Modules and files

10.1. Modules

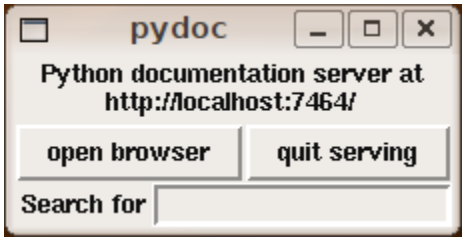
A **module** is a file containing Python definitions and statements intended for use in other Python programs. There are many Python modules that come with Python as part of the **standard library**. We have seen two of these already, the doctest module and the string module.

10.2. pydoc

You can use **pydoc** to search through the Python libraries installed on your system. At the **command prompt** type the following:

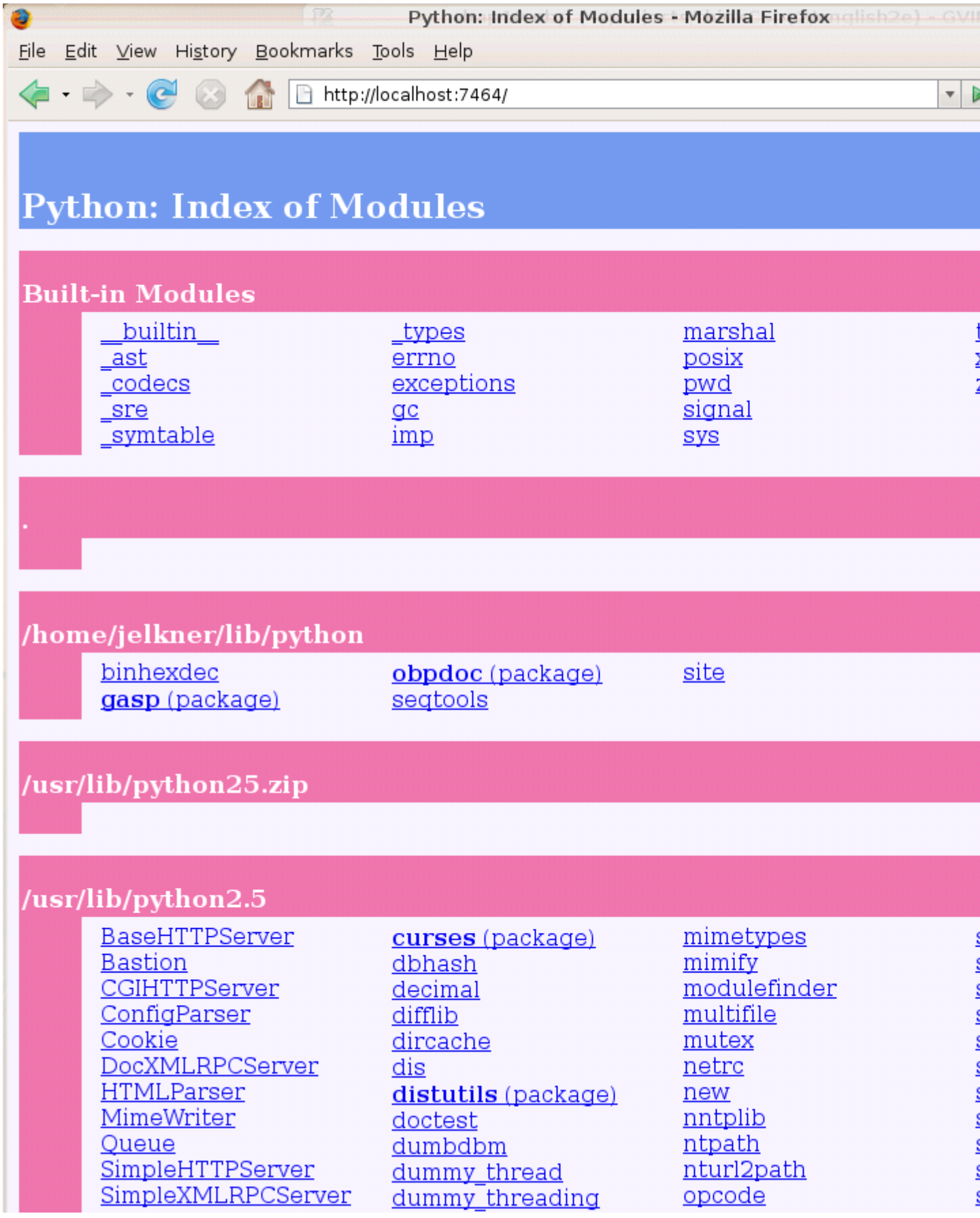
```
$ pydoc -g
```

and the following will appear:

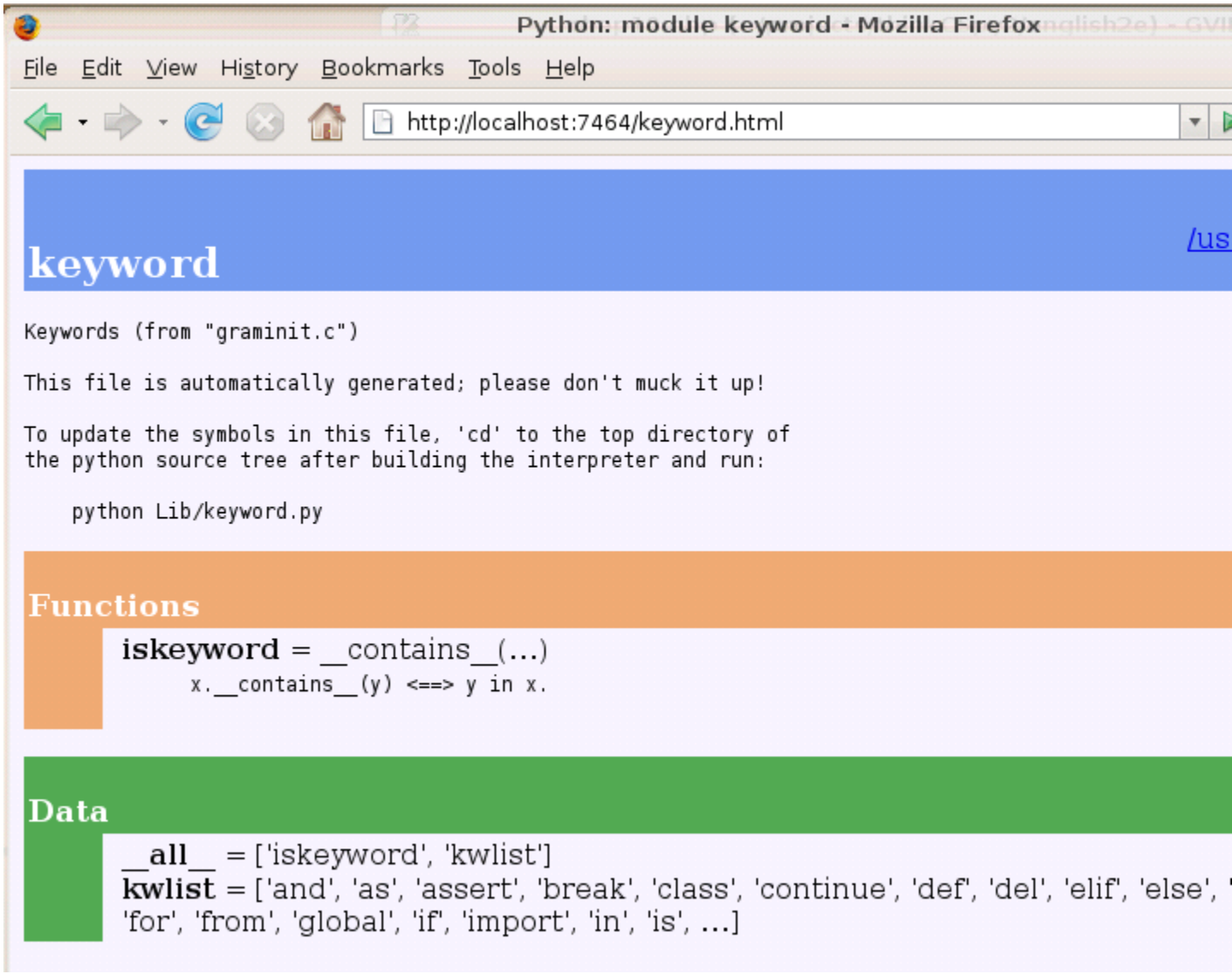


(*note*: see exercise 2 if you get an error)

Click on the open browser button to launch a web browser window containing the documentation generated by pydoc:


Python: Index of Modules - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:7464/
Python: Index of Modules
Built-in Modules
__builtin__ _ast _codecs _sre _symtable
_types errno exceptions gc imp
marshal posix pwd signal sys
.
/home/jelkner/lib/python
binhexdec gasp (package)
obpdoc (package) seqtools
site
/usr/lib/python25.zip
/usr/lib/python2.5
BaseHTTPServer Bastion CGIHTTPServer ConfigParser Cookie DocXMLRPCServer HTMLParser MimeWriter Queue SimpleHTTPServer SimpleXMLRPCServer
curses (package) dbhash decimal difflib dircache dis distutils (package) doctest dumbdbm dummy_thread dummy_threading
mimetypes mimify modulefinder multifile mutex netrc new nntplib ntpath nturl2path opcode

This is a listing of all the python libraries found by Python on your system. Clicking on a module name opens a new page with documenation for that module. Clickingkeyword, for example, opens the following page:



Documentation for most modules contains three color coded sections:

- *Classes* in pink
- *Functions* in orange
- *Data* in green

Classes will be discussed in later chapters, but for now we can use pydoc to see the functions and data contained within modules.

The keyword module contains a single function, iskeyword, which as its name suggests is a boolean function that returns True if a string passed to it is a keyword:

```
>>> from keyword import *
>>> iskeyword('for')
True
>>> iskeyword('all')
False
>>>
```

The data item, kwlist contains a list of all the current keywords in Python:

```
>>> from keyword import *
>>> print kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import',
'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try',
'while', 'with', 'yield']
>>>
```

We encourage you to use pydoc to explore the extensive libraries that come with Python. There are so many treasures to discover!

10.3. Creating modules

All we need to create a module is a text file with a .py extension on the filename:

```
# seqtools.py
#
def remove_at(pos, seq):
    return seq[:pos] + seq[pos+1:]
```

We can now use our module in both scripts and the Python shell. To do so, we must first *import* the module. There are two ways to do this:

```
>>> from seqtools import remove_at
>>> s = "A string!"
>>> remove_at(4, s)
'A sting!'
```

and:

```
>>> import seqtools
>>> s = "A string!"
>>> seqtools.remove_at(4, s)
'A sting!'
```

In the first example, remove_at is called just like the functions we have seen previously. In the second example the name of the module and a dot (.) are written before the function name.

Notice that in either case we do not include the .py file extension when importing. Python expects the file names of Python modules to end in .py, so the file extention is not included in the **import statement**.

The use of modules makes it possible to break up very large programs into manageable sized parts, and to keep related parts together.

10.4. Namespaces

A **namespace** is a syntactic container which permits the same name to be used in different modules or functions (and as we will see soon, in classes and methods).

Each module determines its own namespace, so we can use the same name in multiple modules without causing an identification problem.

```
# module1.py

question = "What is the meaning of life, the Universe, and everything?"
answer = 42
# module2.py

question = "What is your quest?"
answer = "To seek the holy grail."
```

We can now import both modules and access question and answer in each:

```
>>> import module1
>>> import module2
>>> print module1.question
What is the meaning of life, the Universe, and everything?
```

```
>>> print module2.question
What is your quest?
>>> print module1.answer
42
>>> print module2.answer
To seek the holy grail.
>>>
```

If we had used from module1 import * and from module2 import * instead, we would have a **naming collision** and would not be able to access question and answer frommodule1.

Functions also have their own namespace:

```
def f():
    n = 7
    print "printing n inside of f: %d" % n

def g():
    n = 42
    print "printing n inside of g: %d" % n

n = 11
print "printing n before calling f: %d" % n
f()
print "printing n after calling f: %d" % n
g()
print "printing n after calling g: %d" % n
```

Running this program produces the following output:

```
printing n before calling f: 11
printing n inside of f: 7
printing n after calling f: 11
printing n inside of g: 42
printing n after calling g: 11
```

The three n‘s here do not collide since they are each in a different namespace.

Namespaces permit several programmers to work on the same project without having naming collisions.

10.5. Attributes and the dot operator

Variables defined inside a module are called **attributes** of the module. They are accessed by using the **dot operator** (.). The question attribute of module1 and module2are accessed using module1.question and module2.question.

Modules contain functions as well as attributes, and the dot operator is used to access them in the same way. seqtools.remove_at refers to the remove_at function in theseqtools module.

In Chapter 7 we introduced the find function from the string module. The string module contains many other useful functions:

```
>>> import string
>>> string.capitalize('maryland')
'Maryland'
>>> string.capwords("what's all this, then, amen?")
"What's All This, Then, Amen?"
>>> string.center('How to Center Text Using Python', 70)
'                How to Center Text Using Python                 '
>>> string.upper('angola')
'ANGOLA'
>>>
```

You should use pydoc to browse the other functions and attributes in the string module.

10.6. String and list methods

As the Python language developed, most of functions from the string module have also been added as **methods** of string objects. A method acts much like a function, but the syntax for calling it is a bit different:

```
>>> 'maryland'.capitalize()
'Maryland'
>>> "what's all this, then, amen?".title()
"What'S All This, Then, Amen?"
>>> 'How to Center Text Using Python'.center(70)
```

```
'                How to Center Text Using Python                '
>>> 'angola'.upper()
'ANGOLA'
>>>
```

String methods are built into string objects, and they are *invoked* (called) by following the object with the dot operator and the method name.

We will be learning how to create our own objects with their own methods in later chapters. For now we will only be using methods that come with Python's built-in objects.

The dot operator can also be used to access built-in methods of list objects:

```
>>> mylist = []
>>> mylist.append(5)
>>> mylist.append(27)
>>> mylist.append(3)
>>> mylist.append(12)
>>> mylist
[5, 27, 3, 12]
>>>
```

append is a list method which adds the argument passed to it to the end of the list. Continuing with this example, we show several other list methods:

```
>>> mylist.insert(1, 12)
>>> mylist
[5, 12, 27, 3, 12]
>>> mylist.count(12)
2
>>> mylist.extend([5, 9, 5, 11])
>>> mylist
[5, 12, 27, 3, 12, 5, 9, 5, 11])
>>> mylist.index(9)
6
>>> mylist.count(5)
3
>>> mylist.reverse()
```

```
>>> mylist
[11, 5, 9, 5, 12, 3, 27, 12, 5]
>>> mylist.sort()
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 12, 27]
>>> mylist.remove(12)
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 27]
>>>
```

Experiment with the list methods in this example until you feel confident that you understand how they work.

10.7. Reading and writing text files

While a program is running, its data is stored in *random access memory* (RAM). RAM is fast and inexpensive, but it is also **volatile**, which means that when the program ends, or the computer shuts down, data in RAM disappears. To make data available the next time you turn on your computer and start your program, you have to write it to a **non-volatile** storage medium, such as a hard drive, usb drive, or CD-RW.

Data on non-volatile storage media is stored in named locations on the media called **files**. By reading and writing files, programs can save information between program runs.

Working with files is a lot like working with a notebook. To use a notebook, you have to open it. When you're done, you have to close it. While the notebook is open, you can either write in it or read from it. In either case, you know where you are in the notebook. You can read the whole notebook in its natural order or you can skip around.

All of this applies to files as well. To open a file, you specify its name and indicate whether you want to read or write.

Opening a file creates a file object. In this example, the variable myfile refers to the new file object.

```
>>> myfile = open('test.dat', 'w')
>>> print myfile
<open file 'test.dat', mode 'w' at 0x2aaaaab80cd8>
```

The open function takes two arguments. The first is the name of the file, and the second is the **mode**. Mode 'w' means that we are opening the file for writing.

If there is no file named test.dat, it will be created. If there already is one, it will be replaced by the file we are writing.

When we print the file object, we see the name of the file, the mode, and the location of the object.

To put data in the file we invoke the write method on the file object:

```
>>> myfile.write("Now is the time")
>>> myfile.write("to close the file")
```

Closing the file tells the system that we are done writing and makes the file available for reading:

```
>>> myfile.close()
```

Now we can open the file again, this time for reading, and read the contents into a string. This time, the mode argument is 'r' for reading:

```
>>> myfile = open('test.dat', 'r')
```

If we try to open a file that doesn't exist, we get an error:

```
>>> myfile = open('test.cat', 'r')
IOError: [Errno 2] No such file or directory: 'test.cat'
```

Not surprisingly, the read method reads data from the file. With no arguments, it reads the entire contents of the file into a single string:

```
>>> text = myfile.read()
>>> print text
Now is the timeto close the file
```

There is no space between time and to because we did not write a space between the strings.

read can also take an argument that indicates how many characters to read:

```
>>> myfile = open('test.dat', 'r')
>>> print myfile.read(5)
Now i
```

If not enough characters are left in the file, read returns the remaining characters. When we get to the end of the file, read returns the empty string:

```
>>> print myfile.read(1000006)
s the timeto close the file
>>> print myfile.read()

>>>
```

The following function copies a file, reading and writing up to fifty characters at a time. The first argument is the name of the original file; the second is the name of the new file:

```
def copy_file(oldfile, newfile):
    infile = open(oldfile, 'r')
    outfile = open(newfile, 'w')
    while True:
        text = infile.read(50)
        if text == "":
            break
        outfile.write(text)
    infile.close()
    outfile.close()
    return
```

This functions continues looping, reading 50 characters from infile and writing the same 50 charaters to outfile until the end of infile is reached, at which point text is empty and the break statement is executed.

10.8. Text files

A **text file** is a file that contains printable characters and whitespace, organized into lines separated by newline characters. Since Python is specifically designed to process text files, it provides methods that make the job easy.

To demonstrate, we'll create a text file with three lines of text separated by newlines:

```
>>> outfile = open("test.dat","w")
>>> outfile.write("line one\nline two\nline three\n")
>>> outfile.close()
```

The readline method reads all the characters up to and including the next newline character:

```
>>> infile = open("test.dat","r")
>>> print infile.readline()
line one

>>>
```

readlines returns all of the remaining lines as a list of strings:

```
>>> print infile.readlines()
['line two\012', 'line three\012']
```

In this case, the output is in list format, which means that the strings appear with quotation marks and the newline character appears as the escape sequence \\012.

At the end of the file, readline returns the empty string and readlines returns the empty list:

```
>>> print infile.readline()

>>> print infile.readlines()
[]
```

The following is an example of a line-processing program. filter makes a copy of oldfile, omitting any lines that begin with #:

```
def filter(oldfile, newfile):
    infile = open(oldfile, 'r')
    outfile = open(newfile, 'w')
    while True:
        text = infile.readline()
        if text == "":
```

```
            break
        if text[0] == '#':
            continue
        outfile.write(text)
    infile.close()
    outfile.close()
    return
```

The **continue statement** ends the current iteration of the loop, but continues looping. The flow of execution moves to the top of the loop, checks the condition, and proceeds accordingly.

Thus, if text is the empty string, the loop exits. If the first character of text is a hash mark, the flow of execution goes to the top of the loop. Only if both conditions fail do we copy text into the new file.

10.9. Directories

Files on non-volatile storage media are organized by a set of rules known as a **file system**. File systems are made up of files and **directories**, which are containers for both files and other directories.

When you create a new file by opening it and writing, the new file goes in the current directory (wherever you were when you ran the program). Similarly, when you open a file for reading, Python looks for it in the current directory.

If you want to open a file somewhere else, you have to specify the **path** to the file, which is the name of the directory (or folder) where the file is located:

```
>>> wordsfile = open('/usr/share/dict/words', 'r')
>>> wordlist = wordsfile.readlines()
>>> print wordlist[:6]
['\n', 'A\n', "A's\n", 'AOL\n', "AOL's\n", 'Aachen\n']
```

This example opens a file named words that resides in a directory named dict, which resides in share, which resides in usr, which resides in the top-level directory of the system, called /. It then reads in each line into a list using readlines, and prints out the first 5 elements from that list.

You cannot use / as part of a filename; it is reserved as a **delimiter** between directory and filenames.

The file /usr/share/dict/words should exist on unix based systems, and contains a list of words in alphabetical order.

10.10. Counting Letters

The ord function returns the integer representation of a character:

```
>>> ord('a')
97
>>> ord('A')
65
>>>
```

This example explains why 'Apple' < 'apple' evaluates to True.

The chr function is the inverse of ord. It takes an integer as an argument and returns its character representation:

```
>>> for i in range(65, 71):
...     print chr(i)
...
A
B
C
D
E
F
>>>
```

The following program, countletters.py counts the number of times each character occurs in the book Alice in Wonderland:

```
#
# countletters.py
#

def display(i):
```

```
    if i == 10: return 'LF'
    if i == 13: return 'CR'
    if i == 32: return 'SPACE'
    return chr(i)

infile = open('alice_in_wonderland.txt', 'r')
text = infile.read()
infile.close()

counts = 128 * [0]

for letter in text:
    counts[ord(letter)] += 1

outfile = open('alice_counts.dat', 'w')
outfile.write("%-12s%s\n" % ("Character", "Count"))
outfile.write("=================\n")

for i in range(len(counts)):
    if counts[i]:
        outfile.write("%-12s%d\n" % (display(i), counts[i]))

outfile.close()
```

Run this program and look at the output file it generates using a text editor. You will be asked to analyze the program in the exercises below.

10.11. The sys module and argv

The sys module contains functions and variables which provide access to the *environment* in which the python interpreter runs.

The following example shows the values of a few of these variables on one of our systems:

```
>>> import sys
>>> sys.platform
'linux2'
>>> sys.path
```

```
['', '/home/jelkner/lib/python', '/usr/lib/python25.zip', '/usr/lib/python2.5',
'/usr/lib/python2.5/plat-linux2', '/usr/lib/python2.5/lib-tk',
'/usr/lib/python2.5/lib-dynload', '/usr/local/lib/python2.5/site-packages',
'/usr/lib/python2.5/site-packages', '/usr/lib/python2.5/site-packages/Numeric',
'/usr/lib/python2.5/site-packages/gst-0.10',
'/var/lib/python-support/python2.5', '/usr/lib/python2.5/site-packages/gtk-2.0',
'/var/lib/python-support/python2.5/gtk-2.0']
>>> sys.version
'2.5.1 (r251:54863, Mar  7 2008, 04:10:12) \n[GCC 4.1.3 20070929 (prerelease)
(Ubuntu 4.1.2-16ubuntu2)]'
>>>
```

Starting **Jython** on the same machine produces different values for the same variables:

```
>>> import sys
>>> sys.platform
'java1.6.0_03'
>>> sys.path
['', '/home/jelkner/.', '/usr/share/jython/Lib', '/usr/share/jython/Lib-cpython']
>>> sys.version
'2.1'
>>>
```

The results will be different on your machine of course.

The argv variable holds a list of strings read in from the **command line** when a Python script is run. These **command line arguments** can be used to pass information into a program at the same time it is invoked.

```
#
# demo_argv.py
#
import sys

print sys.argv
```

Running this program from the unix command prompt demonstrates how sys.argv works:

```
$ python demo_argv.py this and that 1 2 3
['demo_argv.py', 'this', 'and', 'that', '1', '2', '3']
$
```

argv is a list of strings. Notice that the first element is the name of the program. Arguments are separated by white space, and separated into a list in the same way thatstring.split operates. If you want an argument with white space in it, use quotes:

```
$ python demo_argv.py "this and" that "1 2" 3
['demo_argv.py', 'this and', 'that', '1 2', '3']
$
```

With argv we can write useful programs that take their input directly from the command line. For example, here is a program that finds the sum of a series of numbers:

```
#
# sum.py
#
from sys import argv

nums = argv[1:]

for index, value in enumerate(nums):
    nums[index] = float(value)

print sum(nums)
```

In this program we use the from <module> import <attribute> style of importing, so argv is brought into the module's main namespace.

We can now run the program from the command prompt like this:

```
$ python sum.py 3 4 5 11
23.0
$ python sum.py 3.5 5 11 100
119.5
```

You are asked to write similar programs as exercises.

10.12. Glossary

argv

argv is short for *argument vector* and is a variable in the sys module which stores a list of command line arguments passed to a program at run time.

attribute

A variable defined inside a module (or class or instance – as we will see later). Module attributes are accessed by using the **dot operator** (.).

command line

The sequence of characters read into the *command interpreter* in a *command line interface* (see the Wikipedia article on command line interface for more information).

command line argument

A value passed to a program along with the program's invocation at the *command prompt* of a command line interface (CLI).

command prompt

A string displayed by a command line interface indicating that commands can be entered.

continue statement

A statement that causes the current iteration of a loop to end. The flow of execution goes to the top of the loop, evaluates the condition, and proceeds accordingly.

delimiter

A sequence of one or more characters used to specify the boundary between separate parts of text.

directory

A named collection of files, also called a folder. Directories can contain files and other directories, which are refered to as *subdirectories* of the directory that contains them.

dot operator

The dot operator (.) permits access to attributes and functions of a module (or attributes and methods of a class or instance – as we will see later).

file

A named entity, usually stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of characters.

file system

A method for naming, accessing, and organizing files and the data they contain.

import statement

A statement which makes the objects contained in a module available for use within another module. There are two forms for the import statement. Using a hypothetical module named mymod containing functions f1 and f2, and variables v1 and v2, examples of these two forms include:

```
import mymod
```

and

```
from mymod import f1, f2, v1, v2
```

The second form brings the imported objects into the namespace of the importing module, while the first form preserves a seperate namespace for the imported module, requiring mymod.v1 to access the v1 variable.

Jython

An implementation of the Python programming language written in Java. (see the Jython home page at http://www.jython.org for more information.)

method

Function-like attribute of an object. Methods are *invoked* (called) on an object using the dot operator. For example:

```
>>> s = "this is a string."
>>> s.upper()
'THIS IS A STRING.'
>>>
```

We say that the method, upper is invoked on the string, s. s is implicitly the first argument to upper.

mode

A distinct method of operation within a computer program. Files in Python can be opened in one of three modes: read ('r'), write ('w'), and append ('a').

module

A file containing Python definitions and statements intended for use in other Python programs. The contents of a module are made available to the other program by using the import statement.

namespace

A syntactic container providing a context for names so that the same name can reside in different namespaces without ambiguity. In Python, modules, classes, functions and methods all form namespaces.

naming collision

A situation in which two or more names in a given namespace cannot be unambiguously resolved. Using

```
import string
```

instead of

```
from string import *
```

prevents naming collisions.

non-volatile memory

Memory that can maintain its state without power. Hard drives, flash drives, and rewritable compact disks (CD-RW) are each examples of non-volatile memory.

path

The name and location of a file within a file system. For example:

```
/usr/share/dict/words
```

indicates a file named words found in the dict subdirectory of the share subdirectory of the usr directory.

pydoc

A documentation generator that comes with the Python standard library.

standard library

A library is a collection of software used as tools in the development of other software. The standard library of a programming language is the set of such tools

that are distributed with the core programming language. Python comes with an extensive standard library.

text file

A file that contains printable characters organized into lines separated by newline characters.

volatilememory

Memory which requires an electrical current to maintain state. The *main memory* or RAM of a computer is volatile. Information stored in RAM is lost when the computer is turned off.

10.13. Exercises

Complete the following:

Start the pydoc server with the command pydoc -g at the command prompt.

Click on the open browser button in the pydoc tk window.

Find the calendar module and click on it.

While looking at the *Functions* section, try out the following in a Python shell:

```
>>> import calendar
>>> year = calendar.calendar(2008)
>>> print year                    # What happens here?
```

Experiment with calendar.isleap. What does it expect as an argument? What does it return as a result? What kind of a function is this?

Make detailed notes about what you learned from this exercise.

If you don't have Tkinter installed on your computer, then pydoc -g will return an error, since the graphics window that it opens requires Tkinter. An alternative is to start the web server directly:

```
$ pydoc -p 7464
```

This starts the pydoc web server on port 7464. Now point your web browser at:

```
http://localhost:7464
```

and you will be able to browse the Python libraries installed on your system. Use this approach to start pydoc and take a look at the math module.

How many functions are there in the math module?

What does math.ceil do? What about math.floor? (*hint:* both floor and ceil expect floating point arguments.)

Describe how we have been computing the same value as math.sqrt without using the math module.

What are the two data constants in the math module?

Record detailed notes of your investigation in this exercise.

Use pydoc to investigate the copy module. What does deepcopy do? In which exercises from last chapter would deepcopy have come in handy?

Create a module named mymodule1.py. Add attributes myage set to your current age, and year set to the current year. Create another module named mymodule2.py. Add attributes myage set to 0, and year set to the year you were born. Now create a file named namespace_test.py. Import both of the modules above and write the following statement:

```
print (mymodule2.myage - mymodule1.myage) == (mymodule2.year - mymodule1.year)
```

When you will run namespace_test.py you will see either True or False as output depending on whether or not you've already had your birthday this year.

Add the following statement to mymodule1.py, mymodule2.py, and namespace_test.py from the previous exercise:

```
print "My name is %s" % __name__
```

Run namespace_test.py. What happens? Why? Now add the following to the bottom of mymodule1.py:

```
if __name__ == '__main__':
```

```
print "This won't run if I'm  imported."
```

Run mymodule1.py and namespace_test.py again. In which case do you see the new print statement?

In a Python shell try the following

```
>>> import this
```

What does Tim Peters have to say about namespaces?

Use pydoc to find and test three other functions from the string module. Record your findings.

Rewrite matrix_mult from the last chapter using what you have learned about list methods.

The dir function, which we first saw in Chapter 7, prints out a list of the *attributes* of an object passed to it as an argument. In other words, dir returns the contents of the *namespace* of its argument. Use dir(str) and dir(list) to find at least three string and list methods which have not been introduced in the examples in the chapter. You should ignore anything that begins with double underscore (__) for the time being. Be sure to make detailed notes of your findings, including names of the new methods and examples of their use. (*hint:* Print the docstring of a function you want to explore. For example, to find out how str.join works, printstr.join.__doc__)

Give the Python interpreter's response to each of the following from a continuous interpreter session:

```
>>> s = "If we took the bones out, it wouldn't be crunchy, would it?"
>>> s.split()
>>> type(s.split())
>>> s.split('o')
>>> s.split('i')
>>> '0'.join(s.split('o'))
```

Be sure you understand why you get each result. Then apply what you have learned to fill in the body of the function below using the split and join methods of strobjects:

```
def myreplace(old, new, s):
```

```
    """
Replace all occurences of old with new in the string s.

>>> myreplace(',', ';', 'this, that, and, some, other, thing')
this; that; and; some; other; thing'
>>> myreplace(' ', '**', 'Words will now be separated by stars.')
'Words**will**now**be**separated**by**stars.'
  """
```

Your solution should pass all doctests.

Create a module named wordtools.py with the following at the bottom:

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Explain how this statement makes both using and testing this module convenient. What will be the value of __name__ when wordtools.py is imported from another module? What will it be when it is run as a main program? In which case will the doctests run? Now add bodies to each of the following functions to make the doctests pass:

```
def cleanword(word):
    """
>>> cleanword('what?')
'what'
     >>> cleanword('"now!"')
'now'
>>> cleanword('?+="word!,@$()"')
'word'
    """
def has_dashdash(s):
    """
>>> has_dashdash('distance--but')
     True
     >>> has_dashdash('several')
     False
     >>> has_dashdash('critters')
```

```
    False
    >>> has_dashdash('spoke--fancy')
    True
    >>> has_dashdash('yo-yo')
    False
    """
def extract_words(s):
    """
    >>> extract_words('Now is the time!  "Now", is the time? Yes, now.')
    ['now', 'is', 'the', 'time', 'now', 'is', 'the', 'time', 'yes', 'now']
    >>> extract_words('she tried to curtsey as she spoke--fancy')
    ['she', 'tried', 'to', 'curtsey', 'as', 'she', 'spoke', 'fancy']
    """
def wordcount(word, wordlist):
    """
    >>> wordcount('now', ['now', 'is', 'time', 'is', 'now', 'is', 'is'])
    ['now', 2]
    >>> wordcount('is', ['now', 'is', 'time', 'is', 'now', 'is', 'the', 'is'])
    ['is', 4]
    >>> wordcount('time', ['now', 'is', 'time', 'is', 'now', 'is', 'is'])
    ['time', 1]
    >>> wordcount('frog', ['now', 'is', 'time', 'is', 'now', 'is', 'is'])
    ['frog', 0]
    """
def wordset(wordlist):
    """
    >>> wordset(['now', 'is', 'time', 'is', 'now', 'is', 'is'])
    ['is', 'now', 'time']
    >>> wordset(['I', 'a', 'a', 'is', 'a', 'is', 'I', 'am'])
    ['I', 'a', 'am', 'is']
    >>> wordset(['or', 'a', 'am', 'is', 'are', 'be', 'but', 'am'])
    ['a', 'am', 'are', 'be', 'but', 'is', 'or']
    """
def longestword(wordset):
    """    >>> longestword(['a', 'apple', 'pear', 'grape'])
    5
    >>> longestword(['a', 'am', 'I', 'be'])
```

```
    2
    >>> longestword(['this', 'that', 'supercalifragilisticexpialidocious'])
    34
    """
```

Save this module so you can use the tools it contains in your programs.

unsorted_fruits.txt contains a list of 26 fruits, each one with a name that begins with a different letter of the alphabet. Write a program named sort_fruits.py that reads in the fruits from unsorted_fruits.txt and writes them out in alphabetical order to a file named sorted_fruits.txt.

Answer the following questions about countletters.py:

Explain in detail what the three lines do:

```
infile = open('alice_in_wonderland.txt', 'r')
text = infile.read()
infile.close()What would type(text) return after these lines have been executed?
```

What does the expression 128 * [0] evaluate to? Read about ASCII in Wikipedia and explain why you think the variable, counts is assigned to 128 * [0] in light of what you read.

What does

```
for letter in text:
    counts[ord(letter)] += 1
do to counts?
```

Explain the purpose of the display function. Why does it check for values 10, 13, and 32? What is special about those values?

Describe in detail what the lines

```
outfile = open('alice_counts.dat', 'w')
outfile.write("%-12s%s\n" % ("Character", "Count"))
outfile.write("================\n")
```

do. What will be in alice_counts.dat when they finish executing?

Finally, explain in detail what

```
for i in range(len(counts)):
    if counts[i]:
        outfile.write("%-12s%d\n" % (display(i), counts[i]))
```

does. What is the purpose of if counts[i]?

Write a program named mean.py that takes a sequence of numbers on the command line and returns the mean of their values.

```
$ python mean.py 3 4
3.5
$ python mean.py 3 4 5
4.0
```

```
$ python mean.py 11 15 94.5 22
```

```
35.625
```

A session of your program running on the same input should produce the same output as the sample session above.

Write a program named median.py that takes a sequence of numbers on the command line and returns the median of their values.

```
$ python median.py 3 7 11
7
$ python median.py 19 85 121
85
$ python median.py 11 15 16 22
```

```
15.5
```

A session of your program running on the same input should produce the same output as the sample session above.

Modify the countletters.py program so that it takes the file to open as a command line argument. How will you handle the naming of the output file?

Source: http://openbookproject.net/thinkcs/python/english2e/ch10.html