing, because of some overlapping terminology. Elements in an XML document have attributes, and Python objects also have attributes. When we parse an XML document, we get a bunch of Python objects that represent all the pieces of the XML document, and some of these Python objects represent attributes of the XML elements. But the (Python) objects that represent the (XML) attributes also have (Python) attributes, which are used to access various parts of the (XML) attribute that the object represents. I told you it was confusing. I am open to suggestions on how to distinguish these more clearly.

Example 5.23. Accessing element attributes

```
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref')
>>> bitref = reflist[0]
>>> print bitref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> bitref.attributes          (1)
<xml.dom.minidom.NamedNodeMap instance at 0x81e0c9c>
>>> bitref.attributes.keys()   (2) (3)
[u'id']
>>> bitref.attributes.values() (4)
[<xml.dom.minidom.Attr instance at 0x81d5044>]
>>> bitref.attributes["id"]    (5)
<xml.dom.minidom.Attr instance at 0x81d5044>
```

(1) Each `Element` object has an attribute called `attributes`, which is a `NamedNodeMap` object. This sounds scary, but it's not, because a `NamedNodeMap` is an object that acts like a dictionary, so you already know how to use it.

(2) Treating the `NamedNodeMap` as a dictionary, we can get a list of the names of the attributes of this element by using `attributes.keys()`. This element has only one attribute, `'id'`.

(3) Attribute names, like all other text in an XML document, are stored in unicode.

(4) Again treating the `NamedNodeMap` as a dictionary, we can get a list of the values of the attributes by using `attributes.values()`. The values are themselves objects, of type `Attr`. We'll see how to get useful information out of this object in the next example.

(5) Still treating the `NamedNodeMap` as a dictionary, we can access an individual attribute by name, using normal dictionary syntax. (Readers who have been paying extra−close attention will already know how the `NamedNodeMap` class accomplishes this neat trick: by defining a `__getitem__` special method. Other readers can take comfort in the fact that they don't need to understand how it works in order to use it effectively.)

Example 5.24. Accessing individual attributes

```
>>> a = bitref.attributes["id"]
>>> a
<xml.dom.minidom.Attr instance at 0x81d5044>
>>> a.name  (1)
u'id'
>>> a.value (2)
u'bit'
```

(1) The `Attr` object completely represents a single XML attribute of a single XML element. The name of the attribute (the same name as we used to find this object in the `bitref.attributes` `NamedNodeMap` pseudo−dictionary) is stored in `a.name`.

(2) The actual text value of this XML attribute is stored in `a.value`.

Note: Attributes have no order
Like a dictionary, attributes of an XML element have no ordering. Attributes may *happen to be* listed in a certain order in the original XML document, and the `Attr` objects may *happen to be* listed in a certain order when the XML document is parsed into Python objects, but these orders are arbitrary and should carry no special meaning. You should always access individual attributes by name, like the keys of a dictionary.

5.7. Abstracting input sources

One of Python's greatest strengths is its dynamic binding, and one powerful use of dynamic binding is the *file−like object*.

Many functions which require an input source could simply take a filename, go open the file for reading, read it, and close it when they're done. But they don't. Instead, they take a *file−like object*.

In the simplest case, a *file−like object* is any object with a `read` method with an optional `size` parameter, which returns a string. When called with no `size` parameter, it reads everything there is to read from the input source and returns all the data as a single string. When called with a `size` parameter, it reads that much from the input source and returns that much data; when called again, it picks up where it left off and returns the next chunk of data.

This is how reading from real files works; the difference is that we're not limiting ourselves to real files. The input source could be anything: a file on disk, a web page, even a hard−coded string. As long as we pass a file−like object to the function, and the function simply calls the object's `read` method, the function can handle any kind of input source without specific code to handle each kind.

In case you were wondering how this relates to XML processing, `minidom.parse` is one such function which can take a file−like object.

Example 5.25. Parsing XML from a file

```
>>> from xml.dom import minidom
>>> fsock = open('binary.xml')    (1)
>>> xmldoc = minidom.parse(fsock) (2)
>>> fsock.close()                 (3)
>>> print xmldoc
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

(1) First, we open the file on disk. This gives us a file object.

(2) We pass the file object to `minidom.parse`, which calls the `read` method of `fsock` and reads the XML document from the file on disk.

(3) Be sure to call the `close` method of the file object after we're done with it. `minidom.parse` will not do this for you.

Well, that all seems like a colossal waste of time. After all, we've already seen that `minidom.parse` can simply take the filename and do all the opening and closing nonsense automatically. And it's true that if you know you're just going to be parsing a local file, you can pass the filename and `minidom.parse` is smart enough to Do The Right Thing(tm). But notice how similar −− and easy −− it is to parse an XML document straight from the Internet.

Example 5.26. Parsing XML from a URL

```
>>> import urllib
>>> usock = urllib.urlopen('http://slashdot.org/slashdot.rdf') (1)
>>> xmldoc = minidom.parse(usock)                              (2)
>>> usock.close()                                              (3)
>>> print xmldoc.toxml()                                       (4)
<?xml version="1.0" ?>
<rdf:RDF xmlns="http://my.netscape.com/rdf/simple/0.9/"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<channel>
<title>Slashdot</title>
<link>http://slashdot.org/</link>
<description>News for nerds, stuff that matters</description>
</channel>

<image>
<title>Slashdot</title>
<url>http://images.slashdot.org/topics/topicslashdot.gif</url>
<link>http://slashdot.org/</link>
</image>

<item>
<title>To HDTV or Not to HDTV?</title>
<link>http://slashdot.org/article.pl?sid=01/12/28/0421241</link>
</item>

[...snip...]
```

(1) As we saw in the previous chapter, `urlopen` takes a web page URL and returns a file−like object. Most importantly, this object has a `read` method which returns the HTML source of the web page.

(2) Now we pass the file−like object to `minidom.parse`, which obediently calls the `read` method of the object and parses the XML data that the `read` method returns. The fact that this XML data is now coming straight from a web page is completely irrelevant. `minidom.parse` doesn't know about web pages, and it doesn't care about web pages; it just knows about file−like objects.

(3) As soon as you're done with it, be sure to close the file−like object that `urlopen` gives you.

(4) By the way, this URL is real, and it really is XML. It's an XML representation of the current headlines on Slashdot, a technical news and gossip site.

Example 5.27. Parsing XML from a string (the easy but inflexible way)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> xmldoc = minidom.parseString(contents) (1)
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

(1) `minidom` has a method, `parseString`, which takes an entire XML document as a string and parses it. You can use this instead of `minidom.parse` if you know you already have your entire XML document in a string.

OK, so we can use the `minidom.parse` function for parsing both local files and remote URLs, but for parsing strings, we use... a different function. That means that if we want to be able to take input from a file, a URL, or a string, we'll need special logic to check whether it's a string, and call the `parseString` function instead. How unsatisfying.

If there were a way to turn a string into a file−like object, then we could simply pass this object to `minidom.parse`. And in fact, there is a module specifically designed for doing just that: `StringIO`.

Example 5.28. Introducing `StringIO`

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> import StringIO
>>> ssock = StringIO.StringIO(contents)   (1)
>>> ssock.read()                          (2)
"<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock.read()                          (3)
''
>>> ssock.seek(0)                         (4)
>>> ssock.read(15)                        (5)
'<grammar><ref i'
>>> ssock.read(15)
"d='bit'><p>0</p"
>>> ssock.read()
'><p>1</p></ref></grammar>'
>>> ssock.close()                         (6)
```

(1) The `StringIO` module contains a single class, also called `StringIO`, which allows you to turn a string into a file−like object. The `StringIO` class takes the string as a parameter when creating an instance.

(2) Now we have a file−like object, and we can do all sorts of file−like things with it. Like `read`, which returns the original string.

(3) Calling `read` again returns an empty string. This is how real file objects work too; once you read the entire file, you can't read any more without explicitly seeking to the beginning of the file. The `StringIO` object works the same way.

(4) You can explicitly seek to the beginning of the string, just like seeking through a file, by using the `seek` method of the `StringIO` object.

(5) You can also read the string in chunks, by passing a `size` parameter to the `read` method.

(6) At any time, `read` will return the rest of the string that you haven't read yet. All of this is exactly how file objects work; hence the term *file−like object*.

Example 5.29. Parsing XML from a string (the file−like object way)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock = StringIO.StringIO(contents)
>>> xmldoc = minidom.parse(ssock) (1)
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

(1) Now we can pass the file−like object (really a `StringIO`) to `minidom.parse`, which will call the object's `read` method and happily parse away, never knowing that its input came from a hard−coded string.

So now we know how to use a single function, `minidom.parse`, to parse an XML document stored on a web page, in a local file, or in a hard−coded string. For a web page, we use `urlopen` to get a file−like object; for a local file, we use `open`; and for a string, we use `StringIO`. Now let's take it one step further and generalize *these* differences as well.

Example 5.30. `openAnything`

```
def openAnything(source):                  (1)
    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)      (2)
```

```
    except (IOError, OSError):
        pass

    # try to open with native open function (if source is pathname)
    try:
        return open(source)                (3)
    except (IOError, OSError):
        pass

    # treat source as string
    import StringIO
    return StringIO.StringIO(str(source))  (4)
```

(1) The `openAnything` function takes a single parameter, `source`, and returns a file−like object. `source` is a string of some sort; it can either be a URL (like `'http://slashdot.org/slashdot.rdf'`), a full or partial pathname to a local file (like `'binary.xml'`), or a string that contains actual XML data to be parsed.

(2) First, we see if `source` is a URL. We do this through brute force: we try to open it as a URL and silently ignore errors caused by trying to open something which is not a URL. This is actually elegant in the sense that, if `urllib` ever supports new types of URLs in the future, we will also support them without recoding.

(3) If `urllib` yelled at us and told us that `source` wasn't a valid URL, we assume it's a path to a file on disk and try to open it. Again, we don't do anything fancy to check whether `source` is a valid filename or not (the rules for valid filenames vary wildly between different platforms anyway, so we'd probably get them wrong anyway). Instead, we just blindly open the file, and silently trap any errors.

(4) By this point, we have to assume that `source` is a string that has hard−coded data in it (since nothing else worked), so we use `StringIO` to create a file−like object out of it and return that. (In fact, since we're using the `str` function, `source` doesn't even need to be a string; it could be any object, and we'll use its string representation, as defined by its `__str__` special method.)

Now we can use this `openAnything` function in conjunction with `minidom.parse` to make a function that takes a `source` that refers to an XML document somehow (either as a URL, or a local filename, or a hard−coded XML document in a string) and parses it.

Example 5.31. Using `openAnything` in `kgp.py`

```
class KantGenerator:
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc
```

5.8. Standard input, output, and error

UNIX users are already familiar with the concept of standard input, standard output, and standard error. This section is for the rest of you.

Standard output and standard error (commonly abbreviated `stdout` and `stderr`) are pipes that are built into every UNIX system. When you `print` something, it goes to the `stdout` pipe; when your program crashes and prints out debugging information (like a traceback in Python), it goes to the `stderr` pipe. Both of these pipes are ordinarily just connected to the terminal window where you are working, so when a program prints, you see the output, and when a program crashes, you see the debugging information. (If you're working on a system with a window−based Python IDE, `stdout` and `stderr` default to your "Interactive Window".)

Example 5.32. Introducing `stdout` and `stderr`

```
>>> for i in range(3):
...     print 'Dive in'             (1)
Dive in
Dive in
Dive in
>>> import sys
>>> for i in range(3):
...     sys.stdout.write('Dive in') (2)
Dive inDive inDive in
>>> for i in range(3):
...     sys.stderr.write('Dive in') (3)
Dive inDive inDive in
```

(1) As we saw in Example 3.28, we can use Python's built−in `range` function to build simple counter loops that repeat something a set number of times.

(2) `stdout` is a file−like object; calling its `write` function will print out whatever string you give it. In fact, this is what the `print` function really does; it adds a carriage return to the end of the string you're printing, and calls `sys.stdout.write`.

(3) In the simplest case, `stdout` and `stderr` send their output to the same place: the Python IDE (if you're in one), or the terminal (if you're running Python from the command line). Like `stdout`, `stderr` does not add carriage returns for you; if you want them, add them yourself.

`stdout` and `stderr` are both file−like objects, like the ones we discussed in *Abstracting input sources*, but they are both write−only. They have no `read` method, only `write`. Still, they are file−like objects, and you can assign any other file− or file−like object to them to redirect their output.

Example 5.33. Redirecting output

```
[f8dy@oliver kgp]$ python stdout.py
Dive in
[f8dy@oliver kgp]$ cat out.log
This message will be logged instead of displayed
```

If you have not already done so, you can download this and other examples used in this book.

```
#stdout.py
import sys

print 'Dive in'                                          (1)
saveout = sys.stdout                                     (2)
fsock = open('out.log', 'w')                             (3)
sys.stdout = fsock                                       (4)
print 'This message will be logged instead of displayed' (5)
sys.stdout = saveout                                     (6)
fsock.close()                                            (7)
```

(1) This will print to the IDE "Interactive Window" (or the terminal, if running the script from the command line).

(2) Always save `stdout` before redirecting it, so you can set it back to normal later.

(3) Open a new file for writing.

(4) Redirect all further output to the new file we just opened.

(5) This will be "printed" to the log file only; it will not be visible in the IDE window or on the screen.

(6) Set `stdout` back to the way it was before we mucked with it.

(7) Close the log file.

Redirecting `stderr` works exactly the same way, using `sys.stderr` instead of `sys.stdout`.

Example 5.34. Redirecting error information

```
[f8dy@oliver kgp]$ python stderr.py
[f8dy@oliver kgp]$ cat error.log
Traceback (most recent line last):
  File "stderr.py", line 5, in ?
    raise Exception, 'this error will be logged'
Exception: this error will be logged
```

If you have not already done so, you can download this and other examples used in this book.

```
#stderr.py
import sys

fsock = open('error.log', 'w')               (1)
sys.stderr = fsock                           (2)
raise Exception, 'this error will be logged' (3) (4)
```

(1) Open the log file where we want to store debugging information.

(2) Redirect standard error by assigning the file object of our newly−opened log file to `stderr`.

(3) Raise an exception. Note from the screen output that this does *not* print anything on screen. All the normal traceback information has been written to `error.log`.

(4) Also note that we're not explicitly closing our log file, nor are we setting `stderr` back to its original value. This is fine, since once the program crashes (due to our exception), Python will clean up and close the file for us, and it doesn't make any difference that `stderr` is never restored, since, as I mentioned, the program crashes and Python ends. Restoring the original is more important for `stdout`, if you expect to go do other stuff within the same script afterwards.

Standard input, on the other hand, is a read−only file object, and it represents the data flowing into the program from some previous program. This will likely not make much sense to classic Mac OS users, or even Windows users unless you were ever fluent on the MS−DOS command line. The way it works is that you can construct a chain of commands in a single line, so that one program's output becomes the input for the next program in the chain. The first program simply outputs to standard output (without doing any special redirecting itself, just doing normal `print` statements or whatever), and the next program reads from standard input, and the operating system takes care of connecting one program's output to the next program's input.

Example 5.35. Chaining commands

```
[f8dy@oliver kgp]$ python kgp.py −g binary.xml         (1)
01100111
[f8dy@oliver kgp]$ cat binary.xml                      (2)
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "−//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
```

```
</grammar>
[f8dy@oliver kgp]$ cat binary.xml | python kgp.py −g − (3) (4)
10110001
```

(1) As we saw in *Diving in*, this will print a string of eight random bits, `0` or `1`.

(2) This simply prints out the entire contents of `binary.xml`. (Windows users should use `type` instead of `cat`.)

(3) This prints the contents of `binary.xml`, but the "`|`" character, called the "pipe" character, means that the contents will not be printed to the screen. Instead, they will become the standard input of the next command, which in this case calls our Python script.

(4) Instead of specifying a module (like `binary.xml`), we specify "`−`", which causes our script to load the grammar from standard input instead of from a specific file on disk. (More on how this happens in the next example.) So the effect is the same as the first syntax, where we specified the grammar filename directly, but think of the expansion possibilities here. Instead of simply doing `cat binary.xml`, we could run a script that dynamically generates the grammar, then we can pipe it into our script. It could come from anywhere: a database, or some grammar−generating meta−script, or whatever. The point is that we don't have to change our `kgp.py` script at all to incorporate any of this functionality. All we have to do is be able to take grammar files from standard input, and we can separate all the other logic into another program.

So how does our script "know" to read from standard input when the grammar file is "`−`"? It's not magic; it's just code.

Example 5.36. Reading from standard input in `kgp.py`

```
def openAnything(source):
    if source == "−":    (1)
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:

[... snip ...]
```

(1) This is the `openAnything` function from `toolbox.py`, which we previously examined in *Abstracting input sources*. All we've done is add three lines of code at the beginning of the function to check if the source is "`−`"; if so, we return `sys.stdin`. Really, that's it! Remember, `stdin` is a file−like object with a `read` method, so the rest of our code (in `kgp.py`, where we call `openAnything`) doesn't change a bit.

5.9. Caching node lookups

`kgp.py` employs several tricks which may or may not be useful to you in your XML processing. The first one takes advantage of the consistent structure of the input documents to build a cache of nodes.

A grammar file defines a series of `ref` elements. Each `ref` contains one or more `p` elements, which can contain lots of different things, including `xref`s. Whenever we encounter an `xref`, we look for a corresponding `ref` element with the same `id` attribute, and choose one of the `ref` element's children and parse it. (We'll see how this random choice is made in the next section.)

This is how we build up our grammar: define `ref` elements for the smallest pieces, then define `ref` elements which "include" the first `ref` elements by using `xref`, and so forth. Then we parse the "largest" reference and follow each `xref`, and eventually output real text. The text we output depends on the (random) decisions we make each time we fill in an `xref`, so the output is different each time.

This is all very flexible, but there is one downside: performance. When we find an `xref` and need to find the corresponding `ref` element, we have a problem. The `xref` has an `id` attribute, and we want to find the `ref` element that has that same `id` attribute, but there is no easy way to do that. The slow way to do it would be to get the entire list of `ref` elements each time, then manually loop through and look at each `id` attribute. The fast way is to do that once and build a cache, in the form of a dictionary.

Example 5.37. `loadGrammar`

```
def loadGrammar(self, grammar):
    self.grammar = self._load(grammar)
    self.refs = {}                                       (1)
    for ref in self.grammar.getElementsByTagName("ref"): (2)
        self.refs[ref.attributes["id"].value] = ref      (3) (4)
```

(1) Start by creating an empty dictionary, `self.refs`.

(2) As we saw in *Searching for elements*, `getElementsByTagName` returns a list of all the elements of a particular name. We easily can get a list of all the `ref` elements, then simply loop through that list.

(3) As we saw in *Accessing element attributes*, we can access individual attributes of an element by name, using standard dictionary syntax. So the keys of our `self.refs` dictionary will be the values of the `id` attribute of each `ref` element.

(4) The values of our `self.refs` dictionary will be the `ref` elements themselves. As we saw in *Parsing XML*, each element, each node, each comment, each piece of text in a parsed XML document is an object.

Once we build this cache, whenever we come across an `xref` and need to find the `ref` element with the same `id` attribute, we can simply look it up in `self.refs`.

Example 5.38. Using our `ref` element cache

```
def do_xref(self, node):
    id = node.attributes["id"].value
    self.parse(self.randomChildElement(self.refs[id]))
```

We'll explore the `randomChildElement` function in the next section.

5.10. Finding direct children of a node

Another useful techique when parsing XML documents is finding all the direct child elements of a particular element. For instance, in our grammar files, a `ref` element can have several `p` elements, each of which can contain many things, including other `p` elements. We want to find just the `p` elements that are children of the `ref`, not `p` elements that are children of other `p` elements.

You might think we could simply use `getElementsByTagName` for this, but we can't. `getElementsByTagName` searches recursively and returns a single list for all the elements it finds. Since `p` elements can contain other `p` elements, we can't use `getElementsByTagName`, because it would return nested `p` elements that we don't want. To find only direct child elements, we'll need to do it ourselves.

Example 5.39. Finding direct child elements

```
def randomChildElement(self, node):
    choices = [e for e in node.childNodes
               if e.nodeType == e.ELEMENT_NODE] (1) (2) (3)
```

```
    chosen = random.choice(choices)             (4)
    return chosen
```

(1) As we saw in Example 5.9, the `childNodes` attribute returns a list of all the child nodes of an element.

(2) However, as we saw in Example 5.11, the list returned by `childNodes` contains all different types of nodes, including text nodes. That's not what we're looking for here. We only want the children that are elements.

(3) Each node has a `nodeType` attribute, which can be `ELEMENT_NODE`, `TEXT_NODE`, `COMMENT_NODE`, or any number of other values. The complete list of possible values is in the `__init__.py` file in the `xml.dom` package. (See *Packages* for more on packages.) But we're just interested in nodes that are elements, so we can filter the list to only include those nodes whose `nodeType` is `ELEMENT_NODE`.

(4) Once we have a list of actual elements, choosing a random one is easy. Python comes with a module called `random` which includes several useful functions. The `random.choice` function takes a list of any number of items and returns a random item. In this case, the list contains `p` elements, so `chosen` is now a `p` element selected at random from the children of the `ref` element we were given.

5.11. Creating separate handlers by node type

The third useful XML processing tip involves separating your code into logical functions, based on node types and element names. Parsed XML documents are made up of various types of nodes, each represented by a Python object. The root level of the document itself is represented by a `Document` object. The `Document` then contains one or more `Element` objects (for actual XML tags), each of which may contain other `Element` objects, `Text` objects (for bits of text), or `Comment` objects (for embedded comments). Python makes it easy to write a dispatcher to separate the logic for each node type.

Example 5.40. Class names of parsed XML objects

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('kant.xml') (1)
>>> xmldoc
<xml.dom.minidom.Document instance at 0x01359DE8>
>>> xmldoc.__class__                   (2)
<class xml.dom.minidom.Document at 0x01105D40>
>>> xmldoc.__class__.__name__          (3)
'Document'
```

(1) Assume for a moment that `kant.xml` is in the current directory.

(2) As we saw in *Packages*, the object returned by parsing an XML document is a `Document` object, as defined in the `minidom.py` in the `xml.dom` package. As we saw in *Instantiating classes*, `__class__` is built−in attribute of every Python object.

(3) Furthermore, `__name__` is a built−in attribute of every Python class, and it is a string. This string is not mysterious; it's the same as the class name you type when you define a class yourself. (See *Defining classes*.)

Fine, so now we can get the class name of any particular XML node (since each XML node is represented as a Python object). How can we use this to our advantage to separate the logic of parsing each node type? The answer is `getattr`, which we first saw in *Getting object references with getattr*.

Example 5.41. `parse`, a generic XML node dispatcher

```
    def parse(self, node):
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__) (1) (2)
        parseMethod(node) (3)
```

(1) First off, notice that we're constructing a larger string based on the class name of the node we were passed (in the `node` argument). So if we're passed a `Document` node, we're constructing the string `'parse_Document'`, and so forth.

(2) Now we can treat that string as a function name, and get a reference to the function itself using `getattr`

(3) Finally, we can call that function and pass the node itself as an argument. The next example shows the definitions of each of these functions.

Example 5.42. Functions called by the `parse` dispatcher

```
def parse_Document(self, node): (1)
    self.parse(node.documentElement)

def parse_Text(self, node):    (2)
    text = node.data
    if self.capitalizeNextWord:
        self.pieces.append(text[0].upper())
        self.pieces.append(text[1:])
        self.capitalizeNextWord = 0
    else:
        self.pieces.append(text)

def parse_Comment(self, node): (3)
    pass

def parse_Element(self, node): (4)
    handlerMethod = getattr(self, "do_%s" % node.tagName)
    handlerMethod(node)
```

(1) `parse_Document` is only ever called once, since there is only one `Document` node in an XML document, and only one `Document` object in the parsed XML representation. It simply turns around and parses the root element of the grammar file.

(2) `parse_Text` is called on nodes that represent bits of text. The function itself does some special processing to handle automatic capitalization of the first word of a sentence, but otherwise simply appends the represented text to a list.

(3) `parse_Comment` is just a `pass`, since we don't care about embedded comments in our grammar files. Note, however, that we still need to define the function and explicitly make it do nothing. If the function did not exist, our generic `parse` function would fail as soon as it stumbled on a comment, because it would try to find the non−existent `parse_Comment` function. Defining a separate function for every node type, even ones we don't use, allows the generic `parse` function to stay simple and dumb.

(4) The `parse_Element` method is actually itself a dispatcher, based on the name of the element's tag. The basic idea is the same: take what distinguishes elements from each other (their tag names) and dispatch to a separate function for each of them. We construct a string like `'do_xref'` (for an `<xref>` tag), find a function of that name, and call it. And so forth for each of the other tag names that might be found in the course of parsing a grammar file (`<p>` tags, `<choice>` tags).

In this example, the dispatch functions `parse` and `parse_Element` simply find other methods in the same class. If your processing is very complex (or you have many different tag names), you could break up your code into separate modules, and use dynamic importing to import each module and call whatever functions you needed. Dynamic importing will be discussed in *Data−Centric Programming*.

5.12. Handling command line arguments

Python fully supports creating programs that can be run on the command line, complete with command−line arguments and either short− or long−style flags to specify various options. None of this is XML−specific, but this script makes good use of command−line processing, so it seemed like a good time to mention it.

It's difficult to talk about command line processing without understanding how command line arguments are exposed to your Python program, so let's write a simple program to see them.

Example 5.43. Introducing `sys.argv`

If you have not already done so, you can download this and other examples used in this book.

```
#argecho.py
import sys

for arg in sys.argv: (1)
    print arg
```

(1) Each command line argument passed to the program will be in `sys.argv`, which is just a list. Here we are printing each argument on a separate line.

Example 5.44. The contents of `sys.argv`

```
[f8dy@oliver py]$ python argecho.py             (1)
argecho.py
[f8dy@oliver py]$ python argecho.py abc def     (2)
argecho.py
abc
def
[f8dy@oliver py]$ python argecho.py −−help      (3)
argecho.py
−−help
[f8dy@oliver py]$ python argecho.py −m kant.xml (4)
argecho.py
−m
kant.xml
```

(1) The first thing to know about `sys.argv` is that it contains the name of the script we're calling. We will actually use this knowledge to our advantage later, in *Data−Centric Programming*. Don't worry about it for now.

(2) Command line arguments are separated by spaces, and each shows up as a separate element in the `sys.argv` list.

(3) Command line flags, like `−−help`, also show up as their own element in the `sys.argv` list.

(4) To make things even more interesting, some command line flags themselves take arguments. For instance, here we have a flag (`−m`) which takes an argument (`kant.xml`). Both the flag itself and the flag's argument are simply sequential elements in the `sys.argv` list. No attempt is made to associate one with the other; all you get is a list.

So as we can see, we certainly have all the information passed on the command line, but then again, it doesn't look like it's going to be all that easy to actually use it. For simple programs that only take a single argument and have no flags, you can simply use `sys.argv[1]` to access the argument. There's no shame in this; I do it all the time. For more complex programs, you need the `getopt` module.

Example 5.45. Introducing `getopt`

```
def main(argv):
    grammar = "kant.xml"                 (1)
```

```
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="]) (2)
    except getopt.GetoptError:           (3)
        usage()                          (4)
        sys.exit(2)

...

if __name__ == "__main__":
    main(sys.argv[1:])
```

(1) First off, look at the bottom of the example and notice that we're calling the `main` function with `sys.argv[1:]`. Remember, `sys.argv[0]` is the name of the script that we're running; we don't care about that for command line processing, so we chop it off and pass the rest of the list.

(2) This is where all the interesting processing happens. The `getopt` function of the `getopt` takes three parameters: the argument list (which we got from `sys.argv[1:]`), a string containing all the possible single−character command line flags that this program accepts, and a list of longer command line flags that are equivalent to the single−character versions. This is quite confusing at first glance, and is explained in more detail below.

(3) If anything goes wrong trying to parse these command line flags, `getopt` will raise an exception, which we catch. We told `getopt` all the flags we understand, so this probably means that the end user passed some command line flag that we don't understand.

(4) As is standard practice in the UNIX world, when our script is passed flags it doesn't understand, we print out a summary of proper usage and exit gracefully. Note that I haven't shown the `usage` function here. We would still need to code that somewhere and have it print out the appropriate summary; it's not automatic.

So what are all those parameters we pass to the `getopt` function? Well, the first one is simply the raw list of command line flags and arguments (not including the first element, the script name, which we already chopped off before calling our `main` function). The second is the list of short command line flags that our script accepts.

`"hg:d"`

`−h`
: print usage summary

`−g ...`
: use specified grammar file or URL

`−d`
: show debugging information while parsing

The first and third flags are simply standalone flags; you specify them or you don't, and they do things (print help) or change state (turn on debugging). However, the second flag (`−g`) *must* be followed by an argument, which is the name of the grammar file to read from. In fact it can be a filename or a web address, and we don't know which yet (we'll figure it out later), but we know it has to be *something*. So we tell `getopt` this by putting a colon after the `g` in that second parameter to the `getopt` function.

To further complicate things, our script accepts either short flags (like `−h`) or long flags (like `−−help`), and we want them to do the same thing. This is what the third parameter to `getopt` is for, to specify a list of the long flags that correspond to the short flags we specified in the second parameter.

`["help", "grammar="]`

`−−help`
: print usage summary

--grammar ...
: use specified grammar file or URL

Three things of note here:

1. All long flags are preceded by two dashes on the command line, but we don't include those dashes when calling `getopt`. They are understood.
2. The `--grammar` flag must always be followed by an additional argument, just like the `-g` flag. This is notated by an equals sign, `"grammar="`.
3. The list of long flags is shorter than the list of short flags, because the `-d` flag does not have a corresponding long version. This is fine; only `-d` will turn on debugging. But the order of short and long flags needs to be the same, so you'll need to specify all the short flags that *do* have corresponding long flags first, then all the rest of the short flags.

Confused yet? Let's look at the actual code and see if it makes sense in context.

Example 5.46. Handling command−line arguments in `kgp.py`

```
def main(argv):                          (1)
    grammar = "kant.xml"
    try:
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:                (2)
        if opt in ("-h", "--help"):      (3)
            usage()
            sys.exit()
        elif opt == '-d':                (4)
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"): (5)
            grammar = arg

    source = "".join(args)               (6)

    k = KantGenerator(grammar, source)
    print k.output()
```

(1) The `grammar` variable will keep track of the grammar file we're using. We initialize it here in case it's not specified on the command line (using either the `-g` or the `--grammar` flag).

(2) The `opts` variable that we get back from `getopt` contains a list of tuples, flag and argument. If the flag doesn't take an argument, then `arg` will simply be `None`. This makes it easier to loop through the flags.

(3) `getopt` validates that the command line flags are acceptable, but it doesn't do any sort of conversion between short and long flags. If you specify the `-h` flag, `opt` will contain `"-h"`; if you specify the `--help` flag, `opt` will contain `"--help"`. So we need to check for both.

(4) Remember, the `-d` flag didn't have a corresponding long flag, so we only need to check for the short form. If we find it, we set a global variable that we'll refer to later to print out debugging information. (I used this during the development of the script. What, you thought all these examples worked on the first try?)

(5) If we find a grammar file, either with a `-g` flag or a `--grammar` flag, we save the argument that followed it (stored in `arg`) into our `grammar` variable, overwriting the default that we initialized at the top of the `main` function.

(6) That's it. We've looped through and dealt with all the command line flags. That means that anything left must

be command line arguments. These come back from the `getopt` function in the `args` variable. In this case, we're treating them as source material for our parser. If there are no command line arguments specified, `args` will be an empty list, and `source` will end up as the empty string.

5.13. Putting it all together

We've covered a lot of ground. Let's step back and see how all the pieces fit together.

To start with, this is a script that takes its arguments on the command line, using the `getopt` module.

```
def main(argv):
...
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
...
    for opt, arg in opts:
...
```

We create a new instance of the `KantGenerator` class, and pass it the grammar file and source that may or may not have been specified on the command line.

```
    k = KantGenerator(grammar, source)
```

The `KantGenerator` instance automatically loads the grammar, which is an XML file. We use our custom `openAnything` function to open the file (which could be stored in a local file or a remote web server), then use the built−in `minidom` parsing functions to parse the XML into a tree of Python objects.

```
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
```

Oh, and along the way, we take advantage of our knowledge of the structure of the XML document to set up a little cache of references, which are just elements in the XML document.

```
    def loadGrammar(self, grammar):
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref
```

If we specified some source material on the command line, we use that; otherwise we rip through the grammar looking for the "top−level" reference (that isn't referenced by anything else) and use that as a starting point.

```
    def getDefaultSource(self):
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)
```

Now we rip through our source material. The source material is also XML, and we parse it one node at a time. To keep our code separated and more maintainable, we use separate handlers for each node type.

```
    def parse_Element(self, node):
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

We bounce through the grammar, parsing all the children of each `p` element,

```
    def do_p(self, node):
...
        if doit:
            for child in node.childNodes: self.parse(child)
```

replacing `choice` elements with a random child,

```
    def do_choice(self, node):
        self.parse(self.randomChildElement(node))
```

and replacing `xref` elements with a random child of the corresponding `ref` element, which we previously cached.

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

Eventually, we parse our way down to plain text,

```
    def parse_Text(self, node):
        text = node.data
...
            self.pieces.append(text)
```

which we print out.

```
def main(argv):
...
    k = KantGenerator(grammar, source)
    print k.output()
```

5.14. Summary

Python comes with powerful libraries for parsing and manipulating XML documents. The `minidom` takes an XML file and parses it into Python objects, providing for random access to arbitrary elements. Furthermore, this chapter shows how Python can be used to create a "real" standalone command−line script, complete with command−line flags, command−line arguments, error handling, even the ability to take input from the piped result of a previous program.

Before moving on to the next chapter, you should be comfortable doing all of these things:

- Parsing XML documents using `minidom`, searching through the parsed document, and accessing arbitrary element attributes and element children
- Organizing complex libraries into packages
- Converting unicode strings to different character encodings
- Chaining programs with standard input and output
- Defining command−line flags and validating them with `getopt`

---

[11] This, sadly, is *still* an oversimplification. Unicode now has been extended to handle ancient Chinese, Korean, and Japanese texts, which had so many different characters that the 2−byte unicode system could not represent them all. But Python doesn't currently support that out of the box, and I don't know if there is a project afoot to add it. You've reached the limits of my expertise, sorry.

[12] Actually, Python has had unicode support since version 1.6, but version 1.6 was a contractual obligation release that nobody likes to talk about, a bastard stepchild of a hippie youth best left forgotten. Even the official Python documentation claims that unicode was "new in version 2.0". It's a lie, but, like the lies of presidents who say they inhaled but didn't enjoy it, we choose to believe it because we remember our own misspent youths a bit too vividly.

Chapter 6. Unit Testing

6.1. Diving in

In previous chapters, we "dived in" by immediately looking at code and trying to understanding it as quickly as possible. Now that you have some Python under your belt, we're going to step back and look at the steps that happen *before* the code gets written.

In this chapter we're going to write a set of utility functions to convert to and from Roman numerals. You've most likely seen Roman numerals, even if you didn't recognize them. You may have seen them in copyrights of old movies and television shows ("Copyright `MCMXLVI`" instead of "Copyright `1946`"), or on the dedication walls of libraries or universities ("established `MDCCCLXXXVIII`" instead of "established `1888`"). You may also have seen them in outlines and bibliographical references. It's a system of representing numbers that really does date back to the ancient Roman empire (hence the name).

In Roman numerals, there are seven characters which are repeated and combined in various ways to represent numbers.

1. `I` = `1`
2. `V` = `5`
3. `X` = `10`
4. `L` = `50`
5. `C` = `100`
6. `D` = `500`
7. `M` = `1000`

There are some general rules for constructing Roman numerals:

1. Characters are additive. `I` is `1`, `II` is `2`, and `III` is `3`. `VI` is `6` (literally, "`5` and `1`"), `VII` is `7`, and `VIII` is `8`.
2. The tens characters (`I`, `X`, `C`, and `M`) can be repeated up to three times. At `4`, you have to subtract from the next highest fives character. You can't represent `4` as `IIII`; instead, it is represented as `IV` ("`1` less than `5`"). `40` is written as `XL` ("`10` less than `50`"), `41` as `XLI`, `42` as `XLII`, `43` as `XLIII`, and then `44` as `XLIV` ("`10` less than `50`, then `1` less than `5`").
3. Similarly, at `9`, you have to subtract from the next highest tens character: `8` is `VIII`, but `9` is `IX` ("`1` less than `10`"), not `VIIII` (since the `I` character can not be repeated four times). `90` is `XC`, `900` is `CM`.
4. The fives characters can not be repeated. `10` is always represented as `X`, never as `VV`. `100` is always `C`, never `LL`.
5. Roman numerals are always written highest to lowest, and read left to right, so order of characters matters very much. `DC` is `600`; `CD` is a completely different number (`400`, "`100` less than `500`"). `CI` is `101`; `IC` is not even a valid Roman numeral (because you can't subtract `1` directly from `100`; you would have to write it as `XCIX`, "`10` less than `100`, then `1` less than `10`").

These rules lead to a number of interesting observations:

1. There is only one correct way to represent a number as Roman numerals.
2. The converse is also true: if a string of characters is a valid Roman numeral, it represents only one number (*i.e.* it can only be read one way).
3. There is a limited range of numbers that can be expressed as Roman numerals, specifically `1` through `3999`. (The Romans did have several ways of expressing larger numbers, for instance by having a bar over a numeral to represent that its normal value should be multiplied by `1000`, but we're not going to deal with that. For the purposes of this chapter, Roman numerals go from `1` to `3999`.)

4. There is no way to represent `0` in Roman numerals. (Amazingly, the ancient Romans had no concept of `0` as a number. Numbers were for counting things you had; how can you count what you don't have?)
5. There is no way to represent negative numbers in Roman numerals.
6. There is no way to represent decimals or fractions in Roman numerals.

Given all of this, what would we expect out of a set of functions to convert to and from Roman numerals?

`roman.py` requirements

1. `toRoman` should return the Roman numeral representation for all integers `1` to `3999`.
2. `toRoman` should fail when given an integer outside the range `1` to `3999`.
3. `toRoman` should fail when given a non−integer decimal.
4. `fromRoman` should take a valid Roman numeral and return the number that it represents.
5. `fromRoman` should fail when given an invalid Roman numeral.
6. If you take a number, convert it to Roman numerals, then convert that back to a number, you should end up with the number you started with. So `fromRoman(toRoman(n)) == n` for all `n` in `1..3999`.
7. `toRoman` should always return a Roman numeral using uppercase letters.
8. `fromRoman` should only accept uppercase Roman numerals (*i.e.* it should fail when given lowercase input).

Further reading

- This site has more on Roman numerals, including a fascinating history of how Romans and other civilizations really used them (short answer: haphazardly and inconsistently).

6.2. Introducing `romantest.py`

Now that we've completely defined the behavior we expect from our conversion functions, we're going to do something a little unexpected: we're going to write a test suite that puts these functions through their paces and makes sure that they behave the way we want them to. You read that right: we're going to write code that tests code that we haven't written yet.

This is called unit testing, since the set of two conversion functions can be written and tested as a unit, separate from any larger program they may become part of later. Python has a framework for unit testing, the appropriately−named `unittest` module.

> **Note: Do you have unittest?**
> `unittest` is included with Python 2.1 and later. Python 2.0 users can download it from `pyunit.sourceforge.net`.

Unit testing is an important part of an overall testing−centric development strategy. If you write unit tests, it is important to write them early (preferably before writing the code that they test), and to keep them updated as code and requirements change. Unit testing is not a replacement for higher−level functional or system testing, but it is important in all phases of development:

- Before writing code, it forces you to detail your requirements in a useful fashion.
- While writing code, it keeps you from over−coding. When all the test cases pass, the function is complete.
- When refactoring code, it assures you that the new version behaves the same way as the old version.
- When maintaining code, it helps you cover your ass when someone comes screaming that your latest change broke their old code. ("But *sir*, all the unit tests passed when I checked it in...")

This is the complete test suite for our Roman numeral conversion functions, which are yet to be written but will eventually be in `roman.py`. It is not immediately obvious how it all fits together; none of these classes or methods reference any of the others. There are good reasons for this, as we'll see shortly.

Example 6.1. `romantest.py`

If you have not already done so, you can download this and other examples used in this book.

```
"""Unit test for roman.py"""

import roman
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
```

```
                (3250, 'MMMCCL'),
                (3313, 'MMMCCCXIII'),
                (3408, 'MMMCDVIII'),
                (3501, 'MMMDI'),
                (3610, 'MMMDCX'),
                (3743, 'MMMDCCXLIII'),
                (3844, 'MMMDCCCXLIV'),
                (3888, 'MMMDCCCLXXXVIII'),
                (3940, 'MMMCMXL'),
                (3999, 'MMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)

    def testDecimal(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
```

```
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

Further reading

- The PyUnit home page has an in−depth discussion of using the `unittest` framework, including advanced features not covered in this chapter.
- The PyUnit FAQ explains why test cases are stored separately from the code they test.
- *Python Library Reference* summarizes the `unittest` module.
- ExtremeProgramming.org discusses why you should write unit tests.
- The Portland Pattern Repository has an ongoing discussion of unit tests, including a standard definition, why you should code unit tests first, and several in−depth case studies.

6.3. Testing for success

The most fundamental part of unit testing is constructing individual test cases. A test case answers a single question about the code it is testing.

A test case should be able to...

- ...run completely by itself, without any human input. Unit testing is about automation.
- ...determine by itself whether the function it is testing has passed or failed, without a human interpreting the results.
- ...run in isolation, separate from any other test cases (even if they test the same functions). Each test case is an island.

Given that, let's build our first test case. We have the following requirement:

1. `toRoman` should return the Roman numeral representation for all integers `1` to `3999`.

Example 6.2. `testToRomanKnownValues`

```
class KnownValues(unittest.TestCase):                          (1)
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
```

```
                 (6, 'VI'),
                 (7, 'VII'),
                 (8, 'VIII'),
                 (9, 'IX'),
                 (10, 'X'),
                 (50, 'L'),
                 (100, 'C'),
                 (500, 'D'),
                 (1000, 'M'),
                 (31, 'XXXI'),
                 (148, 'CXLVIII'),
                 (294, 'CCXCIV'),
                 (312, 'CCCXII'),
                 (421, 'CDXXI'),
                 (528, 'DXXVIII'),
                 (621, 'DCXXI'),
                 (782, 'DCCLXXXII'),
                 (870, 'DCCCLXX'),
                 (941, 'CMXLI'),
                 (1043, 'MXLIII'),
                 (1110, 'MCX'),
                 (1226, 'MCCXXVI'),
                 (1301, 'MCCCI'),
                 (1485, 'MCDLXXXV'),
                 (1509, 'MDIX'),
                 (1607, 'MDCVII'),
                 (1754, 'MDCCLIV'),
                 (1832, 'MDCCCXXXII'),
                 (1993, 'MCMXCIII'),
                 (2074, 'MMLXXIV'),
                 (2152, 'MMCLII'),
                 (2212, 'MMCCXII'),
                 (2343, 'MMCCCXLIII'),
                 (2499, 'MMCDXCIX'),
                 (2574, 'MMDLXXIV'),
                 (2646, 'MMDCXLVI'),
                 (2723, 'MMDCCXXIII'),
                 (2892, 'MMDCCCXCII'),
                 (2975, 'MMCMLXXV'),
                 (3051, 'MMMLI'),
                 (3185, 'MMMCLXXXV'),
                 (3250, 'MMMCCL'),
                 (3313, 'MMMCCCXIII'),
                 (3408, 'MMMCDVIII'),
                 (3501, 'MMMDI'),
                 (3610, 'MMMDCX'),
                 (3743, 'MMMDCCXLIII'),
                 (3844, 'MMMDCCCXLIV'),
                 (3888, 'MMMDCCCLXXXVIII'),
                 (3940, 'MMMCMXL'),
                 (3999, 'MMMCMXCIX'))                         (2)

    def testToRomanKnownValues(self):                          (3)
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)                    (4) (5)
            self.assertEqual(numeral, result)                  (6)
```

(1) To write a test case, first subclass the `TestCase` class of the `unittest` module. This class provides many useful methods which you can use in your test case to test specific conditions.

(2) This is a list of integer/numeral pairs that I verified manually. It includes the lowest ten

numbers, the highest number, every number that translates to a single−character Roman numeral, and a random sampling of other valid numbers. The point of a unit test is not to test every possible input, but to test a representative sample.

(3) Every individual test is its own method, which must take no parameters and return no value. If the method exits normally without raising an exception, the test is considered passed; if the method raises an exception, the test is considered failed.

(4) Here we call the actual `toRoman` function. (Well, the function hasn't be written yet, but once it is, this is the line that will call it.) Notice that we have now defined the API for the `toRoman` function: it must take an integer (the number to convert) and return a string (the Roman numeral representation). If the API is different than that, this test is considered failed.

(5) Also notice that we are not trapping any exceptions when we call `toRoman`. This is intentional. `toRoman` shouldn't raise an exception when we call it with valid input, and these input values are all valid. If `toRoman` raises an exception, this test is considered failed.

(6) Assuming the `toRoman` function was defined correctly, called correctly, completed successfully, and returned a value, the last step is to check whether it returned the *right* value. This is a common question, and the `TestCase` class provides a method, `assertEqual`, to check whether two values are equal. If the result returned from `toRoman` (`result`) does not match the known value we were expecting (`numeral`), `assertEqual` will raise an exception and the test will fail. If the two values are equal, `assertEqual` will do nothing. If every value returned from `toRoman` matches the known value we expect, `assertEqual` never raises an exception, so `testToRomanKnownValues` eventually exits normally, which means `toRoman` has passed this test.

6.4. Testing for failure

It is not enough to test that our functions succeed when given good input; we must also test that they fail when given bad input. And not just any sort of failure; they must fail in the way we expect.

Remember our other requirements for `toRoman`:

2. `toRoman` should fail when given an integer outside the range `1` to `3999`.
3. `toRoman` should fail when given a non−integer decimal.

In Python, functions indicate failure by raising exceptions, and the `unittest` module provides methods for testing whether a function raises a particular exception when given bad input.

Example 6.3. Testing bad input to `toRoman`

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000) (1)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)    (2)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, −1)

    def testDecimal(self):
        """toRoman should fail with non−integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)  (3)
```

(1) The `TestCase` class of the `unittest` provides the `assertRaises` method, which takes the following arguments: the exception we're expecting, the function we're testing, and the arguments we're passing that function. (If the function we're testing takes more than one argument, pass them all to `assertRaises`, in order, and it will pass them right along to the function we're testing.) Pay close attention to what we're doing here: instead of calling `toRoman` directly and manually checking that it raises a particular exception (by wrapping it in a `try...except` block), `assertRaises` has encapsulated all of that for us. All we do is give it the exception (`roman.OutOfRangeError`), the function (`toRoman`), and `toRoman`'s arguments (`4000`), and `assertRaises` takes care of calling `toRoman` and checking to make sure that it raises `roman.OutOfRangeError`. (Have I mentioned recently how handy it is that everything in Python is an object, including functions and exceptions?)

(2) Along with testing numbers that are too large, we need to test numbers that are too small. Remember, Roman numerals cannot express `0` or negative numbers, so we have a test case for each of those (`testZero` and `testNegative`). In `testZero`, we are testing that `toRoman` raises a `roman.OutOfRangeError` exception when called with `0`; if it does *not* raise a `roman.OutOfRangeError` (either because it returns an actual value, or because it raises some other exception), this test is considered failed.

(3) Requirement #3 specifies that `toRoman` cannot accept a non−integer decimal, so here we test to make sure that `toRoman` raises a `roman.NotIntegerError` exception when called with a decimal (`0.5`). If `toRoman` does not raise a `roman.NotIntegerError`, this test is considered failed.

The next two requirements are similar to the first three, except they apply to `fromRoman` instead of `toRoman`:

4. `fromRoman` should take a valid Roman numeral and return the number that it represents.
5. `fromRoman` should fail when given an invalid Roman numeral.

Requirement #4 is handled in the same way as requirement #1, iterating through a sampling of known values and testing each in turn. Requirement #5 is handled in the same way as requirements #2 and #3, by testing a series of bad inputs and making sure `fromRoman` raises the appropriate exception.

Example 6.4. Testing bad input to `fromRoman`

```
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s) (1)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
```

(1) Not much new to say about these; the pattern is exactly the same as the one we used to test bad input to `toRoman`. I will briefly note that we have another exception: `roman.InvalidRomanNumeralError`. That makes a total of three custom exceptions that will need to be defined in `roman.py` (along with `roman.OutOfRangeError` and `roman.NotIntegerError`). We'll see how to define these custom exceptions when we actually start writing `roman.py`, later in this chapter.

6.5. Testing for sanity

Often, you will find that a unit of code contains a set of reciprocal functions, usually in the form of conversion functions where one converts A to B and the other converts B to A. In these cases, it is useful to create a "sanity check" to make sure that you can convert A to B and back to A without losing decimal precision, incurring rounding errors, or triggering any other sort of bug.

Consider this requirement:

6. If you take a number, convert it to Roman numerals, then convert that back to a number, you should end up with the number you started with. So `fromRoman(toRoman(n)) == n` for all `n` in `1..3999`.

Example 6.5. Testing `toRoman` against `fromRoman`

```
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):        (1) (2)
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result) (3)
```

(1) We've seen the `range` function before, but here it is called with two arguments, which returns a list of integers starting at the first argument (`1`) and counting consecutively up to *but not including* the second argument (`4000`). Thus, `1..3999`, which is the valid range for converting to Roman numerals.

(2) I just wanted to mention in passing that `integer` is not a keyword in Python; here it's just a variable name like any other.

(3) The actual testing logic here is straightforward: take a number (`integer`), convert it to a Roman numeral (`numeral`), then convert it back to a number (`result`) and make sure you end up with the same number you started with. If not, `assertEqual` will raise an exception and the test will immediately be considered failed. If all the numbers match, `assertEqual` will always return silently, the entire `testSanity` method will eventually return silently, and the test will be considered passed.

The last two requirements are different from the others because they seem both arbitrary and trivial:

7. `toRoman` should always return a Roman numeral using uppercase letters.
8. `fromRoman` should only accept uppercase Roman numerals (*i.e.* it should fail when given lowercase input).

In fact, they are somewhat arbitrary. We could, for instance, have stipulated that `fromRoman` accept lowercase and mixed case input. But they are not completely arbitrary; if `toRoman` is always returning uppercase output, then `fromRoman` must at least accept uppercase input, or our "sanity check" (requirement #6) would fail. The fact that it *only* accepts uppercase input is arbitrary, but as any systems integrator will tell you, case always matters, so it's worth specifying the behavior up front. And if it's worth specifying, it's worth testing.

Example 6.6. Testing for case

```
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper()) (1)
```

```
    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())           (2) (3)
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())
```

(1) The most interesting thing about this test case is all the things it doesn't test. It doesn't test that the value returned from `toRoman` is right or even consistent; those questions are answered by separate test cases. We have a whole test case just to test for uppercase−ness. You might be tempted to combine this with the sanity check, since both run through the entire range of values and call `toRoman`.[13] But that would violate one of our fundamental rules: each test case should answer only a single question. Imagine that you combined this case check with the sanity check, and then that test case failed. You would have to do further analysis to figure out which part of the test case failed to determine what the problem was. If you have to analyze the results of your unit testing just to figure out what they mean, it's a sure sign that you've mis−designed your test cases.

(2) There's a similar lesson to be learned here: even though "we know" that `toRoman` always returns uppercase, we are explicitly converting its return value to uppercase here to test that `fromRoman` accepts uppercase input. Why? Because the fact that `toRoman` always returns uppercase is an independent requirement. If we changed that requirement so that, for instance, it always returned lowercase, the `testToRomanCase` test case would have to change, but this test case would still work. This was another of our fundamental rules: each test case must be able to work in isolation from any of the others. Every test case is an island.

(3) Note that we're not assigning the return value of `fromRoman` to anything. This is legal syntax in Python; if a function returns a value but nobody's listening, Python just throws away the return value. In this case, that's what we want. This test case doesn't test anything about the return value; it just tests that `fromRoman` accepts the uppercase input without raising an exception.

6.6. `roman.py`, stage 1

Now that our unit test is complete, it's time to start writing the code that our test cases are attempting to test. We're going to do this in stages, so we can see all the unit tests fail, then watch them pass one by one as we fill in the gaps in `roman.py`.

Example 6.7. `roman1.py`

If you have not already done so, you can download this and other examples used in this book.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass                (1)
class OutOfRangeError(RomanError): pass          (2)
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass (3)

def toRoman(n):
    """convert integer to Roman numeral"""
    pass                                         (4)

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

(1)

This is how you define your own custom exceptions in Python. Exceptions are classes, and you create your own by subclassing existing exceptions. It is strongly recommended (but not required) that you subclass `Exception`, which is the base class that all built−in exceptions inherit from. Here I am defining `RomanError` (inherited from `Exception`) to act as the base class for all my other custom exceptions to follow. This is a matter of style; I could just as easily have inherited each individual exception from the `Exception` class directly.

(2) The `OutOfRangeError` and `NotIntegerError` exceptions will eventually be used by `toRoman` to flag various forms of invalid input, as specified in `ToRomanBadInput`.

(3) The `InvalidRomanNumeralError` exception will eventually be used by `fromRoman` to flag invalid input, as specified in `FromRomanBadInput`.

(4) At this stage, we want to define the API of each of our functions, but we don't want to code them yet, so we stub them out using the Python reserved word `pass`.

Now for the big moment (drum roll please): we're finally going to run our unit test against this stubby little module. At this point, every test case should fail. In fact, if any test case passes in stage 1, we should go back to `romantest.py` and re−evaluate why we coded a test so useless that it passes with do−nothing functions.

Run `romantest1.py` with the `−v` command−line option, which will give more verbose output so we can see exactly what's going on as each test case runs. With any luck, your output should look like this:

Example 6.8. Output of `romantest1.py` against `roman1.py`

```
fromRoman should only accept uppercase input ... ERROR
toRoman should always return uppercase ... ERROR
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... FAIL
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non−integer input ... FAIL
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL

======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 154, in testFromRomanCase
    roman1.fromRoman(numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 148, in testToRomanCase
    self.assertEqual(numeral, numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
```

```
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 127, in testRepeatedPairs
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 93, in testToRomanKnownValues
    self.assertEqual(numeral, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: I != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 116, in testDecimal
    self.assertRaises(roman1.NotIntegerError, roman1.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 112, in testNegative
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
```

```
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 104, in testTooLarge
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input                                  (1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 108, in testZero
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError                                         (2)
----------------------------------------------------------------------
Ran 12 tests in 0.040s                                                  (3)

FAILED (failures=10, errors=2)                                          (4)
```

(1) Running the script runs `unittest.main()`, which runs each test case, which is to say each method defined in each class within `romantest.py`. For each test case, it prints out the `doc string` of the method and whether that test passed or failed. As expected, none of our test cases passed.

(2) For each failed test case, `unittest` displays the trace information showing exactly what happened. In this case, our call to `assertRaises` (also called `failUnlessRaises`) raised an `AssertionError` because it was expecting `toRoman` to raise an `OutOfRangeError` and it didn't.

(3) After the detail, `unittest` displays a summary of how many tests were performed and how long it took.

(4) Overall, the unit test failed because at least one test case did not pass. When a test case doesn't pass, `unittest` distinguishes between failures and errors. A failure is a call to an `assertXYZ` method, like `assertEqual` or `assertRaises`, that fails because the asserted condition is not true or the expected exception was not raised. An error is any other sort of exception raised in the code we're testing or the unit test case itself. For instance, the `testFromRomanCase` method ("`fromRoman` should only accept uppercase input") was an error, because the call to `numeral.upper()` raised an `AttributeError` exception, because `toRoman` was supposed to return a string but didn't. But `testZero` ("`toRoman` should fail with 0 input") was a failure, because the call to `fromRoman` did not raise the `InvalidRomanNumeral` exception that `assertRaises` was looking for.

6.7. `roman.py`, stage 2

Now that we have the framework of our `roman` module laid out, it's time to start writing code and passing test cases.

Example 6.9. `roman2.py`

If you have not already done so, you can download this and other examples used in this book.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
```

```
#Define digit mapping
romanNumeralMap = (('M',  1000), (1)
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:      (2)
            result += numeral
            n -= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

(1) `romanNumeralMap` is a tuple of tuples which defines three things:

1. The character representations of the most basic Roman numerals. Note that this is not just the single−character Roman numerals; we're also defining two−character pairs like `CM` ("one hundred less than one thousand"); this will make our `toRoman` code simpler later.
2. The order of the Roman numerals. They are listed in descending value order, from `M` all the way down to `I`.
3. The value of each Roman numeral. Each inner tuple is a pair of `(numeral, value)`.

(2) Here's where our rich data structure pays off, because we don't need any special logic to handle the subtraction rule. To convert to Roman numerals, we simply iterate through `romanNumeralMap` looking for the largest integer value less than or equal to our input. Once found, we add the Roman numeral representation to the end of the output, subtract the corresponding integer value from the input, lather, rinse, repeat.

Example 6.10. How `toRoman` works

If you're not clear how `toRoman` works, add a `print` statement to the end of the `while` loop:

```
        while n >= integer:
            result += numeral
            n -= integer
            print 'subtracting', integer, 'from input, adding', numeral, 'to output'
```

```
>>> import roman2
>>> roman2.toRoman(1424)
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

So toRoman appears to work, at least in our manual spot check. But will it pass the unit testing? Well no, not entirely.

Example 6.11. Output of romantest2.py against roman2.py

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok                  (1)
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok       (2)
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL            (3)
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL
```

(1) toRoman does, in fact, always return uppercase, because our romanNumeralMap defines the Roman numeral representations as uppercase. So this test passes already.

(2) Here's the big news: this version of the toRoman function passes the known values test. Remember, it's not comprehensive, but it does put the function through its paces with a variety of good inputs, including inputs that produce every single−character Roman numeral, the largest possible input (3999), and the input that produces the longest possible Roman numeral (3888). At this point, we can be reasonably confident that the function works for any good input value you could throw at it.

(3) However, the function does not "work" for bad values; it fails every single bad input test. That makes sense, because we didn't include any checks for bad input. Those test cases look for specific exceptions to be raised (via assertRaises), and we're never raising them. We'll do that in the next stage.

Here's the rest of the output of the unit test, listing the details of all the failures. We're down to 10.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 156, in testFromRomanCase
    roman2.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 127, in testRepeatedPairs
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
```

```
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 116, in testDecimal
    self.assertRaises(roman2.NotIntegerError, roman2.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 112, in testNegative
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 104, in testTooLarge
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 108, in testZero
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
```

```
----------------------------------------------------------------------
Ran 12 tests in 0.320s

FAILED (failures=10)
```

6.8. `roman.py`, stage 3

Now that `toRoman` behaves correctly with good input (integers from `1` to `3999`), it's time to make it behave correctly with bad input (everything else).

Example 6.12. `roman3.py`

If you have not already done so, you can download this and other examples used in this book.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):                                             (1)
        raise OutOfRangeError, "number out of range (must be 1..3999)" (2)
    if int(n) <> n:                                                    (3)
        raise NotIntegerError, "decimals can not be converted"

    result = ""                                                        (4)
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

(1) This is a nice Pythonic shortcut: multiple comparisons at once. This is equivalent to `if not ((0 < n) and (n < 4000))`, but it's much easier to read. This is our range check, and it should catch inputs that are too large, negative, or zero.

(2) You raise exceptions yourself with the `raise` statement. You can raise any of the built−in exceptions, or you can raise any of your custom exceptions that you've defined. The second parameter, the error message, is optional; if given, it is displayed in the traceback that is printed if the exception is never handled.

(3) This is our decimal check. Decimals can not be converted to Roman numerals.

(4) The rest of the function is unchanged.

Example 6.13. Watching `toRoman` handle bad input

```
>>> import roman3
>>> roman3.toRoman(4000)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 27, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
>>> roman3.toRoman(1.5)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 29, in toRoman
    raise NotIntegerError, "decimals can not be converted"
NotIntegerError: decimals can not be converted
```

Example 6.14. Output of `romantest3.py` against `roman3.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok (1)
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... ok        (2)
toRoman should fail with negative input ... ok           (3)
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

(1) `toRoman` still passes the known values test, which is comforting. All the tests that passed in stage 2 still pass, so our latest code hasn't broken anything.

(2) More exciting is the fact that all of our bad input tests now pass. This test, `testDecimal`, passes because of the `int(n) <> n` check. When a decimal is passed to `toRoman`, the `int(n) <> n` check notices it and raises the `NotIntegerError` exception, which is what `testDecimal` is looking for.

(3) This test, `testNegative`, passes because of the `not (0 < n < 4000)` check, which raises an `OutOfRangeError` exception, which is what `testNegative` is looking for.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 156, in testFromRomanCase
    roman3.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 127, in testRepeatedPairs
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
----------------------------------------------------------------------
Ran 12 tests in 0.401s

FAILED (failures=6) (1)
```

(1) We're down to 6 failures, and all of them involve `fromRoman`: the known values test, the three separate bad input tests, the case check, and the sanity check. That means that `toRoman` has passed all the tests it can pass by itself. (It's involved in the sanity check, but that also requires that `fromRoman` be written, which it isn't yet.) Which means that we must stop coding `toRoman` now. No tweaking, no twiddling, no extra checks "just in case". Stop. Now. Back away from the keyboard.

> **Note: Know when to stop coding**
> The most important thing that comprehensive unit testing can tell you is when to stop coding. When all the unit tests for a function pass, stop coding the function. When all the unit tests for an entire module pass, stop coding the module.

6.9. `roman.py`, stage 4

Now that `toRoman` is done, it's time to start coding `fromRoman`. Thanks to our rich data structure that maps individual Roman numerals to integer values, this is no more difficult than the `toRoman` function.

Example 6.15. `roman4.py`

If you have not already done so, you can download this and other examples used in this book.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

# toRoman function omitted for clarity (it hasn't changed)

def fromRoman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral: (1)
            result += integer
            index += len(numeral)
    return result
```

(1) The pattern here is the same as `toRoman`. We iterate through our Roman numeral data structure (a tuple of tuples), and instead of matching the highest integer values as often as possible, we match the "highest" Roman numeral character strings as often as possible.

Example 6.16. How `fromRoman` works

If you're not clear how `fromRoman` works, add a `print` statement to the end of the `while` loop:

```
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print 'found', numeral, ', adding', integer
```

```
>>> import roman4
>>> roman4.fromRoman('MCMLXXII')
found M , adding 1000
found CM , adding 900
found L , adding 50
found X , adding 10
found X , adding 10
found I , adding 1
found I , adding 1
1972
```

Example 6.17. Output of `romantest4.py` against `roman4.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... ok (1)
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok                  (2)
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

(1) Two pieces of exciting news here. The first is that `fromRoman` works for good input, at least for all the known values we test.

(2) The second is that our sanity check also passed. Combined with the known values tests, we can be reasonably sure that both `toRoman` and `fromRoman` work properly for all possible good values. (This is not guaranteed; it is theoretically possible that `toRoman` has a bug that produces the wrong Roman numeral for some particular set of inputs, *and* that `fromRoman` has a reciprocal bug that produces the same wrong integer values for exactly that set of Roman numerals that `toRoman` generated incorrectly. Depending on your application and your requirements, this possibility may bother you; if so, write more comprehensive test cases until it doesn't bother you.)

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 156, in testFromRomanCase
    roman4.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 127, in testRepeatedPairs
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
```

```
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 12 tests in 1.222s

FAILED (failures=4)
```

6.10. `roman.py`, stage 5

Now that `fromRoman` works properly with good input, it's time to fit in the last piece of the puzzle: making it work properly with bad input. That means finding a way to look at a string and determine if it's a valid Roman numeral. This is inherently more difficult than validating numeric input in `toRoman`, but we have a powerful tool at our disposal: regular expressions.

If you're not familiar with regular expressions and didn't read *Regular expressions 101*, now would be a good time.

As we saw at the beginning of this chapter, there are several simple rules for constructing a Roman numeral. The first is that the thousands place, if any, is represented by a series of `M` characters.

Example 6.18. Checking for thousands

```
>>> import re
>>> pattern = '^M?M?M?$'       (1)
>>> re.search(pattern, 'M')    (2)
<SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')   (3)
<SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')  (4)
<SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM') (5)
>>> re.search(pattern, '')     (6)
<SRE_Match object at 0106F4A8>
```

(1) This pattern has three parts:

1. `^` − match what follows only at the beginning of the string. If this were not specified, the pattern would match no matter where the `M` characters were, which is not what we want. We want to make sure that the `M` characters, if they're there, are at the beginning of the string.
2. `M?` − optionally match a single `M` character. Since this is repeated three times, we're matching anywhere from 0 to 3 `M` characters in a row.
3. `$` − match what precedes only at the end of the string. When combined with the `^` character at the beginning, this means that the pattern must match the entire string, with no other characters before or after the `M` characters.

(2) The essense of the `re` module is the `search` function, which takes a regular expression (`pattern`) and a string (`'M'`) to try to match against the regular expression. If a match is found, `search` returns an

object which has various methods to describe the match; if no match is found, `search` returns `None`, the Python null value. We won't go into detail about the object that `search` returns (although it's very interesting), because all we care about at the moment is whether the pattern matches, which we can tell by just looking at the return value of `search`. `'M'` matches this regular expression, because the first optional `M` matches and the second and third optional `M` characters are ignored.

(3) `'MM'` matches because the first and second optional `M` characters match and the third `M` is ignored.

(4) `'MMM'` matches because all three `M` characters match.

(5) `'MMMM'` does not match. All three `M` characters match, but then the regular expression insists on the string ending (because of the `$` character), and the string doesn't end yet (because of the fourth `M`). So `search` returns `None`.

(6) Interestingly, an empty string also matches this regular expression, since all the `M` characters are optional. Keep this fact in the back of your mind; it will become more important in the next section.

The hundreds place is more difficult than the thousands, because there are several mutually exclusive ways it could be expressed, depending on its value.

- `100` = `C`
- `200` = `CC`
- `300` = `CCC`
- `400` = `CD`
- `500` = `D`
- `600` = `DC`
- `700` = `DCC`
- `800` = `DCCC`
- `900` = `CM`

So there are four possible patterns:

1. `CM`
2. `CD`
3. 0 to 3 `C` characters (0 if the hundreds place is 0)
4. `D`, followed by 0 to 3 `C` characters

The last two patterns can be combined:

- an optional `D`, followed by 0 to 3 `C` characters

Example 6.19. Checking for hundreds

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$' (1)
>>> re.search(pattern, 'MCM')            (2)
<SRE_Match object at 01070390>
>>> re.search(pattern, 'MD')             (3)
<SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC')         (4)
<SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC')           (5)
>>> re.search(pattern, '')               (6)
<SRE_Match object at 01071D98>
```

(1) This pattern starts out the same as our previous one, checking for the beginning of the string (`^`), then the thousands place (`M?M?M?`). Then we have the new part, in parentheses, which defines a set of three mutually

exclusive patterns, separated by vertical bars: `CM`, `CD`, and `D?C?C?C?` (which is an optional `D` followed by 0 to 3 optional `C` characters). The regular expression parser checks for each of these patterns in order (from left to right), takes the first one that matches, and ignores the rest.

(2) `'MCM'` matches because the first `M` matches, the second and third `M` characters are ignored, and the `CM` matches (so the `CD` and `D?C?C?C?` patterns are never even considered). `MCM` is the Roman numeral representation of `1900`.

(3) `'MD'` matches because the first `M` matches, the second and third `M` characters are ignored, and the `D?C?C?C?` pattern matches `D` (each of the 3 `C` characters are optional and are ignored). `MD` is the Roman numeral representation of `1500`.

(4) `'MMMCCC'` matches because all 3 `M` characters match, and the `D?C?C?C?` pattern matches `CCC` (the `D` is optional and is ignored). `MMMCCC` is the Roman numeral representation of `3300`.

(5) `'MCMC'` does not match. The first `M` matches, the second and third `M` characters are ignored, and the `CM` matches, but then the `$` does not match because we're not at the end of the string yet (we still have an unmatched `C` character). The `C` does *not* match as part of the `D?C?C?C?` pattern, because the mutually exclusive `CM` pattern has already matched.

(6) Interestingly, an empty string still matches this pattern, because all the `M` characters are optional and ignored, and the empty string matches the `D?C?C?C?` pattern where all the characters are optional and ignored.

Whew! See how quickly regular expressions can get nasty? And we've only covered the thousands and hundreds places. (Later in this chapter, we'll see a slightly different syntax for writing regular expressions that, while just as complicated, at least allows some in−line documentation of the different sections of the expression.) Luckily, if you followed all that, the tens and ones places are easy, because they're exactly the same pattern.

Example 6.20. `roman5.py`

If you have not already done so, you can download this and other examples used in this book.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError, "number out of range (must be 1..3999)"
    if int(n) <> n:
```

```
        raise NotIntegerError, "decimals can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' (1)

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not re.search(romanNumeralPattern, s):                                    (2)
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

(1) This is just a continuation of the pattern we saw that handled the thousands and hundreds place. The tens places is either `XC` (`90`), `XL` (`40`), or an optional `L` followed by 0 to 3 optional `X` characters. The ones place is either `IX` (`9`), `IV` (`4`), or an optional `V` followed by 0 to 3 optional `I` characters.

(2) Having encoded all that logic into our regular expression, the code to check for invalid Roman numerals becomes trivial. If `re.search` returns an object, then the regular expression matched and our input is valid; otherwise, our input is invalid.

At this point, you are allowed to be skeptical that that big ugly regular expression could possibly catch all the types of invalid Roman numerals. But don't take my word for it, look at the results:

Example 6.21. Output of `romantest5.py` against `roman5.py`

```
fromRoman should only accept uppercase input ... ok          (1)
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... ok      (2)
fromRoman should fail with repeated pairs of numerals ... ok (3)
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 2.864s

OK                                                           (4)
```

(1) One thing I didn't mention about regular expressions is that, by default, they are case−sensitive. Since our regular expression `romanNumeralPattern` was expressed in uppercase characters, our `re.search` check will reject any input that isn't completely uppercase. So our uppercase input test passes.

(2) More importantly, our bad input tests pass. For instance, the malformed antecedents test checks cases like `MCMC`. As we've seen, this does not match our regular expression, so `fromRoman` raises an `InvalidRomanNumeralError` exception, which is what the malformed antecedents test case is looking for, so the test passes.

(3) In fact, all the bad input tests pass. This regular expression catches everything we could think of when we made our test cases.

(4) And the anticlimax award of the year goes to the word "`OK`", which is printed by the `unittest` module when all the tests pass.

Note: What to do when all your tests pass
When all your tests pass, stop coding.

6.11. Handling bugs

Despite your best efforts to write comprehensive unit tests, bugs happen. What do I mean by "bug"? A bug is a test case you haven't written yet.

Example 6.22. The bug

```
>>> import roman5
>>> roman5.fromRoman("") (1)
0
```

(1) Remember in the previous section when we kept seeing that an empty string would match the regular expression we were using to check for valid Roman numerals? Well, it turns out that this is still true for the final version of the regular expression. And that's a bug; we want an empty string to raise an `InvalidRomanNumeralError` exception just like any other sequence of characters that don't represent a valid Roman numeral.

After reproducing the bug, and before fixing it, you should write a test case that fails, thus illustrating the bug.

Example 6.23. Testing for the bug (`romantest61.py`)

```
class FromRomanBadInput(unittest.TestCase):

    # previous test cases omitted for clarity (they haven't changed)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, "") (1)
```

(1) Pretty simple stuff here. Call `fromRoman` with an empty string and make sure it raises an `InvalidRomanNumeralError` exception. The hard part was finding the bug; now that we know about it, testing for it is the easy part.

Since our code has a bug, and we now have a test case that tests this bug, the test case will fail:

Example 6.24. Output of `romantest61.py` against `roman61.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... FAIL
fromRoman should fail with malformed antecedents ... ok
```

```
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

======================================================================
FAIL: fromRoman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage6\romantest61.py", line 137, in testBlank
    self.assertRaises(roman61.InvalidRomanNumeralError, roman61.fromRoman, "")
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 13 tests in 2.864s

FAILED (failures=1)
```

Now we can fix the bug.

Example 6.25. Fixing the bug (`roman62.py`)

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s: (1)
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

(1) Only two lines of code are required: an explicit check for an empty string, and a `raise` statement.

Example 6.26. Output of `romantest62.py` against `roman62.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok (1)
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

```
----------------------------------------------------------------------
Ran 13 tests in 2.834s

OK (2)
```

(1) The blank string test case now passes, so the bug is fixed.

(2) All the other test cases still pass, which means that this bug fix didn't break anything else. Stop coding.

Coding this way does not make fixing bugs any easier. Simple bugs (like this one) require simple test cases; complex bugs will require complex test cases. In a testing−centric environment, it may *seem* like it takes longer to fix a bug, since you have to articulate in code exactly what the bug is (to write the test case), then fix the bug itself. Then if the test case doesn't pass right away, you have to figure out whether the fix was wrong, or whether the test case itself has a bug in it. However, in the long run, this back−and−forth between test code and code tested pays for itself, because it makes it more likely that bugs are fixed correctly the first time. Also, since you can easily re−run *all* the test cases along with your new one, you are much less likely to break old code when fixing new code. Today's unit test is tomorrow's regression test.

6.12. Handling changing requirements

Despite your best efforts to pin your customers to the ground and extract exact requirements from them on pain of horrible nasty things involving scissors and hot wax, requirements will change. Most customers don't know what they want until they see it, and even if they do, they aren't that good at articulating what they want precisely enough to be useful. And even if they do, they'll want more in the next release anyway. So be prepared to update your test cases as requirements change.

Suppose, for instance, that we wanted to expand the range of our Roman numeral conversion functions. Remember the rule that said that no character could be repeated more than three times? Well, the Romans were willing to make an exception to that rule by having 4 `M` characters in a row to represent `4000`. If we make this change, we'll be able to expand our range of convertible numbers from `1..3999` to `1..4999`. But first, we need to make some changes to our test cases.

Example 6.27. Modifying test cases for new requirements (`romantest71.py`)

If you have not already done so, you can download this and other examples used in this book.

```
import roman71
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
```

```
                 (294, 'CCXCIV'),
                 (312, 'CCCXII'),
                 (421, 'CDXXI'),
                 (528, 'DXXVIII'),
                 (621, 'DCXXI'),
                 (782, 'DCCLXXXII'),
                 (870, 'DCCCLXX'),
                 (941, 'CMXLI'),
                 (1043, 'MXLIII'),
                 (1110, 'MCX'),
                 (1226, 'MCCXXVI'),
                 (1301, 'MCCCI'),
                 (1485, 'MCDLXXXV'),
                 (1509, 'MDIX'),
                 (1607, 'MDCVII'),
                 (1754, 'MDCCLIV'),
                 (1832, 'MDCCCXXXII'),
                 (1993, 'MCMXCIII'),
                 (2074, 'MMLXXIV'),
                 (2152, 'MMCLII'),
                 (2212, 'MMCCXII'),
                 (2343, 'MMCCCXLIII'),
                 (2499, 'MMCDXCIX'),
                 (2574, 'MMDLXXIV'),
                 (2646, 'MMDCXLVI'),
                 (2723, 'MMDCCXXIII'),
                 (2892, 'MMDCCCXCII'),
                 (2975, 'MMCMLXXV'),
                 (3051, 'MMMLI'),
                 (3185, 'MMMCLXXXV'),
                 (3250, 'MMMCCL'),
                 (3313, 'MMMCCCXIII'),
                 (3408, 'MMMCDVIII'),
                 (3501, 'MMMDI'),
                 (3610, 'MMMDCX'),
                 (3743, 'MMMDCCXLIII'),
                 (3844, 'MMMDCCCXLIV'),
                 (3888, 'MMMDCCCLXXXVIII'),
                 (3940, 'MMMCMXL'),
                 (3999, 'MMMCMXCIX'),
                 (4000, 'MMMM'),                                        (1)
                 (4500, 'MMMMD'),
                 (4888, 'MMMMDCCCLXXXVIII'),
                 (4999, 'MMMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 5000) (2)

    def testZero(self):
```

```
        """toRoman should fail with 0 input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, −1)

    def testDecimal(self):
        """toRoman should fail with non−integer input"""
        self.assertRaises(roman71.NotIntegerError, roman71.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):     (3)
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, "")

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 5000):                                     (4)
            numeral = roman71.toRoman(integer)
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            roman71.fromRoman(numeral.upper())
            self.assertRaises(roman71.InvalidRomanNumeralError,
                              roman71.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

(1) The existing known values don't change (they're all still reasonable values to test), but we need to add a few more in the `4000` range. Here I've included `4000` (the shortest), `4500` (the second shortest), `4888` (the longest), and `4999` (the largest).

(2) The definition of "large input" has changed. This test used to call `toRoman` with `4000` and expect an error; now that `4000-4999` are good values, we need to bump this up to `5000`.

(3) The definition of "too many repeated numerals" has also changed. This test used to call `fromRoman` with `'MMMM'` and expect an error; now that `MMMM` is considered a valid Roman numeral, we need to bump this up to `'MMMMM'`.

(4) The sanity check and case checks loop through every number in the range, from `1` to `3999`. Since the range has now expanded, these `for` loops need to be updated as well to go up to `4999`.

Now our test cases are up to date with our new requirements, but our code is not, so we expect several of our test cases to fail.

Example 6.28. Output of `romantest71.py` against `roman71.py`

```
fromRoman should only accept uppercase input ... ERROR        (1)
toRoman should always return uppercase ... ERROR
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ERROR (2)
toRoman should give known result with known input ... ERROR   (3)
fromRoman(toRoman(n))==n for all n ... ERROR                  (4)
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

(1) Our case checks now fail because they loop from `1` to `4999`, but `toRoman` only accepts numbers from `1` to `3999`, so it will fail as soon the test case hits `4000`.

(2) The `fromRoman` known values test will fail as soon as it hits `'MMMM'`, because `fromRoman` still thinks this is an invalid Roman numeral.

(3) The `toRoman` known values test will fail as soon as it hits `4000`, because `toRoman` still thinks this is out of range.

(4) The sanity check will also fail as soon as it hits `4000`, because `toRoman` still thinks this is out of range.

```
======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 161, in testFromRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 155, in testToRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 102, in testFromRomanKnownValues
```

```
    result = roman71.fromRoman(numeral)
  File "roman71.py", line 47, in fromRoman
    raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
InvalidRomanNumeralError: Invalid Roman numeral: MMMM
======================================================================
ERROR: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 96, in testToRomanKnownValues
    result = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 147, in testSanity
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
----------------------------------------------------------------------
Ran 13 tests in 2.213s

FAILED (errors=5)
```

Now that we have test cases that fail due to the new requirements, we can think about fixing the code to bring it in line with the test cases. (One thing that takes some getting used to when you first start coding unit tests is that the code being tested is never "ahead" of the test cases. While it's behind, you still have some work to do, and as soon as it catches up to the test cases, you stop coding.)

Example 6.29. Coding the new requirements (`roman72.py`)

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
```

```
    if not (0 < n < 5000):                                                  (1)
        raise OutOfRangeError, "number out of range (must be 1..4999)"
    if int(n) <> n:
        raise NotIntegerError, "decimals can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' (2)

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

(1) `toRoman` only needs one small change, in the range check. Where we used to check `0 < n < 4000`, we now check `0 < n < 5000`. And we change the error message that we `raise` to reflect the new acceptable range (`1..4999` instead of `1..3999`). We don't need to make any changes to the rest of the function; it handles the new cases already. (It merrily adds `'M'` for each thousand that it finds; given `4000`, it will spit out `'MMMM'`. The only reason it didn't do this before is that we explicitly stopped it with the range check.)

(2) We don't need to make any changes to `fromRoman` at all. The only change is to `romanNumeralPattern`; if you look closely, you'll notice that we added another optional `M` in the first section of the regular expression. This will allow up to 4 `M` characters instead of 3, meaning we will allow the Roman numeral equivalents of `4999` instead of `3999`. The actual `fromRoman` function is completely general; it just looks for repeated Roman numeral characters and adds them up, without caring how many times they repeat. The only reason it didn't handle `'MMMM'` before is that we explicitly stopped it with the regular expression pattern matching.

You may be skeptical that these two small changes are all that we need. Hey, don't take my word for it; see for yourself:

Example 6.30. Output of `romantest72.py` against `roman72.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
```

```
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 13 tests in 3.685s

OK (1)
```

(1) All the test cases pass. Stop coding.

Comprehensive unit testing means never having to rely on a programmer who says "Trust me."

6.13. Refactoring

The best thing about comprehensive unit testing is not the feeling you get when all your test cases finally pass, or even the feeling you get when someone else blames you for breaking their code and you can actually *prove* that you didn't. The best thing about unit testing is that it gives you the freedom to refactor mercilessly.

Refactoring is the process of taking working code and making it work better. Usually, "better" means "faster", although it can also mean "using less memory", or "using less disk space", or simply "more elegantly". Whatever it means to you, to your project, in your environment, refactoring is important to the long−term health of any program.

Here, "better" means "faster". Specifically, the `fromRoman` function is slower than it needs to be, because of that big nasty regular expression that we use to validate Roman numerals. It's probably not worth trying to do away with the regular expression altogether (it would be difficult, and it might not end up any faster), but we can speed up the function by precompiling the regular expression.

Example 6.31. Compiling regular expressions

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')             (1)
<SRE_Match object at 01090490>
>>> compiledPattern = re.compile(pattern) (2)
>>> compiledPattern
<SRE_Pattern object at 00F06E28>
>>> dir(compiledPattern)                (3)
['findall', 'match', 'scanner', 'search', 'split', 'sub', 'subn']
>>> compiledPattern.search('M')         (4)
<SRE_Match object at 01104928>
```

(1) This is the syntax we've seen before: `re.search` takes a regular expression as a string (`pattern`) and a string to match against it (`'M'`). If the pattern matches, the function returns a match object which can be queried to find out exactly what matched and how.

(2) This is the new syntax: `re.compile` takes a regular expression as a string and returns a pattern object. Note there is no string to match here. Compiling a regular expression has nothing to do with matching it against any specific strings (like `'M'`); it only involves the regular expression itself.

(3) The compiled pattern object returned from `re.compile` has several useful−looking functions, including several (like `search` and `sub`) that are available directly in the `re` module.

(4) Calling the compiled pattern object's `search` function with the string `'M'` accomplishes the same thing as calling `re.search` with both the regular expression and the string `'M'`. Only much, much faster. (In fact, the `re.search` function simply compiles the regular expression and calls the resulting pattern object's `search` method for you.)

Note: Compiling regular expressions
Whenever you are going to use a regular expression more than once, you should compile it to get a pattern object, then call the methods on the pattern object directly.

Example 6.32. Compiled regular expressions in `roman81.py`

If you have not already done so, you can download this and other examples used in this book.

```
# toRoman and rest of module omitted for clarity

romanNumeralPattern = \
    re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$') (1)

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not romanNumeralPattern.search(s):                                    (2)
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

(1) This looks very similar, but in fact a lot has changed. `romanNumeralPattern` is no longer a string; it is a pattern object which was returned from `re.compile`.

(2) That means that we can call methods on `romanNumeralPattern` directly. This will be much, much faster than calling `re.search` every time. The regular expression is compiled once and stored in `romanNumeralPattern` when the module is first imported; then, every time we call `fromRoman`, we can immediately match the input string against the regular expression, without any intermediate steps occurring under the covers.

So how much faster is it to compile our regular expressions? See for yourself:

Example 6.33. Output of `romantest81.py` against `roman81.py`

```
.............          (1)
----------------------------------------------------------------------
Ran 13 tests in 3.385s (2)

OK                     (3)
```

(1) Just a note in passing here: this time, I ran the unit test *without* the `-v` option, so instead of the full `doc string` for each test, we only get a dot for each test that passes. (If a test failed, we'd get an `F`, and if it had an error, we'd get an `E`. We'd still get complete tracebacks for each failure and error, so we could track down any problems.)

(2) We ran `13` tests in `3.385` seconds, compared to `3.685` seconds without precompiling the regular expressions. That's an `8%` improvement overall, and remember that most of the time spent during the unit test is spent doing other things. (Separately, I time−tested the regular expressions by themselves, apart from the rest of the unit tests, and found that compiling this regular expression speeds up the

search by an average of 54%.) Not bad for such a simple fix.

(3) Oh, and in case you were wondering, precompiling our regular expression didn't break anything, and we just proved it.

There is one other performance optimization that I want to try. Given the complexity of regular expression syntax, it should come as no surprise that there is frequently more than one way to write the same expression. After some discussion about this module on comp.lang.python, someone suggested that I try using the {*m*,*n*} syntax for the optional repeated characters.

Example 6.34. roman82.py

If you have not already done so, you can download this and other examples used in this book.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#   re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$')

#new version
romanNumeralPattern = \
    re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$') (1)
```

(1) We have replaced M?M?M?M? with M{0,4}. Both mean the same thing: "match 0 to 4 M characters". Similarly, C?C?C? became C{0,3} ("match 0 to 3 C characters") and so forth for X and I.

This form of the regular expression is a little shorter (though not any more readable). The big question is, is it any faster?

Example 6.35. Output of romantest82.py against roman82.py

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s (1)

OK                     (2)
```

(1) Overall, the unit tests run 2% faster with this form of regular expression. That doesn't sound exciting, but remember that the search function is a small part of the overall unit test; most of the time is spent doing other things. (Separately, I time−tested just the regular expressions, and found that the search function is 11% faster with this syntax.) By precompiling the regular expression and rewriting part of it to use this new syntax, we've improved the regular expression performance by over 60%, and improved the overall performance of the entire unit test by over 10%.

(2) More important than any performance boost is the fact that the module still works perfectly. This is the freedom I was talking about earlier: the freedom to tweak, change, or rewrite any piece of it and verify that you haven't messed anything up in the process. This is not a license to endlessly tweak your code just for the sake of tweaking it; we had a very specific objective ("make fromRoman faster"), and we we able to accomplish that objective without any lingering doubts about whether we introduced new bugs in the process.

One other tweak I would like to make, and then I promise I'll stop refactoring and put this module to bed. As we've seen repeatedly, regular expressions can get pretty hairy and unreadable pretty quickly. I wouldn't like to come back to this module in six months and try to maintain it. Sure, the test cases pass, so I know that it works, but if I can't figure out *how* it works, I won't be able to add new features, fix new bugs, or otherwise maintain it. Documentation is critical, and Python provides a way of verbosely documenting your regular expressions.

Example 6.36. `roman83.py`

If you have not already done so, you can download this and other examples used in this book.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#   re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')

#new version
romanNumeralPattern = re.compile('''
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                        #            or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    ''', re.VERBOSE) (1)
```

(1) The `re.compile` function can take an optional second argument, which is a set of one or more flags that control various options about the compiled regular expression. Here we're specifying the `re.VERBOSE` flag, which tells Python that there are in−line comments within the regular expression itself. The comments and all the whitespace around them are *not* considered part of the regular expression; the `re.compile` function simply strips them all out when it compiles the expression. This new, "verbose" version is identical to the old version, but it is infinitely more readable.

Example 6.37. Output of `romantest83.py` against `roman83.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s (1)

OK                     (2)
```

(1) This new, "verbose" version runs at exactly the same speed as the old version. In fact, the compiled pattern objects are the same, since the `re.compile` function strips out all the stuff we added.

(2) This new, "verbose" version passes all the same tests as the old version. Nothing has changed, except that the programmer who comes back to this module in six months stands a fighting chance of understanding how the function works.

6.14. Postscript

A clever reader read the previous section and took it to the next level. The biggest headache (and performance drain) in the program as it is currently written is the regular expression, which is required because we have no other way of breaking down a Roman numeral. But there's only 5000 of them; why don't we just build a lookup table once, then simply read that? This idea gets even better when you realize that you don't need to use regular expressions at all. As you build the lookup table for converting integers to Roman numerals, you can build the reverse lookup table to convert Roman numerals to integers.

And best of all, he already had a complete set of unit tests. He changed over half the code in the module, but the unit tests stayed the same, so he could prove that his code worked just as well as the original.

Example 6.38. roman9.py

If you have not already done so, you can download this and other examples used in this book.

```
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Roman numerals must be less than 5000
MAX_ROMAN_NUMERAL = 4999

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

#Create tables for fast conversion of roman numerals.
#See fillLookupTables() below.
toRomanTable = [ None ]  # Skip an index since Roman numerals have no zero
fromRomanTable = {}

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n <= MAX_ROMAN_NUMERAL):
        raise OutOfRangeError, "number out of range (must be 1..%s)" % MAX_ROMAN_NUMERAL
    if int(n) <> n:
        raise NotIntegerError, "decimals can not be converted"
    return toRomanTable[n]

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, "Input can not be blank"
    if not fromRomanTable.has_key(s):
        raise InvalidRomanNumeralError, "Invalid Roman numeral: %s" % s
    return fromRomanTable[s]

def toRomanDynamic(n):
    """convert integer to Roman numeral using dynamic programming"""
    result = ""
    for numeral, integer in romanNumeralMap:
        if n >= integer:
            result = numeral
            n -= integer
            break
    if n > 0:
        result += toRomanTable[n]
    return result

def fillLookupTables():
```

```
    """compute all the possible roman numerals"""
    #Save the values in two global tables to convert to and from integers.
    for integer in range(1, MAX_ROMAN_NUMERAL + 1):
        romanNumber = toRomanDynamic(integer)
        toRomanTable.append(romanNumber)
        fromRomanTable[romanNumber] = integer

fillLookupTables()
```

So how fast is it?

Example 6.39. Output of `romantest9.py` against `roman9.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 0.791s

OK
```

Remember, the best performance we ever got in the original version was 13 tests in 3.315 seconds. Of course, it's not entirely a fair comparison, because this version will take longer to import (when it fills the lookup tables). But since import is only done once, this is negligible in the long run.

The moral of the story?

- Simplicity is a virtue.
- Especially when regular expressions are involved.
- And unit tests can give you the confidence to do large−scale refactoring... even if you didn't write the original code.

6.15. Summary

Unit testing is a powerful concept which, if properly implemented, can both reduce maintenance costs and increase flexibility in any long−term project. It is also important to understand that unit testing is not a panacea, a Magic Problem Solver, or a silver bullet. Writing good test cases is hard, and keeping them up to date takes discipline (especially when customers are screaming for critical bug fixes). Unit testing is not a replacement for other forms of testing, including functional testing, integration testing, and user acceptance testing. But it is feasible, and it does work, and once you've seen it work, you'll wonder how you ever got along without it.

This chapter covered a lot of ground, and much of it wasn't even Python−specific. There are unit testing frameworks for many languages, all of which require you to understand the same basic concepts:

- Designing test cases that are specific, automated, and independent
- Writing test cases *before* the code they are testing
- Writing tests that test good input and check for proper results
- Writing tests that test bad input and check for proper failures
- Writing and updating test cases to illustrate bugs or reflect new requirements
- Refactoring mercilessly to improve performance, scalability, readability, maintainability, or whatever other −ility you're lacking

Additionally, you should be comfortable doing all of the following Python−specific things:

* Subclassing `unittest.TestCase` and writing methods for individual test cases
* Using `assertEqual` to check that a function returns a known value
* Using `assertRaises` to check that a function raises a known exception
* Calling `unittest.main()` in your `if __name__` clause to run all your test cases at once
* Running unit tests in verbose or regular mode

Further reading

* XProgramming.com has links to download unit testing frameworks for many different languages.

---

[13] "I can resist everything except temptation." −−Oscar Wilde

Chapter 7. Data−Centric Programming

7.1. Diving in

In *Unit Testing*, we discussed the philosophy of unit testing and stepped through the implementation of it in Python. This chapter will focus more on advanced Python−specific techniques, centered around the `unittest` module. If you haven't read *Unit Testing*, you'll get lost about halfway through this chapter. You have been warned.

The following is a complete Python program that acts as a cheap and simple regression testing framework. It takes unit tests that you've written for individual modules, collects them all into one big test suite, and runs them all at once. I actually use this script as part of the build process for this book; I have unit tests for several of the example programs (not just the `roman.py` module featured in *Unit Testing*), and the first thing my automated build script does is run this program to make sure all my examples still work. If this regression test fails, the build immediately stops. I don't want to release non−working examples any more than you want to download them and sit around scratching your head and yelling at your monitor and wondering why they don't work.

Example 7.1. `regression.py`

If you have not already done so, you can download this and other examples used in this book.

```
"""Regression testing framework

This module will search for scripts in the same directory named
XYZtest.py.  Each such script should be a test suite that tests a
module through PyUnit.  (As of Python 2.1, PyUnit is included in
the standard library as "unittest".)  This script will aggregate all
found test suites into one big test suite and run them all at once.
"""

import sys, os, re, unittest

def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
    load = unittest.defaultTestLoader.loadTestsFromModule
    return unittest.TestSuite(map(load, modules))

if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest")
```

Running this script in the same directory as the rest of the example scripts that come with this book will find all the unit tests, named *module*`test.py`, run them as a single test, and pass or fail them all at once.

Example 7.2. Sample output of `regression.py`

```
[f8dy@oliver py]$ python regression.py −v
help should fail with no object ... ok                           (1)
help should return known result for apihelper ... ok
help should honor collapse argument ... ok
```

```
help should honor spacing argument ... ok
buildConnectionString should fail with list input ... ok           (2)
buildConnectionString should fail with string input ... ok
buildConnectionString should fail with tuple input ... ok
buildConnectionString handles empty dictionary ... ok
buildConnectionString returns known result with known input ... ok
fromRoman should only accept uppercase input ... ok                (3)
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non−integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
kgp a ref test ... ok
kgp b ref test ... ok
kgp c ref test ... ok
kgp d ref test ... ok
kgp e ref test ... ok
kgp f ref test ... ok
kgp g ref test ... ok

----------------------------------------------------------------------
Ran 29 tests in 2.799s

OK
```

(1) The first 5 tests are from `apihelpertest.py`, which tests the example script from *The Power Of Introspection*.

(2) The next 5 tests are from `odbchelpertest.py`, which tests the example script from *Getting To Know Python*.

(3) The rest are from `romantest.py`, which we studied in depth in *Unit Testing*.

7.2. Finding the path

When running Python scripts from the command line, it is sometimes useful to know where the currently running script is located on disk.

This is one of those obscure little tricks that is virtually impossible to figure out on your own, but simple to remember once you see it. The key to it is `sys.argv`. As we saw in *XML Processing*, this is a list that holds the list of command−line arguments. However, it also holds the name of the running script, exactly as it was called from the command line, and this is enough information to determine its location.

Example 7.3. `fullpath.py`

If you have not already done so, you can download this and other examples used in this book.

```
import sys, os

print 'sys.argv[0] =', sys.argv[0]             (1)
pathname = os.path.dirname(sys.argv[0])        (2)
print 'path =', pathname
print 'full path =', os.path.abspath(pathname) (3)
```

(1) Regardless of how you run a script, `sys.argv[0]` will always contain the name of the script, exactly as it appears on the command line. This may or may not include any path information, as we'll see shortly.

(2) `os.path.dirname` takes a filename as a string and returns the directory path portion. If the given filename does not include any path information, `os.path.dirname` returns an empty string.

(3) `os.path.abspath` is the key here. It takes a pathname, which can be partial or even blank, and returns a fully qualified pathname.

`os.path.abspath` deserves further explanation. It is very flexible; it can take any kind of pathname.

Example 7.4. Further explanation of `os.path.abspath`

```
>>> import os
>>> os.getcwd()                       (1)
/home/f8dy
>>> os.path.abspath('')               (2)
/home/f8dy
>>> os.path.abspath('.ssh')           (3)
/home/f8dy/.ssh
>>> os.path.abspath('/home/f8dy/.ssh') (4)
/home/f8dy/.ssh
>>> os.path.abspath('.ssh/../foo/')   (5)
/home/f8dy/foo
```

(1) `os.getcwd()` returns the current working directory.

(2) Calling `os.path.abspath` with an empty string returns the current working directory, same as `os.getcwd()`.

(3) Calling `os.path.abspath` with a partial pathname constructs a fully qualified pathname out of it, based on the current working directory.

(4) Calling `os.path.abspath` with a full pathname simply returns it.

(5) `os.path.abspath` also *normalizes* the pathname it returns. Note that this example worked even though I don't actually have a 'foo' directory. `os.path.abspath` never checks your actual disk; this is all just string manipulation.

Note: os.path.abspath does not validate pathnames
The pathnames and filenames you pass to `os.path.abspath` do not need to exist.

Note: Normalizing pathnames
`os.path.abspath` not only constructs full path names, it also normalizes them. If you are in the `/usr/` directory, `os.path.abspath('bin/../local/bin'` will return `/usr/local/bin`. If you just want to normalize a pathname without turning it into a full pathname, use `os.path.normpath` instead.

Example 7.5. Sample output from `fullpath.py`

```
[f8dy@oliver py]$ python /home/f8dy/diveintopython/common/py/fullpath.py (1)
sys.argv[0] = /home/f8dy/diveintopython/common/py/fullpath.py
path = /home/f8dy/diveintopython/common/py
full path = /home/f8dy/diveintopython/common/py
[f8dy@oliver diveintopython]$ python common/py/fullpath.py               (2)
sys.argv[0] = common/py/fullpath.py
path = common/py
full path = /home/f8dy/diveintopython/common/py
[f8dy@oliver diveintopython]$ cd common/py
[f8dy@oliver py]$ python fullpath.py                                     (3)
```

```
sys.argv[0] = fullpath.py
path =
full path = /home/f8dy/diveintopython/common/py
```

(1) In the first case, `sys.argv[0]` includes the full path of the script. We can then use the `os.path.dirname` function to strip off the script name and return the full directory name, and `os.path.abspath` simply returns what we give it.

(2) If the script is run by using a partial pathname, `sys.argv[0]` will still contain exactly what appears on the command line. `os.path.dirname` will then give us a partial pathname (relative to the current directory), and `os.path.abspath` will construct a full pathname from the partial pathname.

(3) If the script is run from the current directory without giving any path, `os.path.dirname` will simply return an empty string. Given an empty string, `os.path.abspath` returns the current directory, which is what we want, since the script was run from the current directory.

> **Note: os.path.abspath is cross−platform**
> Like the other functions in the `os` and `os.path` modules, `os.path.abspath` is cross−platform. Your results will look slightly different than my examples if you're running on Windows (which uses backslash as a path separator) or Mac OS (which uses colons), but they'll still work. That's the whole point of the `os` module.

Addendum. One reader was dissatisfied with this solution, and wanted to be able to run all the unit tests in the current directory, not the directory where `regression.py` is located. He suggests this approach instead:

Example 7.6. Running scripts in the current directory

```
import sys, os, re, unittest

def regressionTest():
    path = os.getcwd()       (1)
    sys.path.append(path)    (2)
    files = os.listdir(path) (3)
```

(1) Instead of setting `path` to the directory where the currently running script is located, we set it to the current working directory instead. This will be whatever directory you were in before you ran the script, which is not necessarily the same as the directory the script is in. (Read that sentence a few times until you get it.)

(2) Append this directory to the Python library search path, so that when we dynamically import the unit test modules later, Python can find them. We didn't have to do this when `path` was the directory of the currently running script, because Python always looks in that directory.

(3) The rest of the function is the same.

This technique will allow you to re−use this `regression.py` script on multiple projects. Just put the script in a common directory, then change to the project's directory before running it. All of that project's unit tests will be found and tested, instead of the unit tests in the common directory where `regression.py` is located.

7.3. Filtering lists revisited

You're already familiar with using list comprehensions to filter lists. There is another way to accomplish this same thing, which some people feel is more expressive.

Python has a built−in `filter` function which takes two arguments, a function and a list, and returns a list.[14] The function passed as the first argument to `filter` must itself take one argument, and the list that `filter` returns will contain all the elements from the list passed to `filter` for which the function passed to `filter` returns true.

Got all that? It's not as difficult as it sounds.

Example 7.7. Introducting `filter`

```
>>> def odd(n):           (1)
...     return n%2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> filter(odd, li)       (2)
[1, 3, 5, 9, -3]
>>> filteredList = []
>>> for n in li:          (3)
...     if odd(n):
...         filteredList.append(n)
...
>>> filteredList
[1, 3, 5, 9, -3]
```

(1) `odd` uses the built−in mod function "`%`" to return `1` if `n` is odd and `0` if `n` is even.

(2) `filter` takes two arguments, a function (`odd`) and a list (`li`). It loops through the list and calls `odd` with each element. If `odd` returns a true value (remember, any non−zero value is true in Python), then the element is included in the returned list, otherwise it is filtered out. The result is a list of only the odd numbers from the original list, in the same order as they appeared in the original.

(3) You could accomplish the same thing with a `for` loop. Depending on your programming background, this may seem more "straightforward", but functions like `filter` are much more expressive. Not only is it easier to write, it's easier to read, too. Reading the `for` loop is like standing too close to a painting; you see all the details, but it may take a few seconds to be able to step back and see the bigger picture: "Oh, we're just filtering the list!"

Example 7.8. `filter` in `regression.py`

```
    files = os.listdir(path)                      (1)
    test = re.compile("test.py$", re.IGNORECASE) (2)
    files = filter(test.search, files)            (3)
```

(1) As we saw in *Finding the path*, `path` may contain the full or partial pathname of the directory of the currently running script, or it may contain an empty string if the script is being run from the current directory. Either way, `files` will end up with the names of the files in the same directory as this script we're running.

(2) This is a compiled regular expression. As we saw in *Refactoring*, if you're going to use the same regular expression over and over, you should compile it for faster performance. The compiled object has a `search` method which takes a single argument, the string the search. If the regular expression matches the string, the `search` method returns a `Match` object containing information about the regular expression match; otherwise it returns `None`, the Python null value.

(3) For each element in the `files` list, we're going to call the `search` method of the compiled regular expression object, `test`. If the regular expression matches, the method will return a `Match` object, which Python considers to be true, so the element will be included in the list returned by `filter`. If the regular expression does not match, the `search` method will return `None`, which Python considers to be false, so the element will not be included.

Historical note. Versions of Python prior to 2.0 did not have list comprehensions, so you couldn't filter using list comprehensions; the `filter` function was the only game in town. Even with the introduction of list comprehensions in 2.0, some people still prefer the old−style `filter` (and its companion function, `map`, which we'll see later in this chapter). Both techniques work, and neither is going away, so which one you use is a matter of style.

Example 7.9. Filtering using list comprehensions instead

```
    files = os.listdir(path)
    test = re.compile("test.py$", re.IGNORECASE)
files = [f for f in files if test.search(f)] (1)
```

(1) This will accomplish exactly the same result as using the `filter` function. Which way is more expressive? That's up to you.

7.4. Mapping lists revisited

You're already familiar with using list comprehensions to map one list into another. There is another way to accomplish the same thing, using the built−in `map` function. It works much the same way as the `filter` function.

Example 7.10. Introducing `map`

```
>>> def double(n):
...     return n*2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> map(double, li)                       (1)
[2, 4, 6, 10, 18, 20, 512, -6]
>>> [double(n) for n in li]               (2)
[2, 4, 6, 10, 18, 20, 512, -6]
>>> newlist = []
>>> for n in li:                          (3)
...     newlist.append(double(n))
...
>>> newlist
[2, 4, 6, 10, 18, 20, 512, -6]
```

(1) `map` takes a function and a list[15] and returns a new list by calling the function with each element of the list in order. In this case, the function simply multiplies each element by 2.

(2) You could accomplish the same thing with a list comprehension. List comprehensions were first introduced in Python 2.0; `map` has been around forever.

(3) You could, if you insist on thinking like a Visual Basic programmer, use a `for` loop to accomplish the same thing.

Example 7.11. `map` with lists of mixed datatypes

```
>>> li = [5, 'a', (2, 'b')]
>>> map(double, li)                       (1)
[10, 'aa', (2, 'b', 2, 'b')]
```

(1) As a side note, I'd like to point out that `map` works just as well with lists of mixed datatypes, as long as the function you're using correctly handles each type. In this case, our `double` function simply multiples the given argument by 2, and Python Does The Right Thing depending on the datatype of the argument. For integers, this means actually multiplying it by 2; for strings, it means concatenating the string with itself; for tuples, it means making a new tuple that has all of the elements of the original, then all of the elements of the original again.

All right, enough play time. Let's look at some real code.

Example 7.12. `map` in `regression.py`

```
filenameToModuleName = lambda f: os.path.splitext(f)[0] (1)
moduleNames = map(filenameToModuleName, files)          (2)
```

(1) As we saw in *Using lambda functions*, `lambda` defines an inline function. And as we saw in Example 3.36, `os.path.splitext` takes a filename and returns a tuple `(name, extension)`. So `filenameToModuleName` is a function which will take a filename and strip off the file extension, and return just the name.

(2) Calling `map` takes each filename listed in `files`, passes it to our function `filenameToModuleName`, and returns a list of the return values of each of those function calls. In other words, we strip the file extension off of each filename, and store the list of all those stripped filenames in `moduleNames`.

As we'll see in the rest of the chapter, we can extend this type of data−centric thinking all the way to our final goal, which is to define and execute a single test suite that contains the tests from all of those individual test suites.

7.5. Data−centric programming

By now you're probably scratching your head wondering why this is better than using `for` loops and straight function calls. And that's a perfectly valid question. Mostly, it's a matter of perspective. Using `map` and `filter` forces you to center your thinking around your data.

In this case, we started with no data at all; the first thing we did was get the directory path of the current script, and got a list of files in that directory. That was our bootstrap, and it gave us real data to work with: a list of filenames.

However, we knew we didn't care about all of those files, only the ones that were actually test suites. We had *too much data*, so we needed to `filter` it. How did we know which data to keep? We needed a test to decide, so we defined one and passed it to the `filter` function. In this case we used a regular expression to decide, but the concept would be the same regardless of how we constructed the test.

Now we had the filenames of each of the test suites (and only the test suites, since everything else had been filtered out), but we really wanted module names instead. We had the right amount of data, but it was *in the wrong format*. So we defined a function that would transform a single filename into a module name, and we mapped that function onto the entire list. From one filename, we can get a module name; from a list of filenames, we can get a list of module names.

Instead of `filter`, we could have used a `for` loop with an `if` statement. Instead of `map`, we could have used a `for` loop with a function call. But using `for` loops like that is busywork. At best, it simply wastes time; at worst, it introduces obscure bugs. For instance, we have to figure out how to test for the condition "is this file a test suite?" anyway; that's our application−specific logic, and no language can write that for us. But once we've figured that out, do we really want go to all the trouble of defining a new empty list and writing a `for` loop and an `if` statement and manually calling `append` to add each element to the new list if it passes the condition and then keeping track of which variable holds the new filtered data and which one holds the old unfiltered data? Why not just define the test condition, then let Python do the rest of that work for us?

Oh sure, you could try to be fancy and delete elements in place without creating a new list. But you've been burned by that before. Trying to modify a data structure that you're looping through can be tricky. You delete an element, then loop to the next element, and suddenly you've skipped one. Is Python one of the languages that works that way? How long would it take you to figure it out? Would you remember for certain whether it was safe the next time you tried? Programmers spend so much time and make so many mistakes dealing with purely technical issues like this, and it's all pointless. It doesn't advance your program at all; it's just busywork.

I resisted list comprehensions when I first learned Python, and I resisted `filter` and `map` even longer. I insisted on making my life more difficult, sticking to the familiar way of `for` loops and `if` statements and step−by−step code−centric programming. And my Python programs looked a lot like Visual Basic programs, detailing every step of every operation in every function. And they had all the same types of little problems and obscure bugs. And it was all pointless.

Let it all go. Busywork code is not important. Data is important. And data is not difficult. It's only data. If you have too much, filter it. If it's not what you want, map it. Focus on the data; leave the busywork behind.

7.6. Dynamically importing modules

Sorry, you've reached the end of the chapter that's been written so far. Please check back at http://diveintopython.org/ for updates.

---

[14] Technically, the second argument to `filter` can be any sequence, including lists, tuples, and custom classes that act like lists by defining the `__getitem__` special method. If possible, `filter` will return the same datatype as you give it, so filtering a list returns a list, but filtering a tuple returns a tuple.

[15] Again, I should point out that `map` can take a list, a tuple, or any object that acts like a sequence. See previous footnote about `filter`.

Appendix A. Further reading

Chapter 1. Getting To Know Python

- 1.3. Documenting functions
    - *Python Style Guide* discusses how to write a good `doc string`.
    - *Python Tutorial* discusses conventions for spacing in `doc string`s.
- 1.4. Everything is an object
    - *Python Reference Manual* explains exactly what it means to say that everything in Python is an object, because some people are pedantic and like to discuss this sort of thing at great length.
    - eff−bot summarizes Python objects.
- 1.5. Indenting code
    - *Python Reference Manual* discusses cross−platform indentation issues and shows various indentation errors.
    - *Python Style Guide* discusses good indentation style.
- 1.6. Testing modules
    - *Python Reference Manual* discusses the low−level details of importing modules.
- 1.7. Introducing dictionaries
    - *How to Think Like a Computer Scientist* teaches about dictionaries and shows how to use dictionaries to model sparse matrices.
    - Python Knowledge Base has lots of example code using dictionaries.
    - Python Cookbook discusses how to sort the values of a dictionary by key.
    - *Python Library Reference* summarizes all the dictionary methods.
- 1.8. Introducing lists
    - *How to Think Like a Computer Scientist* teaches about lists and makes an important point about passing lists as function arguments.
    - *Python Tutorial* shows how to use lists as stacks and queues.
    - Python Knowledge Base answers common questions about lists and has lots of example code using lists.
    - *Python Library Reference* summarizes all the list methods.
- 1.9. Introducing tuples
    - *How to Think Like a Computer Scientist* teaches about tuples and shows how to concatenate tuples.
    - Python Knowledge Base shows how to sort a tuple.
    - *Python Tutorial* shows how to define a tuple with one element.
- 1.10. Defining variables
    - *Python Reference Manual* shows examples of when you can skip the line continuation character and when you have to use it.
- 1.11. Assigning multiple values at once
    - *How to Think Like a Computer Scientist* shows how to use multi−variable assignment to swap the values of two variables.
- 1.12. Formatting strings
    - *Python Library Reference* summarizes all the string formatting format characters.

    - *Effective AWK Programming* discusses all the format characters and advanced string formatting techniques like specifying width, precision, and zero−padding.
- 1.13. Mapping lists
    - *Python Tutorial* discusses another way to map lists using the built−in `map` function.
    - *Python Tutorial* shows how to do nested list comprehensions.
- 1.14. Joining lists and splitting strings
    - Python Knowledge Base answers common questions about strings and has lots of example code using strings.
    - *Python Library Reference* summarizes all the string methods.
    - *Python Library Reference* documents the `string` module.
    - *The Whole Python FAQ* explains why `join` is a string method instead of a list method.

Chapter 2. The Power Of Introspection

- 2.2. Optional and named arguments
    - *Python Tutorial* discusses exactly when and how default arguments are evaluated, which matters when the default value is a list or an expression with side effects.
- 2.3. type, str, dir, and other built−in functions
    - *Python Library Reference* documents all the built−in functions and all the built−in exceptions.
- 2.5. Filtering lists
    - *Python Tutorial* discusses another way to filter lists using the built−in `filter` function.
- 2.6. The peculiar nature of and and or
    - Python Cookbook discusses alternatives to the `and−or` trick.
- 2.7. Using lambda functions
    - Python Knowledge Base discusses using `lambda` to call functions indirectly.
    - *Python Tutorial* shows how to access outside variables from inside a `lambda` function. (PEP 227 explains how this will change in future versions of Python.)
    - *The Whole Python FAQ* has examples of obfuscated one−liners using `lambda`.

Chapter 3. An Object−Oriented Framework

- 3.2. Importing modules using from module import
    - eff−bot has more to say on `import` *`module`* *vs.* `from` *`module`* `import`.
    - *Python Tutorial* discusses advanced import techniques, including `from` *`module`* `import *`.
- 3.3. Defining classes
    - *Learning to Program* has a gentler introduction to classes.
    - *How to Think Like a Computer Scientist* shows how to use classes to model compound datatypes.
    - *Python Tutorial* has an in−depth look at classes, namespaces, and inheritance.
    - Python Knowledge Base answers common questions about classes.
- 3.4. Instantiating classes
    - *Python Library Reference* summarizes built−in attributes like `__class__`.
    - *Python Library Reference* documents the `gc` module, which gives you low−level control over

Python's garbage collection.

* 3.5. UserDict: a wrapper class
    * *Python Library Reference* documents the `UserDict` module and the `copy` module.
* 3.7. Advanced special class methods
    * *Python Reference Manual* documents all the special class methods.
* 3.9. Private functions
    * *Python Tutorial* discusses the inner workings of private variables.
* 3.10. Handling exceptions
    * *Python Tutorial* discusses defining and raising your own exceptions, and handling multiple exceptions at once.
    * *Python Library Reference* summarizes all the built−in exceptions.
    * *Python Library Reference* documents the getpass module.
    * *Python Library Reference* documents the `traceback` module, which provides low−level access to exception attributes after an exception is raised.
    * *Python Reference Manual* discusses the inner workings of the `try...except` block.
* 3.11. File objects
    * *Python Tutorial* discusses reading and writing files, including how to read a file one line at a time into a list.
    * eff−bot discusses efficiency and performance of various ways of reading a file.
    * Python Knowledge Base answers common questions about files.
    * *Python Library Reference* summarizes all the file object methods.
* 3.13. More on modules
    * *Python Tutorial* discusses exactly when and how default arguments are evaluated.
    * *Python Library Reference* documents the `sys` module.
* 3.14. The os module
    * Python Knowledge Base answers questions about the `os` module.
    * *Python Library Reference* documents the `os` module and the `os.path` module.

Chapter 4. HTML Processing

* 4.4. Introducing BaseHTMLProcessor.py
    * W3C discusses character and entity references.
    * *Python Library Reference* confirms your suspicions that the `htmlentitydefs` module is exactly what it sounds like.
* 4.9. Regular expressions 101
    * Regular Expression HOWTO teaches about regular expressions and how to use them in Python.
    * *Python Library Reference* summarizes the `re` module.
* 4.10. Putting it all together
    * You thought I was kidding about the server−side scripting idea. So did I, until I found this web−based dialectizer. I have no idea if it's implemented in Python, but my company's home page is funny as hell in Pig Latin. Unfortunately, source code does not appear to be available.

Chapter 5. XML Processing

- 5.4. Unicode
    - ♦ Unicode.org is the home page of the unicode standard, including a brief technical introduction.
    - ♦ Unicode Tutorial has some more examples of how to use Python's unicode functions, including how to force Python to coerce unicode into ASCII even when it doesn't really want to.
    - ♦ Unicode Proposal is the original technical specification for Python's unicode functionality. For advanced unicode hackers only.

Chapter 6. Unit Testing

- 6.1. Diving in
    - ♦ This site has more on Roman numerals, including a fascinating history of how Romans and other civilizations really used them (short answer: haphazardly and inconsistently).
- 6.2. Introducing romantest.py
    - ♦ The PyUnit home page has an in−depth discussion of using the `unittest` framework, including advanced features not covered in this chapter.
    - ♦ The PyUnit FAQ explains why test cases are stored separately from the code they test.
    - ♦ *Python Library Reference* summarizes the `unittest` module.
    - ♦ ExtremeProgramming.org discusses why you should write unit tests.
    - ♦ The Portland Pattern Repository has an ongoing discussion of unit tests, including a standard definition, why you should code unit tests first, and several in−depth case studies.
- 6.15. Summary
    - ♦ XProgramming.com has links to download unit testing frameworks for many different languages.

Chapter 7. Data−Centric Programming

Appendix B. A 5−minute review

Chapter 1. Getting To Know Python

- 1.1. Diving in

  Here is a complete, working Python program.
- 1.2. Declaring functions

  Python has functions like most other languages, but it does not have separate header files like C++ or `interface`/`implementation` sections like Pascal. When you need a function, just declare it and code it.
- 1.3. Documenting functions

  You can document a Python function by giving it a `doc string`.
- 1.4. Everything is an object

  A function, like everything else in Python, is an object.
- 1.5. Indenting code

  Python functions have no explicit `begin` or `end`, no curly braces that would mark where the function code starts and stops. The only delimiter is a colon ("`:`") and the indentation of the code itself.
- 1.6. Testing modules

  Python modules are objects and have several useful attributes. You can use this to easily test your modules as you write them.
- 1.7. Introducing dictionaries

  One of Python's built−in datatypes is the dictionary, which defines one−to−one relationships between keys and values.
- 1.8. Introducing lists

  Lists are Python's workhorse datatype. If your only experience with lists is arrays in Visual Basic or (God forbid) the datastore in Powerbuilder, brace yourself for Python lists.
- 1.9. Introducing tuples

  A tuple is an immutable list. A tuple can not be changed in any way once it is created.
- 1.10. Defining variables

  Python has local and global variables like most other languages, but it has no explicit variable declarations. Variables spring into existence by being assigned a value, and are automatically destroyed when they go out of scope.
- 1.11. Assigning multiple values at once

  One of the cooler programming shortcuts in Python is using sequences to assign multiple values at once.
- 1.12. Formatting strings

  Python supports formatting values into strings. Although this can include very complicated expressions, the most basic usage is to insert values into a string with the `%s` placeholder.
- 1.13. Mapping lists

One of the most powerful features of Python is the list comprehension, which provides a compact way of mapping a list into another list by applying a function to each of the elements of the list.

- 1.14. Joining lists and splitting strings

  You have a list of key−value pairs in the form *`key=value`*, and you want to join them into a single string. To join any list of strings into a single string, use the `join` method of a string object.
- 1.15. Summary

  The `odbchelper.py` program and its output should now make perfect sense.

Chapter 2. The Power Of Introspection

- 2.1. Diving in

  Here is a complete, working Python program. You should understand a good deal about it just by looking at it. The numbered lines illustrate concepts covered in *Getting To Know Python*. Don't worry if the rest of the code looks intimidating; you'll learn all about it throughout this chapter.
- 2.2. Optional and named arguments

  Python allows function arguments to have default values; if the function is called without the argument, the argument gets its default value. Futhermore, arguments can be specified in any order by using named arguments. Stored procedures in SQL Server Transact/SQL can do this; if you're a SQL Server scripting guru, you can skim this part.
- 2.3. type, str, dir, and other built−in functions

  Python has a small set of extremely useful built−in functions. All other functions are partitioned off into modules. This was actually a conscious design decision, to keep the core language from getting bloated like other scripting languages (cough cough, Visual Basic).
- 2.4. Getting object references with getattr

  You already know that Python functions are objects. What you don't know is that you can get a reference to a function without knowing its name until run−time, using the `getattr` function.
- 2.5. Filtering lists

  As you know, Python has powerful capabilities for mapping lists into other lists, via list comprehensions. This can be combined with a filtering mechanism, where some elements in the list are mapped while others are skipped entirely.
- 2.6. The peculiar nature of and and or

  In Python, `and` and `or` perform boolean logic as you would expect, but they do not return boolean values; they return one of the actual values they are comparing.
- 2.7. Using lambda functions

  Python supports an interesting syntax that lets you define one−line mini−functions on the fly. Borrowed from Lisp, these so−called `lambda` functions can be used anywhere a function is required.
- 2.8. Putting it all together

The last line of code, the only one we haven't deconstructed yet, is the one that does all the work. But by now the work is easy, because everything we need is already set up just the way we need it. All the dominoes are in place; it's time to knock them down.

* 2.9. Summary

  The `apihelper.py` program and its output should now make perfect sense.

Chapter 3. An Object−Oriented Framework

* 3.1. Diving in

  Here is a complete, working Python program. Read the `doc string`s of the module, the classes, and the functions to get an overview of what this program does and how it works. As usual, don't worry about the stuff you don't understand; that's what the rest of the chapter is for.
* 3.2. Importing modules using from module import

  Python has two ways of importing modules. Both are useful, and you should know when to use each. One way, `import module`, you've already seen in chapter 1. The other way accomplishes the same thing but works in subtlely and importantly different ways.
* 3.3. Defining classes

  Python is fully object−oriented: you can define your own classes, inherit from your own or built−in classes, and instantiate the classes you've defined.
* 3.4. Instantiating classes

  Instantiating classes in Python is straightforward. To instantiate a class, simply call the class as if it were a function, passing the arguments that the `__init__` method defines. The return value will be the newly created object.
* 3.5. UserDict: a wrapper class

  As you've seen, `FileInfo` is a class that acts like a dictionary. To explore this further, let's look at the `UserDict` class in the `UserDict` module, which is the ancestor of our `FileInfo` class. This is nothing special; the class is written in Python and stored in a `.py` file, just like our code. In particular, it's stored in the `lib` directory in your Python installation.
* 3.6. Special class methods

  In addition to normal class methods, there are a number of special methods which Python classes can define. Instead of being called directly by your code (like normal methods), special methods are called for you by Python in particular circumstances or when specific syntax is used.
* 3.7. Advanced special class methods

  There are more special methods than just `__getitem__` and `__setitem__`. Some of them let you emulate functionality that you may not even know about.
* 3.8. Class attributes

  You already know about data attributes, which are variables owned by a specific instance of a class. Python also supports class attributes, which are variables owned by the class itself.
* 3.9. Private functions

Like most languages, Python has the concept of private functions, which can not be called from outside their module; private class methods, which can not be called from outside their class; and private attributes, which can not be accessed from outside their class. Unlike most languages, whether a Python function, method, or attribute is private or public is determined entirely by its name.

* 3.10. Handling exceptions

  Like many object−oriented languages, Python has exception handling via `try...except` blocks.
* 3.11. File objects

  Python has a built−in function, `open`, for opening a file on disk. `open` returns a file object, which has methods and attributes for getting information about and manipulating the opened file.
* 3.12. for loops

  Like most other languages, Python has `for` loops. The only reason you haven't seen them until now is that Python is good at so many other things that you don't need them as often.
* 3.13. More on modules

  Modules, like everything else in Python, are objects. Once imported, you can always get a reference to a module through the global dictionary `sys.modules`.
* 3.14. The os module

  The `os` module has lots of useful functions for manipulating files and processes, and `os.path` has functions for manipulating file and directory paths.
* 3.15. Putting it all together

  Once again, all the dominoes are in place. We've seen how each line of code works. Now let's step back and see how it all fits together.
* 3.16. Summary

  The `fileinfo.py` program should now make perfect sense.

Chapter 4. HTML Processing

* 4.1. Diving in

  I often see questions on comp.lang.python like "How can I list all the [headers|images|links] in my HTML document?" "How do I [parse|translate|munge] the text of my HTML document but leave the tags alone?" "How can I [add|remove|quote] attributes of all my HTML tags at once?" This chapter will answer all of these questions.
* 4.2. Introducing sgmllib.py

  HTML processing is broken into three steps: breaking down the HTML into its constituent pieces, fiddling with the pieces, and reconstructing the pieces into HTML again. The first step is done by `sgmllib.py`, a part of the standard Python library.
* 4.3. Extracting data from HTML documents

  To extract data from HTML documents, subclass the `SGMLParser` class and define methods for each tag or entity you want to capture.
* 4.4. Introducing BaseHTMLProcessor.py

SGMLParser doesn't produce anything by itself. It parses and parses and parses, and it calls a method for each interesting thing it finds, but the methods don't do anything. SGMLParser is an HTML *consumer*: it takes HTML and breaks it down into small, structured pieces. As you saw in the previous section, you can subclass SGMLParser to define classes that catch specific tags and produce useful things, like a list of all the links on a web page. Now we'll take this one step further by defining a class that catches everything SGMLParser throws at it and reconstructs the complete HTML document. In technical terms, this class will be an HTML *producer*.

- 4.5. locals and globals

  Python has two built−in functions, locals and globals, which provide dictionary−based access to local and global variables.
- 4.6. Dictionary−based string formatting

  There is an alternative form of string formatting that uses dictionaries instead of tuples of values.
- 4.7. Quoting attribute values

  A common question on comp.lang.python is "I have a bunch of HTML documents with unquoted attribute values, and I want to properly quote them all. How can I do this?"[10] (This is generally precipitated by a project manager who has found the HTML−is−a−standard religion joining a large project and proclaiming that all pages must validate against an HTML validator. Unquoted attribute values are a common violation of the HTML standard.) Whatever the reason, unquoted attribute values are easy to fix by feeding HTML through BaseHTMLProcessor.
- 4.8. Introducing dialect.py

  Dialectizer is a simple (and silly) descendant of BaseHTMLProcessor. It runs blocks of text through a series of substitutions, but it makes sure that anything within a <pre>...</pre> block passes through unaltered.
- 4.9. Regular expressions 101

  Regular expressions are a powerful (and fairly standardized) way of searching, replacing, and parsing text with complex patterns of characters. If you've used regular expressions in other languages (like Perl), you should skip this section and just read the summary of the re module to get an overview of the available functions and their arguments.
- 4.10. Putting it all together

  It's time to put everything we've learned so far to good use. I hope you were paying attention.
- 4.11. Summary

  Python provides you with a powerful tool, sgmllib.py, to manipulate HTML by turning its structure into an object model. You can use this tool in many different ways.

Chapter 5. XML Processing

- 5.1. Diving in

  There are two basic ways to work with XML. One is called SAX ("Simple API for XML"), and it works by reading the XML a little bit at a time and calling a method for each element it finds. (If you read *HTML Processing*, this should sound familiar, because that's how the sgmllib module works.) The other is called DOM ("Document Object Model"), and it works by reading in the entire XML document at once and creating an internal representation of it using native Python classes linked in a tree structure. Python has standard modules for both kinds of parsing, but this chapter will only deal with using the DOM.

* 5.2. Packages

  Actually parsing an XML document is very simple: one line of code. However, before we get to that line of code, we need to take a short detour to talk about packages.
* 5.3. Parsing XML

  As I was saying, actually parsing an XML document is very simple: one line of code. Where you go from there is up to you.
* 5.4. Unicode

  Unicode is a system to represent characters from all the world's different languages. When Python parses an XML document, all data is stored in memory as unicode.
* 5.5. Searching for elements

  Traversing XML documents by stepping through each node can be tedious. If you're looking for something in particular, buried deep within your XML document, there is a shortcut you can use to find it quickly: `getElementsByTagName`.
* 5.6. Accessing element attributes

  XML elements can have one or more attributes, and it is incredibly simple to access them once you have parsed an XML document.
* 5.7. Abstracting input sources

  One of Python's greatest strengths is its dynamic binding, and one powerful use of dynamic binding is the *file−like object*.
* 5.8. Standard input, output, and error

  UNIX users are already familiar with the concept of standard input, standard output, and standard error. This section is for the rest of you.
* 5.9. Caching node lookups

  `kgp.py` employs several tricks which may or may not be useful to you in your XML processing. The first one takes advantage of the consistent structure of the input documents to build a cache of nodes.
* 5.10. Finding direct children of a node

  Another useful techique when parsing XML documents is finding all the direct child elements of a particular element. For instance, in our grammar files, a `ref` element can have several `p` elements, each of which can contain many things, including other `p` elements. We want to find just the `p` elements that are children of the `ref`, not `p` elements that are children of other `p` elements.
* 5.11. Creating separate handlers by node type

  The third useful XML processing tip involves separating your code into logical functions, based on node types and element names. Parsed XML documents are made up of various types of nodes, each represented by a Python object. The root level of the document itself is represented by a `Document` object. The `Document` then contains one or more `Element` objects (for actual XML tags), each of which may contain other `Element` objects, `Text` objects (for bits of text), or `Comment` objects (for embedded comments). Python makes it easy to write a dispatcher to separate the logic for each node type.

- 5.12. Handling command line arguments

    Python fully supports creating programs that can be run on the command line, complete with command−line arguments and either short− or long−style flags to specify various options. None of this is XML−specific, but this script makes good use of command−line processing, so it seemed like a good time to mention it.
- 5.13. Putting it all together

    We've covered a lot of ground. Let's step back and see how all the pieces fit together.
- 5.14. Summary

    Python comes with powerful libraries for parsing and manipulating XML documents. The `minidom` takes an XML file and parses it into Python objects, providing for random access to arbitrary elements. Furthermore, this chapter shows how Python can be used to create a "real" standalone command−line script, complete with command−line flags, command−line arguments, error handling, even the ability to take input from the piped result of a previous program.

Chapter 6. Unit Testing

- 6.1. Diving in

    In previous chapters, we "dived in" by immediately looking at code and trying to understanding it as quickly as possible. Now that you have some Python under your belt, we're going to step back and look at the steps that happen *before* the code gets written.
- 6.2. Introducing romantest.py

    Now that we've completely defined the behavior we expect from our conversion functions, we're going to do something a little unexpected: we're going to write a test suite that puts these functions through their paces and makes sure that they behave the way we want them to. You read that right: we're going to write code that tests code that we haven't written yet.
- 6.3. Testing for success

    The most fundamental part of unit testing is constructing individual test cases. A test case answers a single question about the code it is testing.
- 6.4. Testing for failure

    It is not enough to test that our functions succeed when given good input; we must also test that they fail when given bad input. And not just any sort of failure; they must fail in the way we expect.
- 6.5. Testing for sanity

    Often, you will find that a unit of code contains a set of reciprocal functions, usually in the form of conversion functions where one converts A to B and the other converts B to A. In these cases, it is useful to create a "sanity check" to make sure that you can convert A to B and back to A without losing decimal precision, incurring rounding errors, or triggering any other sort of bug.
- 6.6. roman.py, stage 1

    Now that our unit test is complete, it's time to start writing the code that our test cases are attempting to test. We're going to do this in stages, so we can see all the unit tests fail, then watch them pass one by one as we fill in the gaps in `roman.py`.

- 6.7. roman.py, stage 2

  Now that we have the framework of our `roman` module laid out, it's time to start writing code and passing test cases.
- 6.8. roman.py, stage 3

  Now that `toRoman` behaves correctly with good input (integers from `1` to `3999`), it's time to make it behave correctly with bad input (everything else).
- 6.9. roman.py, stage 4

  Now that `toRoman` is done, it's time to start coding `fromRoman`. Thanks to our rich data structure that maps individual Roman numerals to integer values, this is no more difficult than the `toRoman` function.
- 6.10. roman.py, stage 5

  Now that `fromRoman` works properly with good input, it's time to fit in the last piece of the puzzle: making it work properly with bad input. That means finding a way to look at a string and determine if it's a valid Roman numeral. This is inherently more difficult than validating numeric input in `toRoman`, but we have a powerful tool at our disposal: regular expressions.
- 6.11. Handling bugs

  Despite your best efforts to write comprehensive unit tests, bugs happen. What do I mean by "bug"? A bug is a test case you haven't written yet.
- 6.12. Handling changing requirements

  Despite your best efforts to pin your customers to the ground and extract exact requirements from them on pain of horrible nasty things involving scissors and hot wax, requirements will change. Most customers don't know what they want until they see it, and even if they do, they aren't that good at articulating what they want precisely enough to be useful. And even if they do, they'll want more in the next release anyway. So be prepared to update your test cases as requirements change.
- 6.13. Refactoring

  The best thing about comprehensive unit testing is not the feeling you get when all your test cases finally pass, or even the feeling you get when someone else blames you for breaking their code and you can actually *prove* that you didn't. The best thing about unit testing is that it gives you the freedom to refactor mercilessly.
- 6.14. Postscript

  A clever reader read the previous section and took it to the next level. The biggest headache (and performance drain) in the program as it is currently written is the regular expression, which is required because we have no other way of breaking down a Roman numeral. But there's only 5000 of them; why don't we just build a lookup table once, then simply read that? This idea gets even better when you realize that you don't need to use regular expressions at all. As you build the lookup table for converting integers to Roman numerals, you can build the reverse lookup table to convert Roman numerals to integers.
- 6.15. Summary

  Unit testing is a powerful concept which, if properly implemented, can both reduce maintenance costs and increase flexibility in any long−term project. It is also important to understand that unit testing is not a panacea, a Magic Problem Solver, or a silver bullet.

Writing good test cases is hard, and keeping them up to date takes discipline (especially when customers are screaming for critical bug fixes). Unit testing is not a replacement for other forms of testing, including functional testing, integration testing, and user acceptance testing. But it is feasible, and it does work, and once you've seen it work, you'll wonder how you ever got along without it.

Chapter 7. Data−Centric Programming

- 7.1. Diving in

  In *Unit Testing*, we discussed the philosophy of unit testing and stepped through the implementation of it in Python. This chapter will focus more on advanced Python−specific techniques, centered around the `unittest` module. If you haven't read *Unit Testing*, you'll get lost about halfway through this chapter. You have been warned.
- 7.2. Finding the path

  When running Python scripts from the command line, it is sometimes useful to know where the currently running script is located on disk.
- 7.3. Filtering lists revisited

  You're already familiar with using list comprehensions to filter lists. There is another way to accomplish this same thing, which some people feel is more expressive.
- 7.4. Mapping lists revisited

  You're already familiar with using list comprehensions to map one list into another. There is another way to accomplish the same thing, using the built−in `map` function. It works much the same way as the `filter` function.
- 7.5. Data−centric programming

  By now you're probably scratching your head wondering why this is better than using `for` loops and straight function calls. And that's a perfectly valid question. Mostly, it's a matter of perspective. Using `map` and `filter` forces you to center your thinking around your data.
- 7.6. Dynamically importing modules

  Sorry, you've reached the end of the chapter that's been written so far. Please check back at http://diveintopython.org/ for updates.

Appendix C. Tips and tricks

Chapter 1. Getting To Know Python

- 1.1. Diving in

  Tip: Run module (Windows)
  In the Python IDE on Windows, you can run a module with File−>Run... (**Ctrl−R**). Output is displayed in the interactive window.

  Tip: Run module (Mac OS)
  In the Python IDE on Mac OS, you can run a module with Python−>Run window... (**Cmd−R**), but there is an important option you must set first. Open the module in the IDE, pop up the module's options menu by clicking the black triangle in the upper−right corner of the window, and make sure "Run as __main__" is checked. This setting is saved with the module, so you only have to do this once per module.

  Tip: Run module (UNIX)
  On UNIX−compatible systems (including Mac OS X), you can run a module from the command line: **`python odbchelper.py`**
- 1.2. Declaring functions

  Note: Python vs. Visual Basic: return values
  In Visual Basic, functions (that return a value) start with `function`, and subroutines (that do not return a value) start with `sub`. There are no subroutines in Python. Everything is a function, all functions return a value (even if it's `None`), and all functions start with `def`.

  Note: Python vs. Java: return values
  In Java, C++, and other statically−typed languages, you must specify the datatype of the function return value and each function argument. In Python, you never explicitly specify the datatype of anything. Based on what value you assign, Python keeps track of the datatype internally.
- 1.3. Documenting functions

  Note: Python vs. Perl: quoting
  Triple quotes are also an easy way to define a string with both single and double quotes, like `qq/.../` in Perl.

  Note: Why doc strings are a Good Thing
  Many Python IDEs use the `doc string` to provide context−sensitive documentation, so that when you type a function name, its `doc string` appears as a tooltip. This can be incredibly helpful, but it's only as good as the `doc string`s you write.
- 1.4. Everything is an object

  Note: Python vs. Perl: import
  `import` in Python is like `require` in Perl. Once you `import` a Python module, you access its functions with *`module.function`*; once you `require` a Perl module, you access its functions with *`module`*`::`*`function`*.
- 1.5. Indenting code

  Note: Python vs. Java: separating statements
  Python uses carriage returns to separate statements and a colon and indentation to separate code blocks. C++ and Java use semicolons to separate statements and curly braces to separate code blocks.
- 1.6. Testing modules

  Note: Python vs. C: comparison and assignment
  Like C, Python uses `==` for comparison and `=` for assignment. Unlike C, Python does not

support in−line assignment, so there's no chance of accidentally assigning the value you thought you were comparing.

Tip: if __name__ on Mac OS
On MacPython, there is an additional step to make the `if __name__` trick work. Pop up the module's options menu by clicking the black triangle in the upper−right corner of the window, and make sure Run as __main__ is checked.

- 1.7. Introducing dictionaries

Note: Python vs. Perl: dictionaries
A dictionary in Python is like a hash in Perl. In Perl, variables which store hashes always start with a `%` character; in Python, variables can be named anything, and Python keeps track of the datatype internally.

Note: Python vs. Java: dictionaries
A dictionary in Python is like an instance of the `Hashtable` class in Java.

Note: Python vs. Visual Basic: dictionaries
A dictionary in Python is like an instance of the `Scripting.Dictionary` object in Visual Basic.

Note: Dictionaries are unordered
Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered. This is an important distinction which will annoy you when you want to access the elements of a dictionary in a specific, repeatable order (like alphabetical order by key). There are ways of doing this, they're just not built into the dictionary.

- 1.8. Introducing lists

Note: Python vs. Perl: lists
A list in Python is like an array in Perl. In Perl, variables which store arrays always start with the `@` character; in Python, variables can be named anything, and Python keeps track of the datatype internally.

Note: Python vs. Java: lists
A list in Python is much more than an array in Java (although it can be used as one if that's really all you want out of life). A better analogy would be to the `Vector` class, which can hold arbitrary objects and can expand dynamically as new items are added.

Note: What's true in Python?
Before version 2.2.1, Python had no separate boolean datatype. To compensate for this, Python accepted almost anything in a boolean context (like an `if` statement), according to the following rules: `0` is false; all other numbers are true. An empty string (`""`) is false, all other strings are true. An empty list (`[]`) is false; all other lists are true. An empty tuple (`()`) is false; all other tuples are true. An empty dictionary (`{}`) is false; all other dictionaries are true. These rules still apply in Python 2.2.1 and beyond, but now you can also use an actual boolean, which has a value of `True` or `False`. Note the capitalization; these values, like everything else in Python, are case−sensitive.

- 1.9. Introducing tuples

Note: Tuples into lists into tuples
Tuples can be converted into lists, and vice−versa. The built−in `tuple` function takes a list and returns a tuple with the same elements, and the `list` function takes a tuple and returns a list. In effect, `tuple` freezes a list, and `list` thaws a tuple.

- 1.10. Defining variables

Note: Multiline commands
When a command is split among several lines with the line continuation marker ("\"), the continued lines can be indented in any manner; Python's normally stringent indentation rules do not apply. If your Python IDE auto−indents the continued line, you should probably accept its default unless you have a burning reason not to.

Note: Implicit multiline commands
Strictly speaking, expressions in parentheses, straight brackets, or curly braces (like defining a dictionary) can be split into multiple lines with or without the line continuation character ("\"). I like to include the backslash even when it's not required because I think it makes the code easier to read, but that's a matter of style.

- 1.12. Formatting strings
  Note: Python vs. C: string formatting
  String formatting in Python uses the same syntax as the `sprintf` function in C.
- 1.14. Joining lists and splitting strings
  Important: You can't join non−strings
  `join` only works on lists of strings; it does not do any type coercion. `join`ing a list that has one or more non−string elements will raise an exception.

  Note: Searching with split
  `*anystring*.split(*delimiter*, 1)` is a useful technique when you want to search a string for a substring and then work with everything before the substring (which ends up in the first element of the returned list) and everything after it (which ends up in the second element).

Chapter 2. The Power Of Introspection

- 2.2. Optional and named arguments
  Note: Calling functions is flexible
  The only thing you have to do to call a function is specify a value (somehow) for each required argument; the manner and order in which you do that is up to you.
- 2.3. type, str, dir, and other built−in functions
  Note: Python is self−documenting
  Python comes with excellent reference manuals, which you should peruse thoroughly to learn all the modules Python has to offer. But whereas in most languages you would find yourself referring back to the manuals (or man pages, or, God help you, MSDN) to remind yourself how to use these modules, Python is largely self−documenting.
- 2.6. The peculiar nature of and and or
  Important: Using and−or effectively
  The `and−or` trick, `*bool* and a or b`, will not work like the C expression `*bool* ? a : b` when `a` is false in a boolean context.
- 2.7. Using lambda functions
  Note: lambda is optional
  `lambda` functions are a matter of style. Using them is never required; anywhere you could use them, you could define a separate normal function and use that instead. I use them in places where I want to encapsulate specific, non−reusable code without littering my code with a lot of little one−line functions.
- 2.8. Putting it all together
  Note: Python vs. SQL: comparing null values
  In SQL, you must use `IS NULL` instead of `= NULL` to compare a null value. In Python, you can use either `== None` or `is None`, but `is None` is faster.

* 3.2. Importing modules using from module import

  Note: Python vs. Perl: from module import
  `from` *`module`* `import *` in Python is like `use` *`module`* in Perl; `import` *`module`* in Python is like `require` *`module`* in Perl.

  Note: Python vs. Java: from module import
  `from` *`module`* `import *` in Python is like `import` *`module`*`.*` in Java; `import` *`module`* in Python is like `import` *`module`* in Java.
* 3.3. Defining classes

  Note: Python vs. Java: pass
  The `pass` statement in Python is like an empty set of braces (`{}`) in Java or C.

  Note: Python vs. Java: ancestors
  In Python, the ancestor of a class is simply listed in parentheses immediately after the class name. There is no special keyword like `extends` in Java.

  Note: Multiple inheritance
  Although I won't discuss it in depth in this book, Python supports multiple inheritance. In the parentheses following the class name, you can list as many ancestor classes as you like, separated by commas.

  Note: Python vs. Java: self
  By convention, the first argument of any class method (the reference to the current instance) is called `self`. This argument fills the role of the reserved word `this` in C++ or Java, but `self` is not a reserved word in Python, merely a naming convention. Nonetheless, please don't call it anything but `self`; this is a very strong convention.

  Note: When to use self
  When defining your class methods, you *must* explicitly list `self` as the first argument for each method, including `__init__`. When you call a method of an ancestor class from within your class, you *must* include the `self` argument. But when you call your class method from outside, you do not specify anything for the `self` argument; you skip it entirely, and Python automatically adds the instance reference for you. I am aware that this is confusing at first; it's not really inconsistent, but it may appear inconsistent because it relies on a distinction (between bound and unbound methods) that you don't know about yet.

  Note: __init__ methods
  `__init__` methods are optional, but when you define one, you must remember to explicitly call the ancestor's `__init__` method. This is more generally true: whenever a descendant wants to extend the behavior of the ancestor, the descendant method must explicitly call the ancestor method at the proper time, with the proper arguments.
* 3.4. Instantiating classes

  Note: Python vs. Java: instantiating classes
  In Python, simply call a class as if it were a function to create a new instance of the class. There is no explicit `new` operator like C++ or Java.
* 3.5. UserDict: a wrapper class

  Tip: Open modules quickly
  In the Python IDE on Windows, you can quickly open any module in your library path with File−>Locate... (**Ctrl−L**).

Note: Python vs. Java: function overloading
Java and Powerbuilder support function overloading by argument list, *i.e.* one class can have multiple methods with the same name but a different number of arguments, or arguments of different types. Other languages (most notably PL/SQL) even support function overloading by argument name; *i.e.* one class can have multiple methods with the same name and the same number of arguments of the same type but different argument names. Python supports neither of these; it has no form of function overloading whatsoever. Methods are defined solely by their name, and there can be only one method per class with a given name. So if a descendant class has an `__init__` method, it *always* overrides the ancestor `__init__` method, even if the descendant defines it with a different argument list. And the same rule applies to any other method.

Note: Guido on derived classes
Guido, the original author of Python, explains method overriding this way: "Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class, may in fact end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)" If that doesn't make sense to you (it confuses the hell out of me), feel free to ignore it. I just thought I'd pass it along.

Note: Always initialize data attributes
Always assign an initial value to all of an instance's data attributes in the `__init__` method. It will save you hours of debugging later, tracking down `AttributeError` exceptions because you're referencing uninitialized (and therefore non−existent) attributes.

- 3.6. Special class methods

  Note: Calling other class methods
  When accessing data attributes within a class, you need to qualify the attribute name: `self.`*`attribute`*. When calling other methods within a class, you need to qualify the method name: `self.`*`method`*.

- 3.7. Advanced special class methods

  Note: Python vs. Java: equality and identity
  In Java, you determine whether two string variables reference the same physical memory location by using `str1 == str2`. This is called *object identity*, and it is written in Python as `str1 is str2`. To compare string values in Java, you would use `str1.equals(str2)`; in Python, you would use `str1 == str2`. Java programmers who have been taught to believe that the world is a better place because == in Java compares by identity instead of by value may have a difficult time adjusting to Python's lack of such "gotchas".

  Note: Physical vs. logical models
  While other object−oriented languages only let you define the physical model of an object ("this object has a `GetLength` method"), Python's special class methods like `__len__` allow you to define the logical model of an object ("this object has a length").

- 3.8. Class attributes

  Note: Class attributes in Java
  In Java, both static variables (called class attributes in Python) and instance variables (called data attributes in Python) are defined immediately after the class definition (one with the `static` keyword, one without). In Python, only class attributes can be defined here; data attributes are defined in the `__init__` method.

- 3.9. Private functions

  Note: What's private in Python?

If the name of a Python function, class method, or attribute starts with (but doesn't end with) two underscores, it's private; everything else is public.

Note: Method naming conventions
In Python, all special methods (like `__setitem__`) and built−in attributes (like `__doc__`) follow a standard naming convention: they both start with and end with two underscores. Don't name your own methods and attributes this way; it will only confuse you (and others) later.

Note: No protected methods
Python has no concept of protected class methods (accessible only in their own class and descendant classes). Class methods are either private (accessible only in their own class) or public (accessible from anywhere).

- 3.10. Handling exceptions

  Note: Python vs. Java: exception handling
  Python uses `try...except` to handle exceptions and `raise` to generate them. Java and C++ use `try...catch` to handle exceptions, and `throw` to generate them.
- 3.14. The os module

  Note: When to use the os module
  Whenever possible, you should use the functions in `os` and `os.path` for file, directory, and path manipulations. These modules are wrappers for platform−specific modules, so functions like `os.path.split` work on UNIX, Windows, Mac OS, and any other supported Python platform.

Chapter 4. HTML Processing

- 4.2. Introducing sgmllib.py

  Important: Language evolution: DOCTYPE
  Python 2.0 had a bug where `SGMLParser` would not recognize declarations at all (`handle_decl` would never be called), which meant that `DOCTYPE`s were silently ignored. This is fixed in Python 2.1.

  Tip: Specifying command line arguments in Windows
  In the Python IDE on Windows, you can specify command line arguments in the "Run script" dialog. Separate multiple arguments with spaces.
- 4.4. Introducing BaseHTMLProcessor.py

  Important: Processing HTML with embedded script
  The HTML specification requires that all non−HTML (like client−side JavaScript) must be enclosed in HTML comments, but not all web pages do this properly (and all modern web browsers are forgiving if they don't). `BaseHTMLProcessor` is not forgiving; if script is improperly embedded, it will be parsed as if it were HTML. For instance, if the script contains less−than and equals signs, `SGMLParser` may incorrectly think that it has found tags and attributes. `SGMLParser` always converts tags and attribute names to lowercase, which may break the script, and `BaseHTMLProcessor` always encloses attribute values in double quotes (even if the original HTML document used single quotes or no quotes), which will certainly break the script. Always protect your client−side script within HTML comments.
- 4.5. locals and globals

  Important: Language evolution: nested scopes
  Python 2.2 introduced a subtle but important change that affects the namespace search order: nested scopes. In versions of Python prior to 2.2, when you reference a variable within a nested function or `lambda` function, Python will search for that variable in the current

(nested or `lambda`) function's namespace, then in the module's namespace. Python 2.2 will search for the variable in the current (nested or `lambda`) function's namespace, *then in the parent function's namespace*, then in the module's namespace. Python 2.1 can work either way; by default, it works like Python 2.0, but you can add the following line of code at the top of your module to make your module work like Python 2.2:

```
from __future__ import nested_scopes
```

Note: Accessing variables dynamically
Using the `locals` and `globals` functions, you can get the value of arbitrary variables dynamically, providing the variable name as a string. This mirrors the functionality of the `getattr` function, which allows you to access arbitrary functions dynamically by providing the function name as a string.

- 4.6. Dictionary−based string formatting

Important: Performance issues with locals
Using dictionary−based string formatting with `locals` is a convenient way of making complex string formatting expressions more readable, but it comes with a price. There is a slight performance hit in making the call to `locals`, since `locals` builds a copy of the local namespace.

Chapter 5. XML Processing

- 5.2. Packages

Note: What makes a package
A package is a directory with the special `__init__.py` file in it. The `__init__.py` file defines the attributes and methods of the package. It doesn't have to define anything; it can just be an empty file, but it has to exist. But if `__init__.py` doesn't exist, the directory is just a directory, not a package, and it can't be imported or contain modules or nested packages.

- 5.6. Accessing element attributes

Note: XML attributes and Python attributes
This section may be a little confusing, because of some overlapping terminology. Elements in an XML document have attributes, and Python objects also have attributes. When we parse an XML document, we get a bunch of Python objects that represent all the pieces of the XML document, and some of these Python objects represent attributes of the XML elements. But the (Python) objects that represent the (XML) attributes also have (Python) attributes, which are used to access various parts of the (XML) attribute that the object represents. I told you it was confusing. I am open to suggestions on how to distinguish these more clearly.

Note: Attributes have no order
Like a dictionary, attributes of an XML element have no ordering. Attributes may *happen to be* listed in a certain order in the original XML document, and the `Attr` objects may *happen to be* listed in a certain order when the XML document is parsed into Python objects, but these orders are arbitrary and should carry no special meaning. You should always access individual attributes by name, like the keys of a dictionary.

Chapter 6. Unit Testing

- 6.2. Introducing romantest.py

Note: Do you have unittest?
`unittest` is included with Python 2.1 and later. Python 2.0 users can download it from `pyunit.sourceforge.net`.

- 6.8. roman.py, stage 3

Note: Know when to stop coding
The most important thing that comprehensive unit testing can tell you is when to stop coding. When all the unit tests for a function pass, stop coding the function. When all the unit tests for an entire module pass, stop coding the module.

- 6.10. roman.py, stage 5

Note: What to do when all your tests pass
When all your tests pass, stop coding.

- 6.13. Refactoring

Note: Compiling regular expressions
Whenever you are going to use a regular expression more than once, you should compile it to get a pattern object, then call the methods on the pattern object directly.

Chapter 7. Data−Centric Programming

- 7.2. Finding the path

Note: os.path.abspath does not validate pathnames
The pathnames and filenames you pass to `os.path.abspath` do not need to exist.

Note: Normalizing pathnames
`os.path.abspath` not only constructs full path names, it also normalizes them. If you are in the `/usr/` directory, `os.path.abspath('bin/../local/bin'` will return `/usr/local/bin`. If you just want to normalize a pathname without turning it into a full pathname, use `os.path.normpath` instead.

Note: os.path.abspath is cross−platform
Like the other functions in the `os` and `os.path` modules, `os.path.abspath` is cross−platform. Your results will look slightly different than my examples if you're running on Windows (which uses backslash as a path separator) or Mac OS (which uses colons), but they'll still work. That's the whole point of the `os` module.

Appendix D. List of examples

Chapter 1. Getting To Know Python

Appendix E. Revision history

Revision History

Revision 4.1	28 July 2002

- Added *Caching node lookups*.
- Added *Finding direct children of a node*.
- Added *Creating separate handlers by node type*.
- Added *Handling command line arguments*.
- Added *Putting it all together*.
- Added *Summary*.
- Fixed typo in *The os module*. It's `os.getcwd()`, not `os.path.getcwd()`. Thanks, Abhishek.
- Fixed typo in *Joining lists and splitting strings*. When evaluated (instead of printed), the Python IDE will display single quotes around the output.
- Changed `str` example in *Putting it all together* to use a user−defined function, since Python 2.2 obsoleted the old example by defining a `doc string` for the built−in dictionary methods. Thanks Eric.
- Fixed typo in *Unicode*, "anyway" to "anywhere". Thanks Frank.
- Fixed typo in *Testing for sanity*, doubled word "accept". Thanks Ralph.
- Fixed typo in *Refactoring*, `C?C?C?` matches 0 to 3 `C` characters, not 4. Thanks Ralph.
- Clarified and expanded explanation of implied slice indices in Example 1.18. Thanks Petr.
- Added historical note in *UserDict: a wrapper class* now that Python 2.2 supports subclassing built−in datatypes directly.
- Added explanation of `update` dictionary method in Example 3.11. Thanks Petr.
- Clarified Python's lack of overloading in *UserDict: a wrapper class*. Thanks Petr.
- Fixed typo in Example 4.8. HTML comments end with two dashes and a bracket, not one. Thanks Petr.
- Changed tense of note about nested scopes in *locals and globals* now that Python 2.2 is out. Thanks Petr.
- Fixed typo in Example 4.14; a space should have been a non−breaking space. Thanks Petr.
- Added title to note on derived classes in *UserDict: a wrapper class*. Thanks Petr.
- Added title to note on downloading `unittest` in *Refactoring*. Thanks Petr.
- Fixed typesetting problem in Example 7.6; tabs should have been spaces, and the line numbers were misaligned. Thanks Petr.
- Fixed capitalization typo in the tip on truth values in *Introducing lists*. It's `True` and `False`, not `true` and `false`. Thanks to everyone who pointed this out.
- Changed section titles of *Introducing dictionaries*, *Introducing lists*, and *Introducing tuples*. "Dictionaries 101" was a cute way of saying that this section was an beginner's introduction to dictionaries. American colleges tend to use this numbering scheme to indicate introductory courses with no prerequisites, but apparently this is a distinctly American tradition, and it was unnecessarily confusing my international readers. In my defense, when I initially wrote these sections a year and a half ago, it never occurred to me that I would have international readers.
- Upgraded to version 1.52 of the DocBook XSL stylesheets.
- Upgraded to version 6.52 of the SAXON XSLT processor from Michael Kay.
- Various accessibility−related stylesheet tweaks.
- Somewhere between this revision and the last one, she said yes. The wedding will be next spring.

Revision 4.0−2	26 April 2002

- Fixed typo in Example 2.16.
- Fixed typo in Example 1.6.
- Fixed Windows help file (missing table of contents due to base stylesheet changes).

Revision 4.0	19 April 2002

- Expanded *Everything is an object* to include more about import search paths.
- Fixed typo in Example 1.16. Thanks to Brian for the correction.
- Rewrote the tip on truth values in *Introducing lists*, now that Python has a separate boolean datatype.
- Fixed typo in *Importing modules using from module import* when comparing syntax to Java. Thanks to Rick for the correction.
- Added note in *UserDict: a wrapper class* about derived classes always overriding ancestor classes.
- Fixed typo in Example 3.19. Thanks to Kevin for the correction.
- Added note in *Handling exceptions* that you can define and raise your own exceptions. Thanks to Rony for the suggestion.
- Fixed typo in Example 4.16. Thanks for Rick for the correction.
- Added note in Example 4.17 about what the return codes mean. Thanks to Howard for the suggestion.
- Added `str` function when creating `StringIO` instance in Example 5.30. Thanks to Ganesan for the idea.
- Added link in *Introducing romantest.py* to explanation of why test cases belong in a separate file.
- Changed *Finding the path* to use `os.path.dirname` instead of `os.path.split`. Thanks to Marc for the idea.
- Added code samples (`piglatin.py`, `parsephone.py`, and `plural.py`) for the upcoming regular expressions chapter.
- Updated and expanded list of Python distributions on home page.

Revision 3.9	1 January 2002

- Added *Unicode*.
- Added *Searching for elements*.
- Added *Accessing element attributes*.
- Added *Abstracting input sources*.
- Added *Standard input, output, and error*.
- Added simple counter `for` loop examples (good usage and bad usage) in *for loops*. Thanks to Kevin for the idea.
- Fixed typo in Example 1.33 (two elements of `params.values()` were reversed).
- Fixed mistake in *type, str, dir, and other built−in functions* with regards to the name of the `__builtin__` module. Thanks to Denis for the correction.
- Added additional example in *Finding the path* to show how to run unit tests in the current working directory, instead of the directory where `regression.py` is located.
- Modified explanation of how to derive a negative list index from a positive list index in Example 1.16. Thanks to Renauld for the suggestion.
- Updated links on home page for downloading latest version of Python.
- Added link on home page to Bruce Eckel's preliminary draft of *Thinking in Python*, a marvelous (and advanced) book on design patterns and how to implement them in Python.

Revision 3.8	18 November 2001

- Added *Finding the path*.
- Added *Filtering lists revisited*.
- Added *Mapping lists revisited*.
- Added *Data−centric programming*.
- Expanded sample output in *Diving in*.
- Finished *Parsing XML*.

Revision 3.7	30 September 2001

- Added *Packages*.
- Added *Parsing XML*.
- Cleaned up introductory paragraph in *Diving in*. Thanks to Matt for this suggestion.
- Added Java tip in *Importing modules using from module import*. Thanks to Ori for this suggestion.

* Fixed mistake in *Putting it all together* where I implied that you could not use `is None` to compare to a null value in Python. In fact, you can, and it's faster than `== None`. Thanks to Ori pointing this out.
* Clarified in *Introducing lists* where I said that `li = li + other` was equivalent to `li.extend(other)`. The result is the same, but `extend` is faster because it doesn't create a new list. Thanks to Denis pointing this out.
* Fixed mistake in *Introducing lists* where I said that `li += other` was equivalent to `li = li + other`. In fact, it's equivalent to `li.extend(other)`, since it doesn't create a new list. Thanks to Denis pointing this out.
* Fixed typographical laziness in *Getting To Know Python*; when I was writing it, I had not yet standardized on putting string literals in single quotes within the text. They were set off by typography, but this is lost in some renditions of the book (like plain text), making it difficult to read. Thanks to Denis for this suggestion.
* Fixed mistake in *Declaring functions* where I said that statically typed languages always use explicit variable + datatype declarations to enforce static typing. Most do, but there are some statically typed languages where the compiler figures out what type the variable is based on usage within the code. Thanks to Tony for pointing this out.
* Added link to Spanish translation.

Revision 3.6.4	6 September 2001

* Added code in `BaseHTMLProcessor` to handle non−HTML entity references, and added a note about it in *Introducing BaseHTMLProcessor.py*.
* Modified Example 4.11 to include `htmlentitydefs` in the output.

Revision 3.6.3	4 September 2001

* Fixed typo in *Diving in*.
* Added link to Korean translation.

Revision 3.6.2	31 August 2001

* Fixed typo in *Testing for sanity* (the last requirement was listed twice).

Revision 3.6	31 August 2001

* Finished *HTML Processing* with *Putting it all together* and *Summary*.
* Added *Postscript*.
* Started *XML Processing* with *Diving in*.
* Started *Data−Centric Programming* with *Diving in*.
* Fixed long−standing bug in colorizing script that improperly colorized the examples in *HTML Processing*.
* Added link to French translation. They did the right thing and translated the source XML, so they can re−use all my build scripts and make their work available in six different formats.
* Upgraded to version 1.43 of the DocBook XSL stylesheets.
* Upgraded to version 6.43 of the SAXON XSLT processor from Michael Kay.
* Massive stylesheet changes, moving away from a table−based layout and towards more appropriate use of cascading style sheets. Unfortunately, CSS has as many compatibility problems as anything else, so there are still some tables used in the header and footer. The resulting HTML version looks worse in Netscape 4, but better in modern browsers, including Netscape 6, Mozilla, Internet Explorer 5, Opera 5, Konqueror, and iCab. And it's still completely readable in Lynx. I love Lynx. It was my first web browser. You never forget your first.
* Moved to Ant to have better control over the build process, which is especially important now that I'm juggling six output formats and two languages.
* Consolidated the available downloadable archives; previously, I had different files for each platform, because the .zip files that Python's `zipfile` module creates are non−standard and can't be opened by Aladdin Expander on Mac OS. But the .zip files that Ant creates are completely standard and cross−platform. Go Ant!

<table>
<tr><td colspan="2">

* Now hosting the complete XML source, XSL stylesheets, and associated scripts and libraries on SourceForge. There's also CVS access for the really adventurous.
* Re−licensed the example code under the new−and−improved GPL−compatible Python 2.1.1 license. Thanks, Guido; people really do care, and it really does matter.

</td></tr>
<tr><td>Revision 3.5</td><td>26 June 2001</td></tr>
<tr><td colspan="2">

* Added explanation of strong/weak/static/dynamic datatypes in *Declaring functions*.
* Added case−sensitivity example in *Introducing dictionaries*.
* Use `os.path.normcase` in *An Object−Oriented Framework* to compensate for inferior operating systems whose files aren't case−sensitive.
* Fixed indentation problems in code samples in PDF version.

</td></tr>
<tr><td>Revision 3.4</td><td>31 May 2001</td></tr>
<tr><td colspan="2">

* Added *roman.py, stage 5*.
* Added *Handling bugs*.
* Added *Handling changing requirements*.
* Added *Refactoring*.
* Added *Summary*.
* Fixed yet another stylesheet bug that was dropping nested `</span>` tags.

</td></tr>
<tr><td>Revision 3.3</td><td>24 May 2001</td></tr>
<tr><td colspan="2">

* Added *Diving in*.
* Added *Introducing romantest.py*.
* Added *Testing for success*.
* Added *Testing for failure*.
* Added *Testing for sanity*.
* Added *roman.py, stage 1*.
* Added *roman.py, stage 2*.
* Added *roman.py, stage 3*.
* Added *roman.py, stage 4*.
* Tweaked stylesheets in an endless quest for complete Netscape/Mozilla compatibility.

</td></tr>
<tr><td>Revision 3.2</td><td>3 May 2001</td></tr>
<tr><td colspan="2">

* Added *Introducing dialect.py*.
* Added *Regular expressions 101*.
* Fixed bug in `handle_decl` method that would produce incorrect declarations (adding a space where it couldn't be).
* Fixed bug in CSS (introduced in 2.9) where body background color was missing.

</td></tr>
<tr><td>Revision 3.1</td><td>18 Apr 2001</td></tr>
<tr><td colspan="2">

* Added code in `BaseHTMLProcessor.py` to handle declarations, now that Python 2.1 supports them.
* Added note about nested scopes in *locals and globals*.
* Fixed obscure bug in Example 4.1 where attribute values with character entities would not be properly escaped.
* Now recommending (but not requiring) Python 2.1, due to its support of declarations in `sgmllib.py`.
* Updated download links on the home page to point to Python 2.1, where available.
* Moved to versioned filenames, to help people who redistribute the book.

</td></tr>
<tr><td>Revision 3.0</td><td>16 Apr 2001</td></tr>
<tr><td colspan="2"></td></tr>
</table>

• Fixed minor bug in code listing in *HTML Processing*. • Added link to Chinese translation on home page.	
Revision 2.9	13 Apr 2001
• Added *locals and globals*. • Added *Dictionary−based string formatting*. • Tightened code in *HTML Processing*, specifically `ChefDialectizer`, to use fewer and simpler regular expressions. • Fixed a stylesheet bug that was inserting blank pages between chapters in the PDF version. • Fixed a script bug that was stripping the `DOCTYPE` from the home page. • Added link to Python Cookbook, and added a few links to individual recipes in *Further reading*. • Switched to Google for searching on `http://diveintopython.org/`. • Upgraded to version 1.36 of the DocBook XSL stylesheets, which was much more difficult than it sounds. There may still be lingering bugs.	
Revision 2.8	26 Mar 2001
• Added *Extracting data from HTML documents*. • Added *Introducing BaseHTMLProcessor.py*. • Added *Quoting attribute values*. • Tightened up code in *The Power Of Introspection*, using the built−in function `callable` instead of manually checking types. • Moved *Importing modules using from module import* from *The Power Of Introspection* to *An Object−Oriented Framework*. • Fixed typo in code example in *Diving in* (added colon). • Added several additional downloadable example scripts. • Added Windows Help output format.	
Revision 2.7	16 Mar 2001
• Added *Introducing sgmllib.py*. • Tightened up code in *HTML Processing*. • Changed code in *Getting To Know Python* to use `items` method instead of `keys`. • Moved *Assigning multiple values at once* section to *Getting To Know Python*. • Edited note about `join` string method, and provided a link to the new entry in *The Whole Python FAQ* that explains why `join` is a string method instead of a list method. • Rewrote *The peculiar nature of and and or* to emphasize the fundamental nature of `and` and `or` and de−emphasize the `and−or` trick. • Reorganized language comparisons into `note`s.	
Revision 2.6	28 Feb 2001
• The PDF and Word versions now have colorized examples, an improved table of contents, and properly indented `tip`s and `note`s. • The Word version is now in native Word format, compatible with Word 97. • The PDF and text versions now have fewer problems with improperly converted special characters (like trademark symbols and curly quotes). • Added link to download Word version for UNIX, in case some twisted soul wants to import it into StarOffice or something. • Fixed several `note`s which were missing titles. • Fixed stylesheets to work around bug in Internet Explorer 5 for Mac OS which caused colorized words in the examples to be displayed in the wrong font. (Hello?!? Microsoft? Which part of `<pre>` don't you understand?)	

- Fixed archive corruption in Mac OS downloads.
- In first section of each chapter, added link to download examples. (My access logs show that people skim or skip the two pages where they could have downloaded them (the home page and *Preface*), then scramble to find a download link once they actually start reading.)
- Tightened the home page and *Preface* even more, in the hopes that someday someone will read them.
- Soon I hope to get back to actually writing this book instead of debugging it.

Revision 2.5	23 Feb 2001

- Added *More on modules*.
- Added *The os module*.
- Moved Example 3.36 from *Assigning multiple values at once* to *The os module*.
- Added *Putting it all together*.
- Added *Summary*.
- Added *Diving in*.
- Fixed program listing in Example 3.29 which was missing a colon.

Revision 2.4.1	12 Feb 2001

- Changed newsgroup links to use "news:" protocol, now that `deja.com` is defunct.
- Added file sizes to download links.

Revision 2.4	12 Feb 2001

- Added "further reading" links in most sections, and collated them in *Further reading*.
- Added URLs in parentheses next to external links in text version.

Revision 2.3	9 Feb 2001

- Rewrote some of the code in *An Object−Oriented Framework* to use class attributes and a better example of multi−variable assignment.
- Reorganized *An Object−Oriented Framework* to put the class sections first.
- Added *Class attributes*.
- Added *Handling exceptions*.
- Added *File objects*.
- Merged the "review" section in *An Object−Oriented Framework* into *Diving in*.
- Colorized all program listings and examples.
- Fixed important error in *Declaring functions*: functions that do not explicitly return a value return `None`, so you *can* assign the return value of such a function to a variable without raising an exception.
- Added minor clarifications to *Documenting functions*, *Everything is an object*, and *Defining variables*.

Revision 2.2	2 Feb 2001

- Edited *Getting object references with getattr*.
- Added titles to `xref` tags, so they can have their cute little tooltips too.
- Changed the look of the revision history page.
- Fixed problem I introduced yesterday in my HTML post−processing script that was causing invalid HTML character references and breaking some browsers.
- Upgraded to version 1.29 of the DocBook XSL stylesheets.

Revision 2.1	1 Feb 2001

- Rewrote the example code of *The Power Of Introspection* to use `getattr` instead of `exec` and `eval`, and rewrote explanatory text to match.
- Added example of list operators in *Introducing lists*.

- Added links to relevant sections in the summary lists at the end of each chapter (*Summary* and *Summary*).

Revision 2.0	31 Jan 2001

- Split *Special class methods* into three sections, *UserDict: a wrapper class*, *Special class methods*, and *Advanced special class methods*.
- Changed notes on garbage collection to point out that Python 2.0 and later can handle circular references without additional coding.
- Fixed UNIX downloads to include all relevant files.

Revision 1.9	15 Jan 2001

- Removed introduction to *Getting To Know Python*.
- Removed introduction to *The Power Of Introspection*.
- Removed introduction to *An Object−Oriented Framework*.
- Edited text ruthlessly. I tend to ramble.

Revision 1.8	12 Jan 2001

- Added more examples to *Assigning multiple values at once*.
- Added *Defining classes*.
- Added *Instantiating classes*.
- Added *Special class methods*.
- More minor stylesheet tweaks, including adding titles to `link` tags, which, if your browser is cool enough, will display a description of the link target in a cute little tooltip.

Revision 1.71	3 Jan 2001

- Made several modifications to stylesheets to improve browser compatibility.

Revision 1.7	2 Jan 2001

- Added introduction to *Getting To Know Python*.
- Added introduction to *The Power Of Introspection*.
- Added review section to *An Object−Oriented Framework* [later removed]
- Added *Private functions*.
- Added *for loops*.
- Added *Assigning multiple values at once*.
- Wrote scripts to convert book to new output formats: one single HTML file, PDF, Microsoft Word 97, and plain text.
- Registered the `diveintopython.org` domain and moved the book there, along with links to download the book in all available output formats for offline reading.
- Modified the XSL stylesheets to change the header and footer navigation that displays on each page. The top of each page is branded with the domain name and book version, followed by a breadcrumb trail to jump back to the chapter table of contents, the main table of contents, or the site home page.

Revision 1.6	11 Dec 2000

- Added *Putting it all together*.
- Finished *The Power Of Introspection* with *Summary*.
- Started *An Object−Oriented Framework* with *Diving in*.

Revision 1.5	22 Nov 2000

- Added *The peculiar nature of and and or*.
- Added *Using lambda functions*.

- Added appendix that lists section abstracts.
- Added appendix that lists tips.
- Added appendix that lists examples.
- Added appendix that lists revision history.
- Expanded example of mapping lists in *Mapping lists*.
- Encapsulated several more common phrases into entities.
- Upgraded to version 1.25 of the DocBook XSL stylesheets.

Revision 1.4	14 Nov 2000

- Added *Filtering lists*.
- Added `dir` documentation to *type, str, dir, and other built−in functions*.
- Added `in` example in *Introducing tuples*.
- Added additional note about `if __name__` trick under MacPython.
- Switched to the SAXON XSLT processor from Michael Kay.
- Upgraded to version 1.24 of the DocBook XSL stylesheets.
- Added db−html processing instructions with explicit filenames of each chapter and section, to allow deep links to content even if I add or re−arrange sections later.
- Made several common phrases into entities for easier reuse.
- Changed several `literal` tags to `constant`.

Revision 1.3	9 Nov 2000

- Added section on dynamic code execution.
- Added links to relevant section/example wherever I refer to previously covered concepts.
- Expanded introduction of chapter 2 to explain what the function actually does.
- Explicitly placed example code under the GNU General Public License and added appendix to display license. [Note 8/16/2001: code has been re−licensed under GPL−compatible Python license]
- Changed links to licenses to use `xref` tags, now that I know how to use them.

Revision 1.2	6 Nov 2000

- Added first four sections of chapter 2.
- Tightened up preface even more, and added link to Mac OS version of Python.
- Filled out examples in "Mapping lists" and "Joining strings" to show logical progression.
- Added output in chapter 1 summary.

Revision 1.1	31 Oct 2000

- Finished chapter 1 with sections on mapping and joining, and a chapter summary.
- Toned down the preface, added links to introductions for non−programmers.
- Fixed several typos.

Revision 1.0	30 Oct 2000

- Initial publication

Appendix F. About the book

This book was written in DocBook XML using Emacs, and converted to HTML using the SAXON XSLT processor from Michael Kay with a customized version of Norman Walsh's XSL stylesheets. From there, it was converted to PDF using HTMLDoc, and to plain text using w3m. Program listings and examples were colorized using an updated version of Just van Rossum's `pyfontify.py`, which is included in the example scripts.

If you're interested in learning more about DocBook for technical writing, you can download the XML source and the build scripts, which include the customized XSL stylesheets used to create all the different formats of the book. You should also read the canonical book, *DocBook: The Definitive Guide*. If you're going to do any serious writing in DocBook, I would recommend subscribing to the DocBook mailing lists.

Appendix G. GNU Free Documentation License

Version 1.1, March 2000

G.0. Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

G.1. Applicability and definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

G.2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

G.3. Copying in quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

G.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties−−for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

G.5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

G.6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

G.7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self−contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

G.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

G.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

G.10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

G.11. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front−Cover Texts being LIST, and with the Back−Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front−Cover Texts, write "no Front−Cover Texts" instead of "Front−Cover Texts being LIST"; likewise for Back−Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix H. Python 2.1.1 license

H.A. History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands as a successor of a language called ABC. Guido is Python's principal author, although it includes many contributions from others. The last version released from CWI was Python 1.2. In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI) in Reston, Virginia where he released several versions of the software. Python 1.6 was the last of the versions released by CNRI. In 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. Python 2.0 was the first and only release from BeOpen.com.

Following the release of Python 1.6, and after Guido van Rossum left CNRI to work with commercial software developers, it became clear that the ability to use Python with software available under the GNU Public License (GPL) was very desirable. CNRI and the Free Software Foundation (FSF) interacted to develop enabling wording changes to the Python license. Python 1.6.1 is essentially the same as Python 1.6, with a few minor bug fixes, and with a different license that enables later versions to be GPL−compatible. Python 2.1 is a derivative work of Python 1.6.1, as well as of Python 2.0.

After Python 2.0 was released by BeOpen.com, Guido van Rossum and the other PythonLabs developers joined Digital Creations. All intellectual property added from this point on, starting with Python 2.1 and its alpha and beta releases, is owned by the Python Software Foundation (PSF), a non−profit modeled after the Apache Software Foundation. See http://www.python.org/psf/ for more information about the PSF.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

H.B. Terms and conditions for accessing or otherwise using Python

H.B.1. PSF license agreement

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.1.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.1.1 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001 Python Software Foundation; All Rights Reserved" are retained in Python 2.1.1 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.1.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.1.1.
4. PSF is making Python 2.1.1 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.1.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.1.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.1.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.1.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.2. BeOpen Python open source license agreement version 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non−exclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.3. CNRI open source GPL−compatible license agreement

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright (c) 1995−2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement

may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non−separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.4. CWI permissions statement and disclaimer

> Copyright (c) 1991 − 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.