Fast Lane to Python

A quick, sensible route to the joys of Python coding

Norm Matloff
University of California, Davis

This work is licensed under a Creative Commons Attribution-No Derivative Works 3.0 United States License. Copyright is retained by N. Matloff in all non-U.S. jurisdictions, but permission to use these materials in teaching is still granted, provided the authorship and licensing information here is displayed.

The author has striven to minimize the number of errors, but no guarantee is made as to accuracy of the contents of this book.

Author's Biographical Sketch

Dr. Norm Matloff is a professor of computer science at the University of California at Davis, and was formerly a professor of statistics at that university. He is a former database software developer in Silicon Valley, and has been a statistical consultant for firms such as the Kaiser Permanente Health Plan.

Dr. Matloff was born in Los Angeles, and grew up in East Los Angeles and the San Gabriel Valley. He has a PhD in pure mathematics from UCLA, specializing in probability theory and statistics. He has published numerous papers in computer science and statistics, with current research interests in parallel processing, statistical computing, and regression methodology.

Prof. Matloff is a former appointed member of IFIP Working Group 11.3, an international committee concerned with database software security, established under UNESCO. He was a founding member of the UC Davis Department of Statistics, and participated in the formation of the UCD Computer Science Department as well. He is a recipient of the campuswide Distinguished Teaching Award and Distinguished Public Service Award at UC Davis.

Dr. Matloff is the author of two published textbooks, and of a number of widely-used Web tutorials on computer topics, such as the Linux operating system and the Python programming language. He and Dr. Peter Salzman are authors of *The Art of Debugging with GDB, DDD, and Eclipse*. Prof. Matloff's book on the R programming language, *The Art of R Programming*, was published in 2011. His book, *Parallel Computation for Data Science*, came out in 2015. His current book project, *From Linear Models to Machine Learning: Predictive Insights through R*, will be published in 2016. He is also the author of several open-source textbooks, including *From Algorithms to Z-Scores: Probabilistic and Statistical Modeling in Computer Science* (`http://heather.cs.ucdavis.edu/probstatbook`), and *Programming on Parallel Machines* (`http://heather.cs.ucdavis.edu/~matloff/ParProcBook.pdf`).

Contents

Preface

Congratulations!

Now, I'll bet you are thinking that the reason I'm congratulating you is because you've chosen to learn one of the most elegant, powerful programming languages out there. Well, that indeed calls for celebration, but the real reason I'm congratulating you is that, by virtue of actually bothering to read a book's preface (this one), you obviously belong to that very rare breed of readers—the thoughtful, discerning and creative ones!

So, here in this preface I will lay out what Python is, what I am aiming to accomplish in this book, and how to use the book.

What Are Scripting Languages?

Languages like C and C++ allow a programmer to write code at a very detailed level which has good execution speed (especially in the case of C). But in most applications, execution speed is not important—why should you care about saving 3 microseconds in your e-mail composition?—and in many cases one would prefer to write at a higher level. For example, for text-manipulation applications, the basic unit in C/C++ is a character, while for languages like Python and Perl the basic units are lines of text and words within lines. One can work with lines and words in C/C++, but one must go to greater effort to accomplish the same thing. So, using a scripting language saves you time and makes the programming experience more pleasant.

The term *scripting language* has never been formally defined, but here are the typical characteristics:

- Very casual with regard to typing of variables, e.g. little or no distinction between integer, floating-point or character string variables. Functions can return nonscalars, e.g. arrays. Nonscalars can be used as loop indexes, etc.
- Lots of high-level operations intrinsic to the language, e.g. string concatenation and stack push/pop.
- Interpreted, rather than being compiled to the instruction set of the host machine.

Why Python?

The first really popular scripting language was Perl. It is still in wide usage today, but the languages with momentum are Python and the Python-like Ruby. Many people, including me, greatly prefer Python to Perl, as it is much cleaner and more elegant. Python is very popular among the developers at Google.

Advocates of Python, often called *pythonistas*, say that Python is so clear and so enjoyable to write in that one should use Python for all of one's programming work, not just for scripting work. They believe it is superior to C or C++.[1] Personally, I believe that C++ is bloated and its pieces don't fit together well; Java is nicer, but its strongly-typed nature is in my view a nuisance and an obstacle to clear programming. I was pleased to see that Eric Raymond, the prominent promoter of the open source movement, has also expressed the same views as mine regarding C++, Java and Python.

How to Use This Tutorial

Background Needed

Anyone with even a bit of programming experience should find the material through Section 1.6 to be quite accessible.

The material beginning with Section 1.11 will feel quite comfortable to anyone with background in an object-oriented programming (OOP) language such as C++ or Java. If you lack this background, you will still be able to read these sections, but will probably need to go through them more slowly than those who do know OOP; just focus on the examples, not the terminology.

There will be a couple of places in which we describe things briefly in a Linux context, so some Linux knowledge would be helpful, but it certainly is not required. Python is used on Windows and Macintosh platforms too, not just Linux. (Most statements here made for the Linux context will also apply to Macs.)

Approach

My approach here is different from that of most Python books, or even most Python Web tutorials. The usual approach is to painfully go over all details from the beginning, with little or no context. For example, the usual approach would be to first state all possible forms that a Python integer can take on, all possible forms a Python variable name can have, and for that matter how many different ways one can launch Python with.

I avoid this here. Again, the aim is to enable the reader to quickly acquire a Python foundation. He/she

[1] Again, an exception would be programs which really need fast execution speed.

should then be able to delve directly into some special topic if and when the need arises. So, if you want to know, say, whether Python variable names can include underscores, you've come to the wrong place. If you want to quickly get into Python programming, this is hopefully the right place.

What Parts to Read, When

I would suggest that you first read through Section 1.6, and then give Python a bit of a try yourself. First experiment a bit in Python's interactive mode (Section 1.1.5). Then try writing a few short programs yourself. These can be entirely new programs, or merely modifications of the example programs presented below.[2]

This will give you a much more concrete feel of the language. If your main use of Python will be to write short scripts and you won't be using the Python library, this will probably be enough for you. However, most readers will need to go further, acquiring a basic knowledge of Python's OOP features and Python modules/packages. So you should next read through Section 1.17.

The other chapters are on special topics, such as files and directories, networks and so on.

Don't forget the chapter on debugging! Read it early and often.

My Biases

Programming is a personal, creative activity, so everyone has his/her own view. (Well, those who slavishly believe everything they were taught in programming courses are exceptions, but again, such people are not reading this preface.) Here are my biases as relates to this book:

- I don't regard global variables as evil.
- GUIs are pretty, but they REALLY require a lot of work. I'm the practical sort, and thus if a program has the required functionality in a text-based form, it's fine with me.
- I like the object-oriented paradigm to some degree, especially Python's version of it. However, I think it often gets in my way, causing me to go to a very large amount of extra work, all for little if any extra benefit. So, I use it in moderation.
- Newer is not necessarily better. Sorry, no Python 3 in this book. I have nothing against it, but I don't see its benefit either. And anyway, it's still not in wide usage.

[2] The raw **.tex** source files for this book are downloadable at `http://heather.cs.ucdavis.edu/~matloff/Python/PLN`, so you don't have to type the programs yourself. You can edit a copy of this file, saving only the lines of the program example you want.

But if you do type these examples yourself, make sure to type exactly what appears here, especially the indenting. The latter is crucial, as will be discussed later.

- Abstraction is not necessarily a sign of progress. This relates to my last two points above. I like Python because it combines power with simplicity and elegance, and thus don't put big emphasis on the fancy stuff like decorators.

Chapter 1

Introduction

So, let's get started with programming right away.

1.1 A 5-Minute Introductory Example

1.1.1 Example Program Code

Here is a simple, quick example. Suppose I wish to find the value of

$$g(x) = \frac{x}{1 - x^2}$$

for x = 0.0, 0.1, ..., 0.9. I could find these numbers by placing the following code,

```
for i in range(10):
   x = 0.1*i
   print x
   print x/(1-x*x)
```

in a file, say **fme.py**, and then running the program by typing

```
python fme.py
```

at the command-line prompt. The output will look like this:

```
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
0.5
0.666666666667
0.6
0.9375
0.7
1.37254901961
0.8
2.22222222222
0.9
4.73684210526
```

1.1.2 Python Lists

How does the program work? First, Python's **range()** function is an example of the use of **lists**, i.e. Python arrays,[1] even though not quite explicitly. Lists are absolutely fundamental to Python, so watch out in what follows for instances of the word "list"; resist the temptation to treat it as the English word "list," instead always thinking about the Python construct **list**.

Python's **range()** function returns a list of consecutive integers, in this case the list [0,1,2,3,4,5,6,7,8,9]. Note that this is official Python notation for lists—a sequence of objects (these could be all kinds of things, not necessarily numbers), separated by commas and enclosed by brackets.

1.1.3 Loops

So, the **for** statement above is equivalent to:

```
for i in [0,1,2,3,4,5,6,7,8,9]:
```

As you can guess, this will result in 10 iterations of the loop, with **i** first being 0, then 1, etc.

The code

[1] I loosely speak of them as "arrays" here, but as you will see, they are more flexible than arrays in C/C++.

On the other hand, true arrays can be accessed more quickly. In C/C++, the i^{th} element of an array **X** is i words past the beginning of the array, so we can go right to it. This is not possible with Python lists, so the latter are slower to access. The NumPy add-on package for Python offers true arrays.

```
for i in [2,3,6]:
```

would give us three iterations, with **i** taking on the values 2, 3 and 6.

Python has a **while** construct too (though not an **until**).

There is also a **break** statement like that of C/C++, used to leave loops "prematurely." For example:

```
x = 5
while 1:
   x += 1
   if x == 8:
      print x
      break
```

Also very useful is the **continue** statement, which instructs the Python interpreter to skip the remainder of the current iteration of a loop. For instance, running the code

```
sum = 0
for i in [5,12,13]:
   if i < 10: continue
   sum += i
print sum
```

prints out 12+13, i.e. 25.

The **pass** statement is a "no-op," doing nothing.

1.1.4 Python Block Definition

Now focus your attention on that innocuous-looking colon at the end of the **for** line above, which defines the start of a block. Unlike languages like C/C++ or even Perl, which use braces to define blocks, Python uses a combination of a colon and indenting to define a block. I am using the colon to say to the Python interpreter,

> Hi, Python interpreter, how are you? I just wanted to let you know, by inserting this colon, that a block begins on the next line. I've indented that line, and the two lines following it, further right than the current line, in order to tell you those three lines form a block.

I chose 3-space indenting, but the amount wouldn't matter as long as I am consistent. If for example I were to write[2]

[2] Here **g()** is a function I defined earlier, not shown.

```
for i in range(10):
   print 0.1*i
      print g(0.1*i)
```

the Python interpreter would give me an error message, telling me that I have a syntax error.[3] I am only allowed to indent further-right within a given block if I have a sub-block within that block, e.g.

```
for i in range(10):
   if i%2 == 1:
      print 0.1*i
      print g(0.1*i)
```

Here I am printing out only the cases in which the variable **i** is an odd number; % is the "mod" operator as in C/C++.

Again, note the colon at the end of the **if** line, and the fact that the two **print** lines are indented further right than the **if** line.

Note also that, again unlike C/C++/Perl, there are no semicolons at the end of Python source code statements. A new line means a new statement. If you need a very long line, you can use the backslash character for continuation, e.g.

```
x = y + \
       z
```

Most of the usual C operators are in Python, including the relational ones such as the **==** seen here. The **0x** notation for hex is there, as is the FORTRAN ****** for exponentiation.

Also, the **if** construct can be paired with **else** as usual, and you can abbreviate **else if** as **elif**.

```
>> def f(x):
...    if x > 0: return 1
...    else: return 0
...
>>> f(2)
1
>>> f(-1)
0
```

The boolean operators are **and**, **or** and **not**. In addition to the constants **True** and **False**, these are also represented numerically by nonzero and zero values, respectively. The value **None** is treated as **False**.

You'll see examples as we move along.

By the way, watch out for Python statements like **print a or b or c**, in which the first true (i.e. nonzero) expression is printed and the others ignored; this is a common Python idiom.

[3] Keep this in mind. New Python users are often baffled by a syntax error arising in this situation.

1.1.5 Python Also Offers an Interactive Mode

A really nice feature of Python is its ability to run in interactive mode. You usually won't do this, but it's a great way to do a quick tryout of some feature, to really see how it works. Whenever you're not sure whether something works, your motto should be, "When in doubt, try it out!", and interactive mode makes this quick and easy.

We'll also be doing a lot of that in this tutorial, with interactive mode being an easy way to do a quick illustration of a feature.

Instead of executing this program from the command line in **batch** mode as we did above, we could enter and run the code in **interactive** mode:

```
% python
>>> for i in range(10):
...    x = 0.1*i
...    print x
...    print x/(1-x*x)
...
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
0.5
0.666666666667
0.6
0.9375
0.7
1.37254901961
0.8
2.22222222222
0.9
4.73684210526
>>>
```

Here I started Python, and it gave me its >>> interactive prompt. Then I just started typing in the code, line by line. Whenever I was inside a block, it gave me a special prompt, "...", for that purpose. When I typed a blank line at the end of my code, the Python interpreter realized I was done, and ran the code.[4]

[4] Interactive mode allows us to execute only single Python statements or evaluate single Python expressions. In our case here, we typed in and executed a single **for** statement. Interactive mode is not designed for us to type in an entire program. Technically we could work around this by beginning with something like ”if 1:”, making our program one large **if** statement, but of course it would not be convenient to type in a long program anyway.

While in interactive mode, one can go up and down the command history by using the arrow keys, thus saving typing.

To exit interactive Python, hit ctrl-d.

Automatic printing: By the way, in interactive mode, just referencing or producing an object, or even an expression, without assigning it, will cause its value to print out, even without a **print** statement. For example:

```
>>> for i in range(4):
...    3*i
...
0
3
6
9
```

Again, this is true for general objects, not just expressions, e.g.:

```
>>> open('x')
<open file 'x', mode 'r' at 0xb7eaf3c8>
```

Here we opened the file **x**, which produces a file object. Since we did not assign to a variable, say **f**, for reference later in the code, i.e. we did not do the more typical

```
f = open('x')
```

the object was printed out. We'd get that same information this way:

```
>>> f = open('x')
>>> f
<open file 'x', mode 'r' at 0xb7f2a3c8>
```

1.1.6 Python As a Calculator

Among other things, this means you can use Python as a quick calculator (which I do a lot). If for example I needed to know what 5% above $88.88 is, I could type

```
% python
>>> 1.05*88.88
93.323999999999998
```

Among other things, one can do quick conversions between decimal and hex:

```
>>> 0x12
18
>>> hex(18)
'0x12'
```

One can also look up ASCII codes:

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

If I need math functions, I must **import** the Python math library first. This is analogous to what we do in C/C++, where we must have a **#include** line for the library in our source code and must link in the machine code for the library.

We must refer to imported functions in the context of the library, in this case the math library. For example, the functions **sqrt()** and **sin()** must be prefixed by **math**:[5]

```
>>> import math
>>> math.sqrt(88)
9.3808315196468595
>>> math.sin(2.5)
0.59847214410395655
```

1.2 A 10-Minute Introductory Example

1.2.1 Example Program Code

This program reads a text file, specified on the command line, and prints out the number of lines and words in the file:

```
# reads in the text file whose name is specified on the command line,
# and reports the number of lines and words

import sys

def checkline():
   global l
   global wordcount
   w = l.split()
   wordcount += len(w)

wordcount = 0
f = open(sys.argv[1])
flines = f.readlines()
```

[5] A method for avoiding the prefix is shown in Sec. 1.15.2.

```
linecount = len(flines)
for l in flines:
   checkline()
print linecount, wordcount
```

Say for example the program is in the file **tme.py**, and we have a text file **x** with contents

```

This is an
example of a
text file.

```

(There are five lines in all, the first and last of which are blank.)

If we run this program on this file, the result is:

```
python tme.py x
5 8
```

On the surface, the layout of the code here looks like that of a C/C++ program: First an **import** statement, analogous to **#include** (with the corresponding linking at compile time) as stated above; second the definition of a function; and then the "main" program. This is basically a good way to look at it, but keep in mind that the Python interpreter will execute everything in order, starting at the top. In executing the **import** statement, for instance, that might actually result in some code being executed, if the module being imported has some free-standing code rather than just function definitions. More on this later. Execution of the **def** statement won't execute any code for now, but the act of defining the function is considered execution.

Here are some features in this program which were not in the first example:

- use of command-line arguments
- file-manipulation mechanisms
- more on lists
- function definition
- library importation
- introduction to scope

I will discuss these features in the next few sections.

1.2.2 Command-Line Arguments

First, let's explain **sys.argv**. Python includes a **module** (i.e. library) named **sys**, one of whose member variables is **argv**. The latter is a Python list, analogous to **argv** in C/C++.[6] Element 0 of the list is the script name, in this case **tme.py**, and so on, just as in C/C++. In our example here, in which we run our program on the file **x**, **sys.argv[1]** will be the string 'x' (strings in Python are generally specified with single quote marks). Since **sys** is not loaded automatically, we needed the **import** line.

Both in C/C++ and Python, those command-line arguments are of course strings. If those strings are supposed to represent numbers, we could convert them. If we had, say, an integer argument, in C/C++ we would do the conversion using **atoi()**; in Python, we'd use **int()**. For floating-point, in Python we'd use **float()**.[7]

1.2.3 Introduction to File Manipulation

The function **open()** is similar to the one in C/C++. Our line

```
f = open(sys.argv[1])
```

created an object of **file** class, and assigned it to **f** .

The **readlines()** function of the **file** class returns a list (keep in mind, "list" is an official Python term) consisting of the lines in the file. Each line is a string, and that string is one element of the list. Since the file here consisted of five lines, the value returned by calling **readlines()** is the five-element list

```
['','This is an','example of a','text file','']
```

(Though not visible here, there is an end-of-line character in each string.)

1.2.4 Lack of Declaration

Variables are not declared in Python. A variable is created when the first assignment to it is executed. For example, in the program **tme.py** above, the variable **flines** does not exist until the statement

```
flines = f.readlines()
```

[6] There is no need for an analog of **argc**, though. Python, being an object-oriented language, treats lists as objects, The length of a list is thus incorporated into that object. So, if we need to know the number of elements in **argv**, we can get it via **len(argv)**.

[7] In C/C++, we could use **atof()** if it were available, or **sscanf()**.

is executed.

By the way, a variable which has not been assigned a value yet, such as **wordcount** at first above, has the value **None**. And this can be assigned to a variable, tested for in an **if** statement, etc.

1.2.5 Locals Vs. Globals

Python does not really have global variables in the sense of C/C++, in which the scope of a variable is an entire program. We will discuss this further in Section 1.15.6, but for now assume our source code consists of just a single **.py** file; in that case, Python does have global variables pretty much like in C/C++ (though with important differences).

Python tries to infer the scope of a variable from its position in the code. If a function includes any code which assigns to a variable, then that variable is assumed to be local, unless we use the **global** keyword. So, in the code for **checkline**(), Python would assume that **l** and **wordcount** are local to **checkline()** if we had not specified **global**.

Use of global variables simplifies the presentation here, and I personally believe that the unctuous criticism of global variables is unwarranted. (See `http://heather.cs.ucdavis.edu/~matloff/globals.html`.) In fact, in one of the major types of programming, **threads**, use of globals is basically *mandatory*.

You may wish, however, to at least group together all your globals into a class, as I do. See Appendix 1.25.

1.2.6 A Couple of Built-In Functions

The function **len**() returns the number of elements in a list. In the **tme.py** example above, we used this to find the number of lines in the file, since **readlines()** returned a list in which each element consisted of one line of the file.

The method **split()** is a member of the **string** class.[8] It splits a string into a list of words, for example.[9] So, for instance, in **checkline()** when **l** is ’This is an’ then the list **w** will be equal to [’This’,’is’,’an’]. (In the case of the first line, which is blank, **w** will be equal to the empty list, [].)

1.3 Types of Variables/Values

As is typical in scripting languages, type in the sense of C/C++ **int** or **float** is not declared in Python. However, the Python interpreter does internally keep track of the type of all objects. Thus Python variables

[8] Member functions of classes are referred to via **methods**.

[9] The default is to use blank characters as the splitting criterion, but other characters or strings can be used.

don't have types, but their values do. In other words, a variable **X** might be bound to (i.e. point to) an integer in one place in your program and then be rebound to a class instance at another point.

Python's types include notions of scalars, **sequences** (lists or **tuples**) and dictionaries (associative arrays, discussed in Sec. 1.7), classes, function, etc.

1.4 String Versus Numerical Values

Unlike Perl, Python does distinguish between numbers and their string representations. The functions **eval()** and **str()** can be used to convert back and forth. For example:

```
>>> 2 + '1.5'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> 2 + eval('1.5')
3.5
>>> str(2 + eval('1.5'))
'3.5'
```

There are also **int()** to convert from strings to integers, and **float()**, to convert from strings to floating-point values:

```
>>> n = int('32')
>>> n
32
>>> x = float('5.28')
>>> x
5.2800000000000002
```

See also Section 1.5.3.2.

1.5 Sequences

Lists are actually special cases of **sequences**, which are all array-like but with some differences. Note though, the commonalities; all of the following (some to be explained below) apply to any sequence type:

- the use of brackets to denote individual elements (e.g. **x[i]**)
- the built-in **len()** function to give the number of elements in the sequence[10]

[10] This function is applicable to dictionaries too.

- **slicing** operations, i.e. the extraction of subsequences
- use of **+** and ***** operators for concatenation and replication

1.5.1 Lists (Quasi-Arrays)

As stated earlier, lists are denoted by brackets and commas. For instance, the statement

```
x = [4,5,12]
```

would set **x** to the specified 3-element array.

A number of other operations are available for lists, a few of which are illustrated in the following code:

```
>>> x = [5,12,13,200]
>>> x
[5, 12, 13, 200]
>>> x.append(-2)
>>> x
[5, 12, 13, 200, -2]
>>> del x[2]
>>> x
[5, 12, 200, -2]
>>> z = x[1:3]  # array "slicing": elements 1 through 3-1 = 2
>>> z
[12, 200]
>>> yy = [3,4,5,12,13]
>>> yy[3:]  # all elements starting with index 3
[12, 13]
>>> yy[:3]  # all elements up to but excluding index 3
[3, 4, 5]
>>> yy[-1]  # means "1 item from the right end"
13
>>> x.insert(2,28)  # insert 28 at position 2
>>> x
[5, 12, 28, 200, -2]
>>> 28 in x  # tests for membership; 1 for true, 0 for false
1
>>> 13 in x
0
>>> x.index(28)  # finds the index within the list of the given value
2
>>> x.remove(200)  # different from "delete," since it’s indexed by value
>>> x
[5, 12, 28, -2]
>>> w = x + [1,"ghi"]  # concatenation of two or more lists
>>> w
[5, 12, 28, -2, 1, ’ghi’]
>>> qz = 3*[1,2,3]  # list replication
>>> qz
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> x = [1,2,3]
>>> x.extend([4,5])
>>> x
[1, 2, 3, 4, 5]
>>> g = [1,2,3]
>>> g.append([4,5])
>>> g
[1, 2, 3, [4, 5]]
>>> y = x.pop(0)  # deletes and returns 0th element
>>> y
1
>>> x
[2, 3, 4, 5]
>>> t = [5,12,13]
>>> t.reverse()
>>> t
[13, 12, 5]
```

We also saw the **in** operator in an earlier example, used in a **for** loop.

A list could include mixed elements of different types, including other lists themselves.

Lists may grow dynamically, using the **list** class' **append()** or **extend()** functions. For example, if after the above statement we were to execute

```
x.append(-2)
```

x would now be equal to [4,5,12,-2].

The difference between the two is that **append()** adds one element to a list, while **extend** does concatenation of lists. This can be subtle:

```
>>> y = [1,2,3]
>>> y.extend([8,88])
>>> y
[1, 2, 3, 8, 88]
>>> y.append([8,88])
>>> y
[1, 2, 3, 8, 88, [8, 88]]
```

The Python idiom includes a number of common "Python tricks" involving sequences, e.g. the following quick, elegant way to swap two variables **x** and **y**:

```
>>> x = 5
>>> y = 12
>>> [x,y] = [y,x]
>>> x
12
>>> y
5
```

Note that the elements of the list may be mixed types, e.g.

```
>>> z = [1,'a']
>>> z
[1, 'a']
```

Multidimensional lists can be implemented as lists of lists. For example:

```
>>> x = []
>>> x.append([1,2])
>>> x
[[1, 2]]
>>> x.append([3,4])
>>> x
[[1, 2], [3, 4]]
>>> x[1][1]
4
```

But be careful! Look what can go wrong:

```
>>> x = 4*[0]
>>> y = 4*[x]
>>> y
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> y[0][2]
0
>>> y[0][2] = 1
>>> y
[[0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0]]
```

The problem is that that assignment to **y** was really a list of four references to the same thing (**x**). When the object pointed to by **x** changed, then all four rows of **y** changed.

The Python Wikibook (`http://en.wikibooks.org/wiki/Python_Programming/Lists`) suggests a solution, in the form of **list comprehensions**, which we cover in Section 3.5:

```
>>> z = [[0]*4 for i in range(5)]
>>> z
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> z[0][2] = 1
>>> z
[[0, 0, 1, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

1.5.2 Tuples

Tuples are like lists, but are **immutable**, i.e. unchangeable. They are enclosed by parentheses or nothing at all, rather than brackets. The parentheses are mandatory if there is an ambiguity without them, e.g. in function arguments. A comma must be used in the case of empty or single tuple, e.g. **(,)** and **(5,)**.

Since a tuple can’t grow or shrink, the Python interpreter can store it in a contiguous block of memory, thus create efficient code.

The same operations can be used, except those which would change the tuple. So for example

```
x = (1,2,’abc’)
print x[1]  # prints 2
print len(x)  # prints 3
x.pop()  # illegal, due to immutability
```

A nice function is **zip()**, which strings together corresponding components of several lists, producing tuples, e.g.

```
>>> zip([1,2],[’a’,’b’],[168,168])
[(1, ’a’, 168), (2, ’b’, 168)]
```

1.5.3 Strings

Strings are essentially tuples of character elements. But they are quoted instead of surrounded by parentheses, and have more flexibility than tuples of character elements would have.

Some subtleties can occur, e.g.

```
>>> z = [1,2,3]
>>> z.extend(’abc’)
>>> z
[1, 2, 3, ’a’, ’b’, ’c’]
```

The fact that ’abc’ is considered a 3-element tuple, rather than a 1-element scalar, came into play here.

1.5.3.1 Strings As Turbocharged Tuples

Let’s see some examples of string operations:

```
>>> x = ’abcde’
>>> x[2]
’c’
>>> x[2] = ’q’  # illegal, since strings are immmutable
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn’t support item assignment
>>> x = x[0:2] + ’q’ + x[3:5]
>>> x
’abqde’
```

(You may wonder why that last assignment

```
>>> x = x[0:2] + 'q' + x[3:5]
```

does not violate immmutability. The reason is that **x** is really a pointer, and we are simply pointing it to a new string created from old ones. See Section 1.12.)

As noted, strings are more than simply tuples of characters:

```
>>> x.index('d')  # as expected
3
>>> 'd' in x  # as expected
1
>>> x.index('de')  # pleasant surprise
3
```

As can be seen, the **index()** function from the **str** class has been overloaded, making it more flexible.

Thus a list of character strings is a two-dimensional array:

```
>>> x = ['abc','de','f']
>>> x[1]
'de'
>>> x[1][1]
'e'
```

There are many other handy functions in the **str** class. For example, we saw the **split()** function earlier. The opposite of this function is **join()**. One applies it to a string, with a sequence of strings as an argument. The result is the concatenation of the strings in the sequence, with the original string between each of them:[11]

```
>>> '---'.join(['abc','de','xyz'])
'abc---de---xyz'
>>> q = '\n'.join(('abc','de','xyz'))
>>> q
'abc\nde\nxyz'
>>> print q
abc
de
xyz
```

Here are some more:

[11] The example here shows the "new" usage of **join()**, now that string methods are built-in to Python. See discussion of "new" versus "old" below.

```
>>> x = 'abc'
>>> x.upper()
'ABC'
>>> 'abc'.upper()
'ABC'
>>> 'abc'.center(5) # center the string within a 5-character set
' abc '
>>> 'abc de f'.replace(' ','+')
'abc+de+f'
>>> x = 'abc123'
>>> x.find('c1')  # find index of first occurrence of 'c1' in x
2
>>> x.find('3')
5
>>> x.find('1a')
-1
```

A very rich set of functions for string manipulation is also available in the **re** ("regular expression") module.

The **str** class is built-in for newer versions of Python. With an older version, you will need a statement

```
import string
```

That latter class does still exist, and the newer **str** class does not quite duplicate it.

1.5.3.2 Formatted String Manipulation

String manipulation is useful in lots of settings, one of which is in conjunction with Python's **print** command. For example,

```
print "the factors of 15 are %d and %d" % (3,5)
```

prints out

```
the factors of 15 are 3 and 5
```

The **%d** of course is the integer format familiar from C/C++.

But actually, the above action is a string issue, not a print issue. Let's see why. In

```
print "the factors of 15 are %d and %d" % (3,5)
```

the portion

```
"the factors of 15 are %d and %d" % (3,5)
```

is a string operation, producing a new string; the **print** simply prints that new string.

For example:

```
>>> x = "%d years old" % 12
```

The variable **x** now is the string ’12 years old’.

This is another very common idiom, quite powerful.[12]

Note the importance above of writing ’(3,5)’ rather than ’3,5’. In the latter case, the **%** operator would think that its operand was merely 3, whereas it needs a 2-element tuple. Recall that parentheses enclosing a tuple can be omitted as long as there is no ambiguity, but that is not the case here.

1.5.4 Sorting Sequences

The Python function **sort()** can be applied to any sequence. For nonscalars, one provides a “compare” function, which returns a negative, zero or positive value, signfiying $<$, $=$ or $>$. As an illustration, let’s sort an array of arrays, using the second elements as keys:

```
>>> x = [[1,4],[5,2]]
>>> x
[[1, 4], [5, 2]]
>>> x.sort()
>>> x
[[1, 4], [5, 2]]
>>> def g(u,v):
...    return u[1]-v[1]
...
>>> x.sort(g)
>>> x
[[5, 2], [1, 4]]
```

(This would be more easily done using “lambda” functions. See Section 3.1.)

There is a Python library module, **bisect**, which does binary search and related sorting.

1.6 Determining Object Types

You can use **type()** to determine object type:

[12] Some C/C++ programmers might recognize the similarity to **sprintf()** from the C library.

```
>>> x = [1,2,3]
>>> type(x)
<type ’list’>
>>> type(x) is list
True
```

1.7 Dictionaries (Hashes)

Dictionaries are **associative arrays**. The technical meaning of this will be discussed below, but from a pure programming point of view, this means that one can set up arrays with non-integer indices. The statement

```
x = {’abc’:12,’sailing’:’away’}
```

sets **x** to what amounts to a 2-element array with **x[’abc’]** being 12 and **x[’sailing’]** equal to ’away’. We say that **’abc’** and **’sailing’** are **keys**, and 12 and ’away’ are **values**. Keys can be any immmutable object, i.e. numbers, tuples or strings.[13] Use of tuples as keys is quite common in Python applications, and you should keep in mind that this valuable tool is available.

Internally, **x** here would be stored as a 4-element array, and the execution of a statement like

```
w = x[’sailing’]
```

would require the Python interpreter to search through that array for the key ’sailing’. A linear search would be slow, so internal storage is organized as a hash table. This is why Perl’s analog of Python’s dictionary concept is actually called a **hash**.

Here are examples of usage of some of the member functions of the **dictionary** class:

```
>>> x = {’abc’:12,’sailing’:’away’}
>>> x[’abc’]
12
>>> y = x.keys()
>>> y
[’abc’, ’sailing’]
>>> z = x.values()
>>> z
[12, ’away’]
x[’uv’] = 2
>>> x
{’abc’: 12, ’uv’: 2, ’sailing’: ’away’}
```

[13] Here we see another reason why Python distinguishes between tuples and lists. Allowing mutable keys would be an implementation nightmare, and probably lead to error-prone programming.

Note how we added a new element to **x**.

The keys need not be tuples. For example:

```
>>> x
{'abc': 12, 'uv': 2, 'sailing': 'away'}
>>> f = open('z')
>>> x[f] = 88
>>> x
{<open file 'z', mode 'r' at 0xb7e6f338>: 88, 'abc': 12, 'uv': 2, 'sailing': 'away'}
```

Deletion of an element from a dictionary can be done via **pop()**, e.g.

```
>>> x.pop('abc')
12
>>> x
{<open file 'x', mode 'r' at 0xb7e6f338>: 88, 'uv': 2, 'sailing': 'away'}
```

The **in** operator works on dictionary keys, e.g.

```
>>> x = {'abc': 12, 'uv': 2, 'sailing': 'away'}
>>> 'uv' in x
True
>>> 2 in x
False
```

1.8 Function Definition

Obviously the keyword **def** is used to define a function. Note once again that the colon and indenting are used to define a block which serves as the function body. A function can return a value, using the **return** keyword, e.g.

```
return 8888
```

However, the function does not have a type even if it does return something, and the object returned could be anything—an integer, a list, or whatever.

Functions are **first-class objects**, i.e. can be assigned just like variables. Function names *are* variables; we just temporarily assign a set of code to a name. Consider:

```
>>> def square(x):  # define code, and point the variable square to it
...    return x*x
...
```

```
>>> square(3)
9
>>> gy = square  # now gy points to that code too
>>> gy(3)
9
>>> def cube(x):
...    return x**3
...
>>> cube(3)
27
>>> square = cube  # point the variable square to the cubing code
>>> square(3)
27
>>> square = 8.8
>>> square
8.8000000000000007  # don't be shocked by the 7
>>> gy(3)  # gy still points to the squaring code
9
```

1.9 Use of __name__

In some cases, it is important to know whether a module is being executed on its own, or via **import**. This can be determined through Python's built-in variable __**name**__, as follows.

Whatever the Python interpreter is running is called the **top-level program**. If for instance you type

```
% python x.py
```

then the code in **x.py** is the top-level program. If you are running Python interactively, then the code you type in is the top-level program.

The top-level program is known to the interpreter as __**main**__, and the module currently being run is referred to as __**name**__. So, to test whether a given module is running on its own, versus having been imported by other code, we check whether __**name**__ is __**main**__. If the answer is yes, you are in the top level, and your code was not imported; otherwise it was.

For example, let's add a statement

```
print __name__
```

to our very first code example, from Section 1.1.1, in the file **fme.py**:

```
print __name__
for i in range(10):
   x = 0.1*i
   print x
   print x/(1-x*x)
```

Let’s run the program twice. First, we run it on its own:

```
% python fme.py
__main__
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
... [remainder of output not shown]
```

Now look what happens if we run it from within Python’s interactive interpreter:

```
>>> import fme
fme
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
... [remainder of output not shown]
```

Our module’s statement

```
print __name__
```

printed out **__main__** the first time, but printed out **fme** the second time. Here’s what happened: In the first run, the Python interpreter was running **fme.py**, while in the second one it was running **import fme**. The latter of course resulting in the **fme.py** code running, but that code was now second-level.

It is customary to collect one’s “main program” (in the C sense) into a function, typically named **main()**. So, let’s change our example above to **fme2.py**:

```
def main():
   for i in range(10):
      x = 0.1*i
      print x
      print x/(1-x*x)

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

The advantage of this is that when we import this module, the code won't be executed right away. Instead, **fme2.main()** must be called, either by the importing module or by the interactive Python interpreter. Here is an example of the latter:

```
>>> import fme2
>>> fme2.main()
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
...
```

One of the uses of this involves executable code in imported modules. Say you have a module includes a statement

```
z = 2
```

that is freestanding, i.e. NOT part of some function. You may want that statement to execute if the module is run directly, but not if it is imported. The above test using __**name**__ would enable you to distinguish between the two cases.

Among other things, this will be a vital point in using debugging tools (Section 8). So get in the habit of always setting up access to **main()** in this manner in your programs.

1.10 Extended Example: Computing Final Grades

```
# computes and records final grades

# input line format:

#    name and misc. info, e.g. class level
#    Final Report grade
#    Midterm grade
#    Quiz grades
#    Homework grades

# e.g.
#
#    John Paul Jones SR LCSI 70 A 50 B 69 B+ B+ A 3.52 B+

# comment lines, beginning with #, are ignored for computation but are
# printed out; thus various notes can be put in comment lines; e.g.
# notes on missed or makeup exams

# usage:

#   python FinalGrades.py input_file nq nqd nh wts

#      where there are nq Quizzes, the lowest nqd of which will be
#      deleted; nh Homework assignments; and wts is the set of weights
#      for Final Report, Midterm, Quizzes and Homework

# outputs to stdout the input file with final course grades appended;
# the latter are numerical only, allowing for personal inspection of
# "close" cases, etc.

import sys

def convertltr(lg):  # converts letter grade lg to 4-point-scale
   if lg == ’F’: return 0
   base = lg[0]
   olg = ord(base)
   if len(lg) > 2 or olg < ord(’A’) or olg > ord(’D’):
      print lg, ’is not a letter grade’
      sys.exit(1)
   grade = 4 - (olg-ord(’A’))
   if len(lg) == 2:
      if lg[1] == ’+’: grade += 0.3
      elif lg[1] == ’-’: grade -= 0.3
      else:
         print lg, ’is not a letter grade’
         sys.exit(1)
   return grade

def avg(x,ndrop):
   tmp = []
   for xi in x: tmp.append(convertltr(xi))
   tmp.sort()
   tmp = tmp[ndrop:]
   return float(sum(tmp))/len(tmp)

def main():
   infile = open(sys.argv[1])
   nq = int(sys.argv[2])
   nqd = int(sys.argv[3])
   nh = int(sys.argv[4])
   wts = []
   for i in range(4): wts.append(float(sys.argv[5+i]))
   for line in infile.readlines():
      toks = line.split()
      if toks[0] != ’#’:
         lw = len(toks)
         startpos = lw - nq - nh - 3
         # Final Report
         frgrade = convertltr(toks[startpos])
         # Midterm letter grade (skip over numerical grade)
         mtgrade = convertltr(toks[startpos+2])
         startquizzes = startpos + 3
         qgrade = avg(toks[startquizzes:startquizzes+nq],nqd)
         starthomework = startquizzes + nq
         hgrade = avg(toks[starthomework:starthomework+nh],0)
         coursegrade = 0.0
         coursegrade += wts[0] * frgrade
         coursegrade += wts[1] * mtgrade
         coursegrade += wts[2] * qgrade
         coursegrade += wts[3] * hgrade
         print line[:len(line)-1], coursegrade
      else:
         print line[:len(line)-1]

if __name__ == ’__main__’:
   main()
```

1.11 Object-Oriented Programming

In contrast to Perl, Python has been object-oriented from the beginning, and thus has a much nicer, cleaner, clearer interface for OOP.

1.11.1 Example: Text File

As an illustration, we will develop a class which deals with text files. Here are the contents of the file **tfe.py**:

```
class textfile:
   ntfiles = 0  # count of number of textfile objects
   def __init__(self,fname):
      textfile.ntfiles += 1
      self.name = fname  # name
      self.fh = open(fname)  # handle for the file
      self.lines = self.fh.readlines()
      self.nlines = len(self.lines)  # number of lines
      self.nwords = 0  # number of words
      self.wordcount()
   def wordcount(self):
      "finds the number of words in the file"
      for l in self.lines:
         w = l.split()
         self.nwords += len(w)
   def grep(self,target):
      "prints out all lines containing target"
      for l in self.lines:
         if l.find(target) >= 0:
            print l

a = textfile(’x’)
b = textfile(’y’)
print "the number of text files open is", textfile.ntfiles
print "here is some information about them (name, lines, words):"
for f in [a,b]:
   print f.name,f.nlines,f.nwords
a.grep(’example’)
```

(By the way, note the **docstrings**, the double-quoted, comment-like lines in **wordcount()** and **grep()**. These are “supercomments,” explained in Section 1.18.)

In addition to the file **x** I used in Section 1.2 above, I had the 2-line file **y**. Here is what happened when I ran the program:

```
% python tfe.py
the number of text files opened is 2
here is some information about them (name, lines, words):
x 5 8
y 2 5
example of a
```

1.11.2 The Objects

In this code, we created two **objects**, which we named **a** and **b**. Both were **instances** of the class **textfile**.

1.11.3 Constructors and Destructors

Technically, an instance of a class is created at the time the class name is invoked as a function, as we did above in the line

```
a = textfile(’x’)
```

So, one might say that the class name, called in functional form, is the constructor. However, we’ll think of the constructor as being the **__init()__** method. It is a built-in method in any class, but is typically overridden by our own definition, as we did above. (See Section 1.25 for an example in which we do not override **__init()__**.)

The first argument of **__init()__** is mandatory, and almost every Python programmer chooses to name it **self**, which C++/Java programmers will recognize as the analog of **this** in those languages.

Actually **self** is not a keyword. Unlike the **this** keyword in C++/Java, you do not HAVE TO call this variable **self**. Whatever you place in that first argument of **__init()__** will be used by Python’s interpreter as a pointer to the current instance of the class. If in your definition of **__init()__** you were to name the first argument **me**, and then write “me” instead of “self” throughout the definition of the class, that would work fine. However, you would invoke the wrath of purist pythonistas all over the world. So don’t do it.

Often **__init()__** will have additional arguments, as in this case with a filename.

The destructor is **__del()__**. Note that it is only invoked when garbage collection is done, i.e. when all variables pointing to the object are gone.

1.11.4 Instance Variables

In general OOP terminology, an **instance variable** of a class is a member variable for which each instance of the class has a separate value of that variable. In the example above, the instance variable **fname** has the value ’x’ in object **a**, but that same variable has the value ’y’ in object **b**.

In the C++ or Java world, you know this as a variable which is not declared **static**. The term *instance variable* is the generic OOP term, non-language specific.

1.11.5 Class Variables

A **class variable** is one that is associated with the class itself, not with instances of the class. Again in the C++ or Java world, you know this as a **static** variable. It is designated as such by having some reference to **v** in code which is in the class but not in any method of the class. An example is the code

```
ntfiles = 0  # count of number of textfile objects
```

above.[14]

Note that a class variable **v** of a class **u** is referred to as **u.v** within methods of the class and in code outside the class. For code inside the class but not within a method, it is referred to as simply **v**. Take a moment now to go through our example program above, and see examples of this with our **ntfiles** variable.

1.11.6 Instance Methods

The method **wordcount()** is an **instance method**, i.e. it applies specifically to the given object of this class. Again, in C++/Java terminology, this is a non-**static** method. Unlike C++ and Java, where **this** is an implicit argument to instance methods, Python wisely makes the relation explicit; the argument **self** is required.

The method **grep()** is another instance method, this one with an argument besides **self**.

Note also that **grep()** makes use of one of Python's many string operations, **find()**. It searches for the argument string within the object string, returning the index of the first occurrence of the argument string within the object string, or returning -1 if none is found.[15]

1.11.7 Class Methods

A **class method** is associated with the class itself. It does not have **self** as an argument.

Python has two (slightly differing) ways to designate a function as a class method, via the functions **staticmethod()** and **classmethod()**. We will use only the former.[16]

As our first example, consider following enhancement to the code in within the class **textfile** above:

```
class textfile:
   ...
   def totfiles():
      print "the total number of text files is", textfile.ntfiles
   totfiles = staticmethod(totfiles)

...

# here we are in "main"
```

[14] By the way, though we placed that code at the beginning of the class, it could be at the end of the class, or between two methods, as long as it is not inside a method. In the latter situation **ntfiles** would be considered a local variable in the method, not what we want at all.

[15] Strings are also treatable as lists of characters. For example, 'geometry' can be treated as an 8-element list, and applying **find()** for the substring 'met' would return 3.

[16] See also Section 1.23.

```
...
textfile.totfiles()
...
```

Note that **staticmethod()** is indeed a function, as the above syntax would imply. It takes one function as input, and outputs another function.

In newer versions of Python, one can also designate a function as a class method this way:

```
class textfile:
   ...
   @staticmethod
   def totfiles():
      print "the total number of text files is", textfile.ntfiles
```

A class method can be called even if there are not yet any instances of the class, say **textfile** in this example. Here, 0 would be printed out, since no files had yet been counted.

Note carefully that this is different from the Python value **None**. Even if we have not yet created instances of the class **textfile**, the code

```
ntfiles = 0
```

would still have been executed when we first started execution of the program. As mentioned earlier, the Python interpreter executes the file from the first line onward. When it reaches the line

```
class textfile:
```

it then executes any free-standing code in the definition of the class.

1.11.8 Derived Classes

Inheritance is very much a part of the Python philosophy. A statement like

```
class b(a):
```

starts the definition of a subclass **b** of a class **a**. Multiple inheritance, etc. can also be done.

Note that when the constructor for a derived class is called, the constructor for the base class is not automatically called. If you wish the latter constructor to be invoked, you must invoke it yourself, e.g.

```
class b(a):
   def __init__(self,xinit):  # constructor for class b
      self.x = xinit  # define and initialize an instance variable x
      a.__init__(self)  # call base class constructor
```

1.11.9 Extended Example: Vending Machine Collection

Here we will deal with a class representing a vending machine. Each object of this class represents one machine, but all the machines carry the same items (though the current size of the stock of a given item may vary from machine to machine).

The **inventory** variable will be a dictionary with keys being item names and values being the current stocks of those items, e.g. 'Kit Kat':8 signifying that this machine currently holds a stock of 8 Kit Kat bars.

The method **newstock()** adds to the stocks of the given items; e.g. m.newstock({'Kit Kat':3,' Sun Chips':2) would record that the stocks of Kit Kat bars and bags of Sun Chips at machine **m** have been replenished by 3 bars and 2 bags, respectively.

```
class machine:
   itemnames = []
   def __init__(self):
      # in (itemname, stock) form
      self.inventory = {}
      for nm in machine.itemnames:
         self.inventory[nm] = 0
   # adds the new stock to inventory; items is in dictionary form,
   # (itemname, newstock form)
   def newstock(self,newitems):
      for itm in newitems.keys():
         self.inventory[itm] += newitems[itm]
```

Of course, many more methods could be added.

1.11.10 A Word on Class Implementation

A Python class instance is implemented internally as a dictionary. For example, in our program **tfe.py** above, the object **b** is implemented as a dictionary.

Among other things, this means that you can add member variables to an instance of a class "on the fly," long after the instance is created. We are simply adding another key and value to the dictionary. In our "main" program, for example, we could have a statement like

```
b.name = 'zzz'
```

1.12 Importance of Understanding Object References

A variable which has been assigned a mutable value is actually a pointer to the given object. For example, consider this code:

```
>>> x = [1,2,3]  # x is mutable
>>> y = x  # x and y now both point to [1,2,3]
>>> x[2] = 5  # the mutable object pointed to by x now "mutes"
>>> y[2]  # this means y[2] changes to 5 too!
5
>>> x = [1,2]
>>> y = x
>>> y
[1, 2]
>>> x = [3,4]
>>> y
[1, 2]
```

In the first few lines, **x** and **y** are references to a list, a mutable object. The statement

```
x[2] = 5
```

then changes one aspect of that object, but **x** still points to that object. On the other hand, the code

```
x = [3,4]
```

now changes **x** itself, having it point to a different object, while **y** is still pointing to the first object.

If in the above example we wished to simply copy the list referenced by **x** to **y**, we could use slicing, e.g.

```
y = x[:]
```

Then **y** and **x** would point to different objects; **x** would point to the same object as before, but the statement for **y** would create a new object, which **y** would point to. Even though those two objects have the same values for the time being, if the object pointed to by **x** changes, **y**'s object won't change.

As you can imagine, this gets delicate when we have complex objects. See Python's **copy** module for functions that will do object copying to various depths.

An important similar issue arises with arguments in function calls. Any argument which is a variable which points to a mutable object can change the value of that object from within the function, e.g.:

```
>>> def f(a):
...    a = 2*a  # numbers are immutable
```

```
...
>>> x = 5
>>> f(x)
>>> x  # x doesn't change
5
>>> def g(a):
...    a[0] = 2*a[0]  # lists are mutable
...
>>> y = [5]
>>> g(y)
>>> y  # y changes!
[10]
```

Function names are references to objects too. What we think of as the name of the function is actually just a pointer—a mutable one—to the code for that function. For example,

```
>>> def f():
...    print 1
...
>>> def g():
...    print 2
...
>>> f()
1
>>> g()
2
>>> [f,g] = [g,f]
>>> f()
2
>>> g()
1
```

1.13 Object Deletion

Objects can be deleted from Python's memory by using **del**, e.g.

```
>>> del x
```

NOTE CAREFULLY THAT THIS IS DIFFERENT FROM DELETION FROM A LIST OR DICTIONARY. If you use **remove()** or **pop()**, for instance, you are simply removing the pointer to the object from the given data structure, but as long as there is at least one **reference**, i.e. a pointer, to an object, that object still takes up space in memory.

This can be a major issue in long-running programs. If you are not careful to delete objects, or if they are not simply garbage-collected when their scope disappears, you can accumulate more and more of them, and have a very serious memory problem. If you see your machine running ever more slowly while a program executes, you should immediately suspect this.

1.14 Object Comparison

One can use the < operator to compare sequences, e.g.

```
if x < y:
```

for lists **x** and **y**. The comparison is **lexicographic**. This “dictionary” ordering first compares the first element of one sequence to the first element of the other. If they aren’t equal, we’re done. If not, we compare the second elements, etc.

For example,

```
>>> [12,16] < [12,88]  # 12 = 12 but 16 < 88
True
>>> [5,88] > [12,16]  # 5 is not > 12 (even though 88 > 16)
False
```

Of course, since strings are sequences, we can compare them too:

```
>>> ’abc’ < ’tuv’
True
>>> ’xyz’ < ’tuv’
False
>>> ’xyz’ != ’tuv’
True
```

Note the effects of this on, for example, the **max()** function:

```
>>> max([[1, 2], [0], [12, 15], [3, 4, 5], [8, 72]])
[12, 15]
>>> max([8,72])
72
```

We can set up comparisons for non-sequence objects, e.g. class instances, by defining a __**cmp()**__ function in the class. The definition starts with

```
def __cmp__(self,other):
```

It must be defined to return a negative, zero or positive value, depending on whether **self** is less than, equal to or greater than **other**.

Very sophisticated sorting can be done if one combines Python’s **sort()** function with a specialized __**cmp()**__ function.

1.15 Modules

You've often heard that it is good software engineering practice to write your code in "modular" fashion, i.e. to break it up into components, top-down style, and to make your code "reusable," i.e. to write it in such generality that you or someone else might make use of it in some other programs. Unlike a lot of follow-like-sheep software engineering shiboleths, this one is actually correct! :-)

A **module** is a set of classes, library functions and so on, all in one file. Unlike Perl, there are no special actions to be taken to make a file a module. Any file whose name has a **.py** suffix is a module![17]

1.15.1 Example Program Code

As our illustration, let's take the **textfile** class from our example above. We could place it in a *separate* file **tf.py**, with contents

```
# file tf.py

class textfile:
   ntfiles = 0  # count of number of textfile objects
   def __init__(self,fname):
      textfile.ntfiles += 1
      self.name = fname  # name
      self.fh = open(fname)  # handle for the file
      self.lines = self.fh.readlines()
      self.nlines = len(self.lines)  # number of lines
      self.nwords = 0  # number of words
      self.wordcount()
   def wordcount(self):
      "finds the number of words in the file"
      for l in self.lines:
         w = l.split()
         self.nwords += len(w)
   def grep(self,target):
      "prints out all lines containing target"
      for l in self.lines:
         if l.find(target) >= 0:
            print l
```

Note that even though our module here consists of just a single class, we could have several classes, plus global variables,[18] executable code not part of any function, etc.)

Our test program file, **tftest.py**, might now look like this:

```
# file tftest.py

```

[17] Make sure the base part of the file name begins with a letter, not, say, a digit.

[18] Though they would be global only to the module, not to a program which imports the module. See Section 1.15.6.

```
import tf

a = tf.textfile('x')
b = tf.textfile('y')
print "the number of text files open is", tf.textfile.ntfiles
print "here is some information about them (name, lines, words):"
for f in [a,b]:
   print f.name,f.nlines,f.nwords
a.grep('example')
```

1.15.2 How `import` Works

The Python interpreter, upon seeing the statement **import tf**, would load the contents of the file **tf.py**.[19] Any executable code in **tf.py** is then executed, in this case

```
ntfiles = 0  # count of number of textfile objects
```

(The module's executable code might not only be within classes.)

Later, when the interpreter sees the reference to **tf.textfile**, it would look for an item named **textfile** within the module **tf**, i.e. within the file **tf.py**, and then proceed accordingly.

An alternative approach would be:

```
from tf import textfile

a = textfile('x')
b = textfile('y')
print "the number of text files open is", textfile.ntfiles
print "here is some information about them (name, lines, words):"
for f in [a,b]:
   print f.name,f.nlines,f.nwords
a.grep('example')
```

This saves typing, since we type only "textfile" instead of "tf.textfile," making for less cluttered code. But arguably it is less safe (what if **tftest.py** were to have some other item named **textfile**?) and less clear (**textfile**'s origin in **tf** might serve to clarify things in large programs).

The statement

```
from tf import *
```

[19] In our context here, we would probably place the two files in the same directory, but we will address the issue of search path later.

would import everything in **tf.py** in this manner.

In any event, by separating out the **textfile** class, we have helped to modularize our code, and possibly set it up for reuse.

1.15.3 Using reload() to Renew an Import

Say you are using Python in interactive mode, and are doing code development in a text editor at the same time. If you change the module, simply running **import** again won't bring you the next version. Use **reload()** to get the latter, e.g.

```
reload(tf)
```

1.15.4 Compiled Code

Like the case of Java, the Python interpreter compiles any code it executes to **byte code** for the Python virtual machine. If the code is imported, then the compiled code is saved in a file with suffix **.pyc**, so it won't have to be recompiled again later. Running byte code is faster, since the interpreter doesn't need to translate the Python syntax anymore.

Since modules are objects, the names of the variables, functions, classes etc. of a module are attributes of that module. Thus they are retained in the **.pyc** file, and will be visible, for instance, when you run the **dir()** function on that module (Section 1.24.1).

1.15.5 Miscellaneous

A module's (free-standing, i.e. not part of a function) code executes immediately when the module is imported.

Modules are objects. They can be used as arguments to functions, return values from functions, etc.

The list **sys.modules** shows all modules ever imported into the currently running program.

1.15.6 A Note on Global Variables Within Modules

Python does not truly allow global variables in the sense that C/C++ do. An imported Python module will not have direct access to the globals in the module which imports it, nor vice versa.

For instance, consider these two files, **x.py**,

```
# x.py

import y

def f():
   global x
   x = 6

def main():
   global x
   x = 3
   f()
   y.g()

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

and **y.py**:

```
# y.py

def g():
   global x
   x += 1
```

The variable **x** in **x.py** is visible throughout the module **x.py**, but not in **y.py**. In fact, execution of the line

```
   x += 1
```

in the latter will cause an error message to appear, “global name ’x’ is not defined.” Let’s see why.

The line above the one generating the error message,

```
   global x
```

is telling the Python interpreter that there will be a global variable **x** *in this module*. But when the interpreter gets to the next line,

```
   x += 1
```

the interpreter says, “Hey, wait a minute! You can’t assign to **x** its old value plus 1. It doesn’t have an old value! It hasn’t been assigned to yet!” In other words, the interpreter isn’t treating the **x** in the module **y.py** to be the same as the one in **x.py**.

You can, however, refer to the **x** in **y.py** while you are in **x.py**, as **y.x**.

1.15.7 Data Hiding

Python has no strong form of data hiding comparable to the **private** and other such constructs in C++. It does offer a small provision of this sort, though:

If you prepend an underscore to a variable's name in a module, it will not be imported if the **from** form of **import** is used. For example, if in the module **tf.py** in Section 1.15.1 were to contain a variable **z**, then a statement

```
from tf import *
```

would mean that **z** is accesible as just **z** rather than **tf.z**. If on the other hand we named this variable **_z**, then the above statement would not make this variable accessible as **_z**; we would need to use **tf._z**. Of course, the variable would still be visible from outside the module, but by requiring the **tf.** prefix we would avoid confusion with similarly-named variables in the importing module.

A double underscore results in mangling, with another underscore plus the name of the module prepended.

1.16 Packages

As mentioned earlier, one might place more than one class in a given module, if the classes are closely related. A generalization of this arises when one has several modules that are related. Their contents may not be so closely related that we would simply pool them all into one giant module, but still they may have a close enough relationship that you want to group them in some other way. This is where the notion of a **package** comes in.

For instance, you may write some libraries dealing with some Internet software you've written. You might have one module **web.py** with classes you've written for programs which do Web access, and another module **em.py** which is for e-mail software. Instead of combining them into one big module, you could keep them as separate files put in the same directory, say **net**.

To make this directory a package, simply place a file **__init__.py** in that directory. The file can be blank, or in more sophisticated usage can be used for some startup operations.

In order to import these modules, you would use statements like

```
import net.web
```

This tells the Python interpreter to look for a file **web.py** within a directory **net**. The latter, or more precisely, the parent of the latter, must be in your Python search path, which is a collection of directories in which the interpreter will look for modules.

If for example the full path name for **net** were

```
/u/v/net
```

then the directory **/u/v** would need to be in your Python search path.

The Python search path is stored in an environment variable for your operating system. If you are on a Linux system, for example, and are using the C shell, you could type

```
setenv PYTHONPATH /u/v
```

If you have several special directories like this, string them all together, using colons as delimiters:

```
setenv PYTHONPATH /u/v:/aa/bb/cc
```

You can access the current path from within a Python program in the variable **sys.path**. It consists of a list of strings, one string for each directory, separated by colons. It can be printed out or changed by your code, just like any other variable.[20]

Package directories often have subdirectories, subsubdirectories and so on. Each one must contain a **__init__.py** file.

1.17 Exception Handling (Not Just for Exceptions!)

By the way, Python's built-in and library functions have no C-style error return code to check to see whether they succeeded. Instead, you use Python's **try/except** exception-handling mechanism, e.g.

```
try:
   f = open(sys.argv[1])
except:
   print 'open failed:',sys.argv[1]
```

Here's another example:

```
try:
   i = 5
   y = x[i]
except:
   print 'no such index:', i
```

[20] Remember, you do have to import **sys** first.

But the Python idiom also uses this for code which is not acting in an exception context. Say for example we want to find the index of the number 8 in the list **z**, with the provision that if there is no such number, to first add it to the list. The "ordinary" way would be something like this:

```
# return first index of n in x; if n is not in x, then append it first

def where(x,n):
   if n in x: return x.index(n)
   x.append(n)
   return len(x) - 1
```

Let's try it:

```
>>> x = [5,12,13]
>>> where(x,12)
1
>>> where(x,88)
3
>>> x
[5, 12, 13, 88]
```

But we could also do it with **try/except**:

```
def where1(x,n):
   try:
      return x.index(n)
   except:
      x.append(n)
      return len(x) - 1
```

As seen above, you use **try** to check for an exception; you use **raise** to raise one.

1.18 Docstrings

There is a double-quoted string, "finds the number of words in the file", at the beginning of **wordcount()** in the code in Section 1.11.1. This is called a **docstring**. It serves as a kind of comment, but at runtime, so that it can be used by debuggers and the like. Also, it enables users who have only the compiled form of the method, say as a commercial product, access to a "comment." Here is an example of how to access it, using **tf.py** from above:

```
>>> import tf
>>> tf.textfile.wordcount.__doc__
'finds the number of words in the file'
```

A docstring typically spans several lines. To create this kind of string, use triple quote marks.

By the way, did you notice above how the docstring is actually an attribute of the function, this case **tf.textfile.wordcount.__doc__**? Try typing

```
>>> dir(tf.textfile.wordcount.__doc__)
```

to see the others. You can call **help()** on any of them to see what they do.

1.19 Named Arguments in Functions

Consider this little example:

```
def f(u,v=2):
   return u+v

def main():
   x = 2;
   y = 3;
   print f(x,y)  # prints 5
   print f(x,v=y)  # prints 5, clearer code
   print f(x)  # prints 4

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

Here, the argument **v** is called a **named argument**, with **default value** 2. The "ordinary" argument **u** is called a **mandatory argument**, as it must be specified while **v** need not be. Another term for **u** is **positional argument**, as its value is inferred by its position in the order of declaration of the function's arguments. Mandatory arguments must be declared before named arguments.

1.20 Terminal I/O Issues

1.20.1 Keyboard Input

The **raw_input()** function will display a prompt and read in what is typed. For example,

```
name = raw_input('enter a name: ')
```

would display "enter a name:", then read in a response, then store that response in **name**. Note that the user input is returned in string form, and needs to be converted if the input consists of numbers.

If you don't want the prompt, don't specify one:

```
>>> y = raw_input()
3
>>> y
'3'
```

Alternatively, you can directly specify **stdin**:

```
>>> import sys
>>> z = sys.stdin.readlines()
abc
de
f
>>> z
['abc\n', 'de\n', 'f\n']
```

After typing 'f', I hit ctrl-d to close the **stdin** file.)

1.20.2 Printing Without a Newline or Blanks

A **print** statement automatically prints a newline character. To suppress it, add a trailing comma. For example:

```
print 5,  # nothing printed out yet
print 12  # '5 12' now printed out, with end-of-line
```

The **print** statement automatically separates items with blanks. To suppress blanks, use the string-concatenation operator, **+**, and possibly the **str()** function, e.g.

```
x = 'a'
y = 3
print x+str(y)  # prints 'a3'
```

By the way, **str(None)** is **None**.

1.21 Extended Example: Creating Linked Data Structures in Python

Below is a Python class for implementing a binary tree. The comments should make the program self-explanatory (no pun intended).[21]

[21] But did you get the pun?

```
# bintree.py, a module for handling sorted binary trees; values to be
# stored can be general, not just numeric, as long as an ordering
# relation exists

# here, only have routines to insert and print, but could add delete,
# etc.

class treenode:
   # create a 1-node tree, storing value v; later this node will be
   # added to some tree
   def __init__(self,v):
      self.value = v;
      self.left = None;
      self.right = None;
   # inserts the node nd into tree rooted at self; typically that tree
   # will be a subtree of a larger tree
   def ins(self,nd):
      m = nd.value
      if m < self.value:
         if self.left == None:
            self.left = nd
         else:
            self.left.ins(nd)
      else:
         if self.right == None:
            self.right = nd
         else:
            self.right.ins(nd)
   # finds max value in the subtree rooted at self
   def max(self):  # no name conflict
      s = self.value
      if self.right != None: s = max(s,self.right.max())
      return s
   def prnt(self):  # prints the subtree rooted at self
      if self.value == None: return
      if self.left != None: self.left.prnt()
      print self.value
      if self.right != None: self.right.prnt()

class tree:
   def __init__(self):
      # tree starts out empty, no nodes
      self.root = None
   # create a new tree node, contents m, and add it to the specified
   # tree
   def insrt(self,m):
      newnode = treenode(m)
      if self.root == None:
         self.root = newnode
         return
      self.root.ins(newnode)
```

And here is a test:

```
# trybt1.py:  test of bintree.py
# usage:  python trybt.py numbers_to_insert
```

```

import sys
import bintree

def main():
    tr = bintree.tree()
    for n in sys.argv[1:]:
       tr.insrt(int(n))
    tr.root.prnt()

if __name__ == ’__main__’: main()
```

The good thing about Python is that we can use the same code again for nonnumerical objects, as long as they are comparable. (Recall Section 1.14.) So, we can do the same thing with strings, using the **tree** and **treenode** classes AS IS, NO CHANGE, e.g.

```
# trybt2.py:  test of bintree.py

# usage:  python trybt.py strings_to_insert

import sys
import bintree

def main():
    tr = bintree.tree()
    for s in sys.argv[1:]:
       tr.insrt(s)
    tr.root.prnt()

if __name__ == ’__main__’: main()
```

```
% python trybt2.py abc tuv 12
12
abc
tuv
```

Or even

```
# trybt3.py:  test of bintree.py

import bintree

def main():
    tr = bintree.tree()
    tr.insrt([12,’xyz’])
    tr.insrt([15,’xyz’])
    tr.insrt([12,’tuv’])
    tr.insrt([2,’y’])
    tr.insrt([20,’aaa’])
    tr.root.prnt()

if __name__ == ’__main__’: main()
```

```
% python trybt3.py
[2, 'y']
[12, 'tuv']
[12, 'xyz']
[15, 'xyz']
[20, 'aaa']
```

1.22 Making Use of Python Idioms

In the example in Section 1.11.1, it is worth calling special attention to the line

```
for f in [a,b]:
```

where **a** and **b** are objects of type **textfile**. This illustrates the fact that the elements within a list do not have to be scalars, and that we can loop through a nonscalar list. Much more importantly, it illustrates that really effective use of Python means staying away from classic C-style loops and expressions with array elements. This is what makes for much cleaner, clearer and elegant code. It is where Python really shines.

You should almost never use C/C++ style **for** loops—i.e. where an index (say **j**), is tested against an upper bound (say **j** < **10**), and incremented at the end of each iteration (say **j++**).

Indeed, you can often avoid explicit loops, and should do so whenever possible. For example, the code

```
self.lines = self.fh.readlines()
self.nlines = len(self.lines)
```

in that same program is much cleaner than what we would have in, say, C. In the latter, we would need to set up a loop, which would read in the file one line at a time, incrementing a variable **nlines** in each iteration of the loop.[22]

Another great way to avoid loops is to use Python's **functional programming** features, described in Chapter 3.

Making use of Python idioms is often referred to by the *pythonistas* as the *pythonic* way to do things.

1.23 Decorators

Recall our example in Section 1.11.7 of how to designate a method as a class method:

[22] By the way, note the reference to an object within an object, **self.fh**.

```
   def totfiles():
      print "the total number of text files is", textfile.ntfiles
   totfiles = staticmethod(totfiles)
```

That third line does the designation. But isn't it kind of late? It comes as a surprise, thus making the code more difficult to read. Wouldn't it be better to warn the reader ahead of time that this is going to be a class method? We can do this with a **decorator**:

```
   @staticmethod
   def totfiles():
      print "the total number of text files is", textfile.ntfiles
```

Here we are telling the Python interpreter, "OK, here's what we're going to do. I'm going to define a function **totfiles()**, and then, interpreter, I want you to use that function as input to **staticmethod()**, and then reassign the output back to **totfiles()**.

So, we are really doing the same thing, but in a syntactically more readable manner.

You can do this in general, feeding one function into another via decorators. This enables some very fancy, elegant ways to produce code, somewhat like macros in C/C++. However, we will not pursue that here.

1.24 Online Documentation

1.24.1 The dir() Function

There is a very handy function **dir()** which can be used to get a quick review of what a given object or function is composed of. You should use it often.

To illustrate, in the example in Section 1.11.1 suppose we stop at the line

```
print "the number of text files open is", textfile.ntfiles
```

Then we might check a couple of things with **dir()**, say:

```
(Pdb) dir()
['a', 'b']
(Pdb) dir(textfile)
['__doc__', '__init__', '__module__', 'grep', 'wordcount', 'ntfiles']
```

When you first start up Python, various items are loaded. Let's see:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'IOError',
'ImportError', 'IndentationError', 'IndexError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None',
'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError',
'OverflowWarning', 'PendingDeprecationWarning', 'ReferenceError',
'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_',
'__debug__', '__doc__', '__import__', '__name__', 'abs', 'apply',
'basestring', 'bool', 'buffer', 'callable', 'chr', 'classmethod', 'cmp',
'coerce', 'compile', 'complex', 'copyright', 'credits', 'delattr',
'dict', 'dir', 'divmod', 'enumerate', 'eval', 'execfile', 'exit',
'file', 'filter', 'float', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'intern', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map',
'max', 'min', 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit',
'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

Well, there is a list of all the builtin functions and other attributes for you!

Want to know what functions and other attributes are associated with dictionaries?

```
>>> dir(dict)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__',
'__doc__', '__eq__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setitem__', '__str__', 'clear', 'copy',
'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys',
'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update',
'values']
```

Suppose we want to find out what methods and attributes are associated with strings. As mentioned in Section 1.5.3, strings are now a built-in class in Python, so we can't just type

```
>>> dir(string)
```

But we can use any string object:

```
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__str__', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'replace', 'rfind', 'rindex', 'rjust', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

1.24.2 The help() Function

For example, let's find out about the **pop()** method for lists:

```
>>> help(list.pop)

Help on method_descriptor:

pop(...)
    L.pop([index]) -> item -- remove and return item at index (default
last)
(END)
```

And the **center()** method for strings:

```
>>> help(''.center)
Help on function center:

center(s, width)
    center(s, width) -> string

    Return a center version of s, in a field of the specified
    width. padded with spaces as needed.  The string is never
    truncated.
```

Hit 'q' to exit the help pager.

You can also get information by using **pydoc** at the Linux command line, e.g.

```
% pydoc string.center
[...same as above]
```

1.24.3 PyDoc

The above methods of obtaining help were for use in Python's interactive mode. Outside of that mode, in an OS shell, you can get the same information from PyDoc. For example,

```
pydoc sys
```

will give you all the information about the **sys** module.

For modules outside the ordinary Python distribution, make sure they are in your Python search path, and be sure show the “dot” sequence, e.g.

```
pydoc u.v
```

1.25 Putting All Globals into a Class

As mentioned in Section 1.2.4, instead of using the keyword **global**, we may find it clearer or more organized to group all our global variables into a class. Here, in the file **tmeg.py**, is how we would do this to modify the example in that section, **tme.py**:

```
# reads in the text file whose name is specified on the command line,
# and reports the number of lines and words

import sys

def checkline():
   glb.linecount += 1
   w = glb.l.split()
   glb.wordcount += len(w)

class glb:
   linecount = 0
   wordcount = 0
   l = []

f = open(sys.argv[1])
for glb.l in f.readlines():
   checkline()
print glb.linecount, glb.wordcount
```

Note that when the program is first loaded, the class **glb** will be executed, even before **main()** starts.

1.26 Looking at the Python Virtual Machine

One can inspect the Python virtual machine code for a program. For the program **srvr.py** in Chapter 5, I once did the following:

Running Python in interactive mode, I first imported the module **dis** (“disassembler”). I then imported the program, by typing

```
import dis
import srvr
```

(I first needed to add the usual **if __name_ == '__main__'** code, so that the program wouldn't execute upon being imported.)

I then ran

```
>>> dis.dis(srvr)
```

How do you read the code? You can get a list of Python virtual machine instructions in *Python: the Complete Reference*, by Martin C. Brown, pub. by Osborne, 2001. But if you have background in assembly language, you can probably guess what the code is doing anyway.

1.27 Running Python Scripts Without Explicitly Invoking the Interpreter

Say you have a Python script **x.py**. So far, we have discussed running it via the command[23]

```
% python x.py
```

or by importing **x.py** while in interactive mode. But if you state the location of the Python interpreter in the first line of **x.py**, e.g.

```
#! /usr/bin/python
```

and use the Linux **chmod** command to make **x.py** executable, then you can run **x.py** by merely typing

```
% x.py
```

This is necessary, for instance, if you are invoking the program from a Web page.

Better yet, you can have Linux search your environment for the location of Python, by putting this as your first line in **x.py**:

```
#! /usr/bin/env python
```

This is more portable, as different platforms may place Python in different directories.

[23] This section will be Linux-specific.

Chapter 2

File and Directory Access in Python

Lots of Python applications involve files and directories. This chapter shows you the basics.

2.1 Files

2.1.1 Some Basic File Operations

A list of Python file operations can be obtained through the Python **dir**() command:

```
>>> dir(file)
[’__class__’, ’__delattr__’, ’__doc__’, ’__getattribute__’, ’__hash__’,
’__init__’, ’__iter__’, ’__new__’, ’__reduce__’, ’__reduce_ex__’,
’__repr__’, ’__setattr__’, ’__str__’, ’close’, ’closed’, ’encoding’,
’fileno’, ’flush’, ’isatty’, ’mode’, ’name’, ’newlines’, ’next’, ’read’,
’readinto’, ’readline’, ’readlines’, ’seek’, ’softspace’, ’tell’,
’truncate’, ’write’, ’writelines’, ’xreadlines’]
```

Following is an overview of many of those operations. I am beginning in a directory **/a**, which has a file **x**, consisting of

```
a
bc
def
```

a file **y**, consisting of

```
uuu
vvv
```

and a subdirectory **b**, which is empty.

```
>>> f = open(’x’)  # open, read-only (default)
>>> f.name  # recover name from file object
’x’
>>> f.readlines()
[’a\n’, ’bc\n’, ’def\n’]
>>> f.readlines()  # already at end of file, so this gets nothing
[]
>>> f.seek(0)  # go back to byte 0 of the file
>>> f.readlines()
[’a\n’, ’bc\n’, ’def\n’]
>>> f.seek(2)  # go to byte 2 in the file
>>> f.readlines()
[’bc\n’, ’def\n’]
>>> f.seek(0)
>>> f.read()  # read to EOF, returning bytes in a string
’a\nbc\ndef\n’
>>> f.seek(0)
>>> f.read(3)  # read only 3 bytes this time
’a\nb’
```

If you need to read just one line, use **readline()**.

Starting with Python 2.3, you can use a file as an iterator:

```
>>> f = open(’x’)
>>> i = 0
>>> for l in f:
...    print ’line %d:’ % i,l[:-1]
...    i += 1
...
line 0: a
line 1: bc
line 2: def
```

Note, by the way, how we stripped off the newline character (which we wanted to do because **print** would add one and we don’t want two) by using **l[:-1]** instead of **l**.

So, how do we write to files?

```
>>> g = open(’z’,’w’)  # open for writing
>>> g.write(’It is a nice sunny day’)  # takes just one argument
>>> g.write(’ today.\n’)
>>> g.close()
>>> import os
>>> os.system(’cat z’)  # issue a shell command, which also gives an exit code
It is a nice sunny day today.
0
>>> g = open(’gy’,’w’)
>>> g.writelines([’Here is a line,\n’,’and another.\n’])
>>> os.system(’cat gy’)
0
>>> g.close()
>>> os.system(’cat gy’)
Here is a line,
and another.
0
>>> a = open(’yy’,’w’)
>>> a.write(’abc\n’)
>>> a.close()
>>> os.system(’cat yy’)
abc
0
>>> b = open(’yy’,’a’)  # open in append mode
>>> b.write(’de\n’)
>>> b.close()
>>> os.system(’cat yy’)
abc
de
0
```

As in Unix, **stdin** and **stdout** count as files too (*file-like objects*, in Python parlance), so we can use the same operations, e.g.:

```
>>> import sys
>>> w = sys.stdin.readlines()  # keyboard input, ctrl-d to end
moon, sun
and
stars
>>> w
[’moon, sun\n’, ’and\n’, ’stars\n’]
```

2.2 Directories

2.2.1 Some Basic Directory Operations

The **os** module is huge. Here’s a start to seeing how big it is:

```
>>> import os
>>> dir(os)
[’EX_CANTCREAT’, ’EX_CONFIG’, ’EX_DATAERR’, ’EX_IOERR’, ’EX_NOHOST’,
’EX_NOINPUT’, ’EX_NOPERM’, ’EX_NOUSER’, ’EX_OK’, ’EX_OSERR’,
’EX_OSFILE’, ’EX_PROTOCOL’, ’EX_SOFTWARE’, ’EX_TEMPFAIL’,
’EX_UNAVAILABLE’, ’EX_USAGE’, ’F_OK’, ’NGROUPS_MAX’, ’O_APPEND’,
’O_CREAT’, ’O_DIRECT’, ’O_DIRECTORY’, ’O_DSYNC’, ’O_EXCL’,
’O_LARGEFILE’, ’O_NDELAY’, ’O_NOCTTY’, ’O_NOFOLLOW’, ’O_NONBLOCK’,
’O_RDONLY’, ’O_RDWR’, ’O_RSYNC’, ’O_SYNC’, ’O_TRUNC’, ’O_WRONLY’,
’P_NOWAIT’, ’P_NOWAITO’, ’P_WAIT’, ’R_OK’, ’TMP_MAX’, ’UserDict’,
’WCOREDUMP’, ’WEXITSTATUS’, ’WIFEXITED’, ’WIFSIGNALED’, ’WIFSTOPPED’,
’WNOHANG’, ’WSTOPSIG’, ’WTERMSIG’, ’WUNTRACED’, ’W_OK’, ’X_OK’,
’_Environ’, ’__all__’, ’__builtins__’, ’__doc__’, ’__file__’,
’__name__’, ’_copy_reg’, ’_execvpe’, ’_exists’, ’_exit’,
’_get_exports_list’, ’_make_stat_result’, ’_make_statvfs_result’,
’_pickle_stat_result’, ’_pickle_statvfs_result’, ’_spawnvef’,
’_urandomfd’, ’abort’, ’access’, ’altsep’, ’chdir’, ’chmod’, ’chown’,
’chroot’, ’close’, ’confstr’, ’confstr_names’, ’ctermid’, ’curdir’,
’defpath’, ’devnull’, ’dup’, ’dup2’, ’environ’, ’error’, ’execl’,
’execle’, ’execlp’, ’execlpe’, ’execv’, ’execve’, ’execvp’, ’execvpe’,
’extsep’, ’fchdir’, ’fdatasync’, ’fdopen’, ’fork’, ’forkpty’,
’fpathconf’, ’fstat’, ’fstatvfs’, ’fsync’, ’ftruncate’, ’getcwd’,
’getcwdu’, ’getegid’, ’getenv’, ’geteuid’, ’getgid’, ’getgroups’,
’getloadavg’, ’getlogin’, ’getpgid’, ’getpgrp’, ’getpid’, ’getppid’,
’getsid’, ’getuid’, ’isatty’, ’kill’, ’killpg’, ’lchown’, ’linesep’,
’link’, ’listdir’, ’lseek’, ’lstat’, ’major’, ’makedev’, ’makedirs’,
’minor’, ’mkdir’, ’mkfifo’, ’mknod’, ’name’, ’nice’, ’open’, ’openpty’,
’pardir’, ’path’, ’pathconf’, ’pathconf_names’, ’pathsep’, ’pipe’,
’popen’, ’popen2’, ’popen3’, ’popen4’, ’putenv’, ’read’, ’readlink’,
’remove’, ’removedirs’, ’rename’, ’renames’, ’rmdir’, ’sep’, ’setegid’,
’seteuid’, ’setgid’, ’setgroups’, ’setpgid’, ’setpgrp’, ’setregid’,
’setreuid’, ’setsid’, ’setuid’, ’spawnl’, ’spawnle’, ’spawnlp’,
’spawnlpe’, ’spawnv’, ’spawnve’, ’spawnvp’, ’spawnvpe’, ’stat’,
’stat_float_times’, ’stat_result’, ’statvfs’, ’statvfs_result’,
’strerror’, ’symlink’, ’sys’, ’sysconf’, ’sysconf_names’, ’system’,
’tcgetpgrp’, ’tcsetpgrp’, ’tempnam’, ’times’, ’tmpfile’, ’tmpnam’,
’ttyname’, ’umask’, ’uname’, ’unlink’, ’unsetenv’, ’urandom’, ’utime’,
’wait’, ’waitpid’, ’walk’, ’write’]
```

To see some examples, we start in the directory **/a** as above.

```
>>> os.path.curdir  # get current directory (Pythonic humor)
’.’
>>> t = os.path.abspath(os.path.curdir)  # get its full path
>>> t
’/a’
>>> os.path.basename(t)  # get the tail end of the path
’a’
>>> os.getcwd()  # another way to get the full path
’/a’
>>> cwd = os.getcwd()
>>> os.listdir(cwd)  # see what’s in this directory (excl. ’.’,’..’)
[’b’, ’x’, ’y’, ’z’]
>>> os.stat(’x’)  # get all the information about a file
(33152, 5079544L, 773L, 1, 0, 0, 9L, 1112979025, 1112977272, 1112977272)
>>> # the numbers above are: inode protection mode; inode number; device
>>> # inode resides on; number of links to the inode; user id of the owner;
>>> # group id of the owner; size in bytes; time of last access; time
>>> # of last modification; "ctime" as reported by OS
>>> os.path.getsize(’x’)  # get file size; argument must be string
9L
>>> os.path.isdir(’x’)  # is this file a directory?
False
>>> os.path.isdir(’b’)
True
>>> os.path.getmtime(’x’)  # time of last modification (seconds since Epoch)
1112977272
```

```
>>> import time
>>> time.ctime(os.path.getmtime('x'))  # translate that to English
'Fri Apr  8 09:21:12 2005'
>>> os.chdir('b')  # change to subdirectory b
>>> os.listdir('.')
[]
>>> os.mkdir('c')  # make a new directory
>>> os.listdir('.')
['c']
```

2.2.2 Example: Finding a File

The function **findfile()** searches for a file (which could be a directory) in the specified directory tree, returning the full path name of the first instance of the file found with the specified name, or returning None if not found.

For instance, suppose we have the directory tree **/a** shown in Section 2.1.1, except that **/b** contains a file **z**. Then the code

```
print findfile('/a','y')
print findfile('/a','b')
print findfile('/a','u')
print findfile('/a','z')
print findfile('/a/b','z')
```

produces the output

```
/a/y
/a/b
None
/a/b/z
/a/b/z
```

```
import os

# returns full path name of flname in the tree rooted at treeroot;
# returns None if not found; directories do count as finding the file
def findfile(treeroot,flname):
   os.chdir(treeroot)
   currfls = os.listdir('.')
   for fl in currfls:
      if fl == flname:
         return os.path.abspath(fl)
   for fl in currfls:
      if os.path.isdir(fl):
         tmp = findfile(fl,flname)
         if not tmp == None: return tmp
   return None

def main():
   print findfile('/a','y')
   print findfile('/a','u')
   print findfile('/a','z')
   print findfile('/a/b','z')
```

2.2.3 The Powerful `walk()` Function

The function **os.path.walk()** does a recursive descent down a directory tree, stopping in each subdirectory to perform user-coded actions. This is quite a powerful tool.[1]

The form of the call is

```
os.path.walk(rootdir,f,arg)
```

where **rootdir** is the name of the root of the desired directory tree, **f()** is a user-supplied function, and **arg** will be one of the arguments to **f()**, as explained below.

At each directory **d** visited in the "walk," **walk()** will call **f()**, and will provide **f()** with the list **flst** of the names of the files in **d**. In other words, **walk()** will make this call:

```
f(arg,d,flst)
```

So, the user must write **f()** to perform whatever operations she needs for the given directory. Remember, the user sets **arg** too. According to the Python help file for **walk()**, in many applications the user sets **arg** to **None** (though not in our example here).

The following example is adapted from code written by Leston Buell[2], which found the total number of bytes in a directory tree. The differences are: Buell used classes to maintain the data, while I used a list; I've added a check for linked files; and I've added code to also calculate the number of files and directories (the latter counting the root directory).

[1] Unix users will recognize some similarity to the Unix **find** command.

[2] `http://fizzylogic.com/users/bulbul/programming/dirsize.py`

```
# walkex.py; finds the total number of bytes, number of files and number
# of directories in a given directory tree, dtree (current directory if
# not specified); adapted from code by Leston Buell

# usage:
#    python walkex.py [dtree_root]

import os, sys

def getlocaldata(sms,dr,flst):
   for f in flst:
      # get full path name relative to where program is run; the
      # function os.path.join() adds the proper delimiter for the OS,
      #    e.g. / for Unix, \ for Windows
      fullf = os.path.join(dr,f)
      if os.path.islink(fullf): continue  # don’t count linked files
      if os.path.isfile(fullf):
         sms[0] += os.path.getsize(fullf)
         sms[1] += 1
      else:
         sms[2] += 1

def dtstat(dtroot):
   sums = [0,0,1]  # 0 bytes, 0 files, 1 directory so far
   os.path.walk(dtroot,getlocaldata,sums)
   return sums

def main():
   try:
      root = sys.argv[1]
   except:
      root = ’.’
   report = dtstat(root)
   print report

if __name__== ’__main__’:
   main()
```

Important feature: When **walk()** calls the user-supplied function **f()**, transmitting the list **flist** of files in the currently-visited directory, **f()** may modify that list. The key point is that **walk()** will continue to use that list to find more directories to visit.

For example, suppose there is a subdirectory **qqq** in the currently-visited directory. The function **f()** could delete **qqq** from **flist**, with the result being that **walk()** will NOT visit the subtree having **qqq** as its root.[3]

As another example, here is a function that descends a directory tree, removing all empty directories:

```
# descends the directory tree rooted at rootdir, removing all empty
# directories

import os
```

[3] Of course, if it has already visited that subtree, that can’t be undone.

```

def checklocalempty(dummyarg,dir,flst):
    for f in flst:
        fullf = os.path.join(dir,f)
        if os.path.isdir(fullf):
            subdirfls = os.listdir(fullf)
            if subdirfls == []:  os.rmdir(fullf)

def rmnulldirs(rootdir):
    fullrootdir = os.path.abspath(rootdir)
    os.path.walk(fullrootdir,checklocalempty,None)

def main():
    rmnulldirs('.')
```

2.3 Cross-Platform Issues

Like most scripting languages, Python is nominally cross-platform, usable on Unix, Windows and Macs. It makes a very serious attempt to meet this goal, and succeeds reasonably well. Let's take a closer look at this.

2.3.1 The Small Stuff

Some aspects cross-platform compatibility are easy to deal with. One example of this is file path naming, e.g. **/a/b/c** in Unix and **\a\b\c** in Windows. We saw above that the library functions such as **os.path.abspath()** will place slashes or backslashes according to our underlying OS, thus enabling platform-independent code. In fact, the quantity **os.sep** stores the relevant character, '/' for Unix and '\' for Windows.

Similarly, in Unix, the end-of-line marker (EOL) is a single byte, 0xa, while for Windows it is a pair of bytes, 0xd and 0xa. The EOL marker is stored in **os.linesep**.

2.3.2 How Is It Done?

Let's take a look at the **os** module, which will be in your file **/usr/lib/python2.4/os.py** or something similar. You will see code like

```
_names = sys.builtin_module_names
...
```

```
if 'posix' in _names:
    name = 'posix'
    linesep = '\n'
    from posix import *
    try:
        from posix import _exit
    except ImportError:
        pass
    import posixpath as path

    import posix
    __all__.extend(_get_exports_list(posix))
    del posix
```

(POSIX is the name of "standard" Unix.)

2.3.3 Python and So-Called "Binary" Files

Keep in mind that the term **binary file** is a misnomer. After all, ANY file is "binary," whether it consists of "text" or not, in the sense that it consists of bits, no matter what. So what do people mean when they refer to a "binary" file?

First, let's define the term **text file** to mean a file satisifying all of the following conditions:

(a) Each byte in the file is in the ASCII range 00000000-01111111, i.e. 0-127.

(b) Each byte in the file is *intended* to be thought of as an ASII character.

(c) The file is *intended* to be broken into what we think of (and typically display) as lines. Here the term *line* is defined technically in terms of end-of-line markers.

Any file which does not satisfy the above conditions has traditionally been termed a **binary file**.[4]

The default mode of Python in reading a file is to assume it is a text file. Whenever an EOL marker is encountered, it will be converted to '\n', i.e. 0xa. This would of course be no problem on Unix platforms, since it would involve no change at all, but under Windows, if the file were actually a binary file and simply by coincidence contained some 0xd 0xa pairs, one byte from each pair would be lost, with potentially disastrous consequences. So, you must warn the system if it is a binary file. For example,

```
f = open('yyy','rb')
```

would open the file **yyy** in read-only mode, and treat the file as binary.

[4] Even this definition is arguably too restrictive. If we produce a non-English file which we intend as "text," it will have some non-ASCII bytes.

Chapter 3

Functional Programming in Python

Python includes some functional programming features, used heavily by the “Pythonistas.” You should find them useful too.

These features provide concise ways of doing things which, though certainly doable via more basic constructs, compactify your code and thus make it easier to write and read. They may also make your code run much faster. Moreover, it may help us avoid bugs, since a lot of the infracture we’d need to write ourselves, which would be bug-prone, is automatically taken care of us by the functional programming constructs.

Except for the first feature here (lambda functions), these features eliminate the need for explicit loops and explicit references to list elements. As mentioned in Section 1.22, this makes for cleaner, clearer code.

3.1 Lambda Functions

Lambda functions provide a way of defining short functions. They help you avoid cluttering up your code with a lot of definitions of “one-liner” functions that are called only once. For example:

```
>>> g = lambda u:u*u
>>> g(4)
16
```

Note carefully that this is NOT a typical usage of lambda functions; it was only to illustrate the syntax. Usually a lambda functions would not be defined in a free-standing manner as above; instead, it would be defined inside other functions, as seen next.

Here is a more realistic illustration, redoing the sort example from Section 1.5.4:

```
>>> x = [[1,4],[5,2]]
>>> x
[[1, 4], [5, 2]]
>>> x.sort()
>>> x
[[1, 4], [5, 2]]
>>> x.sort(lambda u,v: u[1]-v[1])
>>> x
[[5, 2], [1, 4]]
```

A bit of explanation is necessary. If you look at the online help for **sort()**, you'll find that the definition to be

```
sort(...)
    L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
    cmp(x, y) -> -1, 0, 1
```

You see that the first argument is a named argument (recall Section 1.19), **cmp**. That is our compare function, which we defined above by writing `lambda u,v: u[1]-v[1].`

At any rate, the point is that using a lambda function above made the code more compact and readable.

The general form of a lambda function is

```
lambda arg 1, arg 2, ...: expression
```

So, multiple arguments are permissible, but the function body itself must be an expression.

3.2 Mapping

The **map()** function converts one sequence to another, by applying the same function to each element of the sequence. For example:

```
>>> z = map(len,["abc","clouds","rain"])
>>> z
[3, 6, 4]
```

So, we have avoided writing an explicit **for** loop, resulting in code which is a little cleaner, easier to write and read. In a large setting, it may give us a good speed increase too.

In the example above we used a built-in function, **len()**. We could also use our own functions; frequently these are conveniently expressed as lambda functions, e.g.:

```
>>> x = [1,2,3]
>>> y = map(lambda z: z*z, x)
>>> y
[1, 4, 9]
```

The condition that a lambda function's body consist only of an expression is rather limiting, for instance not allowing if-then-else constructs. If you really wish to have the latter, you could use a workaround. For example, to implement something like

```
if u > 2: u = 5
```

we could work as follows:

```
>>> x = [1,2,3]
>>> g = lambda u: (u > 2) * 5 + (u <= 2) * u
>>> map(g,x)
[1, 2, 5]
```

Or, a little fancier:

```
>>> g = lambda u: (u > 2) * 5 or u
>>> map(g,x)
[1, 2, 5]
```

Clearly, this is not feasible except for simple situations. For more complex cases, we would use a non-lambda function.

You can use **map()** with more than one data argument. For instance :

```
>>> map(max,(5,12,13),range(7,10))
[7, 12, 13]
```

Here I used Python's built-in **max()** function, but of course you can write your own functions to use in **map()**. And the function need not be scalar-valued. For example:

```
>>> map(lambda u,v: (min(u,v),max(u,v)),(5,12,13),range(7,10))
[(5, 7), (8, 12), (9, 13)]
```

3.3 Filtering

The **filter()** function works like **map()**, except that it culls out the sequence elements which satisfy a certain condition. The function which **filter()** is applied to must be boolean-valued, i.e. return the desired true or false value. For example:

```
>>> x = [5,12,-2,13]
>>> y = filter(lambda z: z > 0, x)
>>> y
[5, 12, 13]
```

Again, this allows us to avoid writing a **for** loop and an **if** statement.

3.4 Reduction

The **reduce()** function is used for applying the sum or other arithmetic-like operation to a list. For example,

```
>>> x = reduce(lambda x,y: x+y, range(5))
>>> x
10
```

Here **range(5)** is of course [0,1,2,3,4]. What **reduce()** does is it first adds the first two elements of [0,1,2,3,4], i.e. with 0 playing the role of **x** and 1 playing the role of **y**. That gives a sum of 1. Then that sum, 1, plays the role of **x** and the next element of [0,1,2,3,4], 2, plays the role of **y**, yielding a sum of 3, etc. Eventually **reduce()** finishes its work and returns a value of 10.

Once again, this allowed us to avoid a **for** loop, plus a statement in which we initialize **x** to 0 before the **for** loop.

3.5 List Comprehension

This allows you to compactify a **for** loop that produces a list. For example:

```
>>> x = [(1,-1), (12,5), (8,16)]
>>> y = [(v,u) for (u,v) in x]
>>> y
[(-1, 1), (5, 12), (16, 8)]
```

This is more compact than first initializing **y** to [], then having a **for** loop in which we call **y.append()**.

It gets even more compact when done in nested form. Say for instance we have a list of lists which we want to concatenate together, ignoring the first element in each. Here's how we could do it using list comprehensions:

```
>>> y
[[0, 2, 22], [1, 5, 12], [2, 3, 33]]
>>> [a for b in y for a in b[1:]]
[2, 22, 5, 12, 3, 33]
```

Here is pseudocode for what is going on:

```
for b = [0, 2, 22],  [1, 5, 12],  [2, 3, 33]
   for a in b[1:]
      emit a
```

Is that compactness worth the loss of readability? Only you can decide. It is amusing that the official Python documentation says, “If youve got the stomach for it, list comprehensions can be nested” (`http://docs.python.org/tutorial/datastructures.html`).

3.6 Example: Textfile Class Revisited

Here is the text file example from Section 1.11.1 again, now redone with functional programming features:

```
class textfile:
   ntfiles = 0  # count of number of textfile objects
   def __init__(self,fname):
      textfile.ntfiles += 1
      self.name = fname  # name
      self.fh = open(fname)  # handle for the file
      self.lines = self.fh.readlines()
      self.nlines = len(self.lines)  # number of lines
      self.nwords = 0  # number of words
      self.wordcount()

   def wordcount(self):
      ”finds the number of words in the file”
      self.nwords = \
         reduce(lambda x,y: x+y,map(lambda line: len(line.split()),self.lines))
   def grep(self,target):
      ”prints out all lines containing target”
      lines = filter(lambda line: line.find(target) >= 0,self.lines)
      print lines

a = textfile(’x’)
b = textfile(’y’)
print ”the number of text files open is”, textfile.ntfiles
print ”here is some information about them (name, lines, words):”
for f in [a,b]:
   print f.name,f.nlines,f.nwords
```

```
a.grep('example')
```

3.7 Example: Prime Factorization

The function **primefact()** below finds the prime factorization of number, relative to the given primes. For example, the call **primefact([2,3,5,7,11],24)** would return [2,3], [3,1]], meaning that $24 = 2^3 3^1$. (It is assumed that the prime factorization of **n** does indeed exist for the numbers in **primes**.)

```
def dividetomax(p,m):
    k = 0
    while True:
        if m % p != 0: return (p,k)
        k += 1
        m /= p

def primefact(primes,n):
    def divmax(p): return dividetomax(p,n)
    tmp = map(divmax,primes)
    print tmp
    tmp = filter(lambda u: u[1] > 0,tmp)
    return tmp
```

Chapter 4

Network Programming with Python

The old ad slogan of Sun Microsystems was "The network is the computer." Though Sun has changed (now part of Oracle), the concept has not. The continuing computer revolution simply wouldn't exist without networks.

4.1 Overview of Networks

The TCP/IP network protocol suite is the standard method for intermachine communication. Though originally integral only to the UNIX operating system, its usage spread to all OS types, and it is the basis of the entire Internet. This document will briefly introduce the subject of TCP/IP programming using the Python language. See `http://heather.cs.ucdavis.edu/~matloff/Networks/Intro/NetIntro.pdf` for a more detailed introduction to networks and TCP/IP.

4.1.1 Networks and MAC Addresses

A **network** means a Local Area Network. Here machines are all on a single cable, or as is more common now, what amounts to a single cable (e.g. multiple wires running through a switch). The machines on the network communicate with each other by the **MAC addresses**, which are 48-bit serial numbers burned into their network interface cards (NICs). If machine A wishes to send something to machine B on the same network, A will put B's MAC address into the message packet. B will see the packet, recognize its own MAC address as destination, and accept the packet.

4.1.2 The Internet and IP Addresses

An internet—note the indefinite article and the lower-case *i*—is simply a connection of two or more networks. One starts, say, with two networks and places a computer on both of them (so it will have two NICs). Machines on one network can send to machines on the other by sending to the computer in common, which is acting as a **router**. These days, many routers are not full computers, but simply boxes that do only routing. One of these two networks can be then connected to a third in the same way, so we get a three-network internet, and so on. In some cases, two networks are connected by having a machine on one network connected to a machine on the other via a high-speed phone line, or even a satellite connection.

The Internet—note the definite article and the capital *I*—consists of millions of these networks connected in that manner.

On the Internet, every machine has an Internet Protocol (IP) address. The original ones were 32 bits wide, and the new ones 128. If machine A on one network wants to send to machine Z on an distant network, A sends to a router, which sends to another router and so on until the message finally reaches Z's network. The local router there then puts Z's MAC address in the packet, and places the packet on that network. Z will see it and accept the packet.

Note that when A sends the packet to a local router, the latter may not know how to get to Z. But it will send to another router "in that direction," and after this process repeats enough times, Z will receive the message. Each router has only limited information about the entire network, but it knows enough to get the journey started.

4.1.3 Ports

The term *port* is overused in the computer world. In some cases it means something physical, such as a place into which you can plug your USB device. In our case here, though, ports are not physical. Instead, they are essentially tied to processes on a machine.

Think of our example above, where machine A sends to machine Z. That phrasing is not precise enough. Instead, we should say that a process on machine A sends to a process on machine Z. These are identified by ports, which are just numbers similar to file descriptors/handles.

So, when A sends to Z, it will send to a certain port at Z. Moreover, the packet will also state which process at A—stated in terms of a port number at A—sent the message, so that Z knows where to send a reply.

The ports below 1024 are reserved for the famous services, for example port 80 for HTTP. These are called **well-known ports**.[1] User programs use port numbers of 1024 or higher. By the way, keep in mind that a port stays in use for a few seconds after a connection is close; trying to start a connection at that port again

[1] On UNIX machines, a list of these is available in **/etc/services**. You cannot start a server at these ports unless you are acting with root privileges.

within this time period will result in an exception.

4.1.4 Connectionless and Connection-Oriented Communication

One can send one single packet at a time. We simply state that our message will consist of a single packet, and state the destination IP address or port. This is called **connectionless** communication, termed UDP under today's Internet protocol. It's simple, but not good for general use. For example, a message might get lost, and the sender would never know. Or, if the message is long, it is subject to corruption, which would not be caught by a connectionless setup, and also long messages monopolize network bandwidth.[2]

So, we usually use a **connection-oriented** method, TCP. What happens here is that a message from A to Z is first broken into pieces, which are sent separately from each other. At Z, the TCP layer of the network protocol stack in the OS will collect all the pieces, check them for errors, put them in the right order, etc. and deliver them to the proper process at Z.

We say that there is a **connection** between A and Z. Again, this is not physical. It merely is an agreement between the OSs at A and Z that they will exchange data between these processes/ports at A and Z in the orderly manner described above.

One point which is crucial to keep in mind is that under TCP, everything from machine A to Z, from the time the connection is opened to the time it is closed is considered one gigantic message. If the process at A, for instance, executes three network writes of 100 bytes each, it is considered one 300-byte message. And though that message may be split into pieces along the way to Z, the piece size will almost certainly not be 100 bytes, nor will the number of pieces likely be three. Of course, the same is true for the material sent from Z to A during this time.

This makes writing the program on the Z side more complicated. Say for instance A wishes to send four lines of text, and suppose the program at B knows it will be sent four lines of text. Under UDP, it would be natural to have the program at A send the data as four network writes, and the program at B would do four network reads.

But under TCP, no matter how many or few network writes the program at A does, the program at B will not know how many network reads to do. So, it must keep reading in a **while** loop until it receives a messages saying that A has nothing more to send and has closed the connection. This makes the program at Z harder to write. For example, that program must break the data into lines on its own. (I'm talking about *user* programs here; in other words, TCP means more trouble for *you*.)

[2] The **bandwidth** is the number of bits per second which can be sent. It should not be confused with **latency**, which is the end-to-end transit time for a message.

4.1.5 Clients and Servers

Connections are not completely symmetric.[3] Instead, a connection consists of a **client** and a **server**. The latter sits around waiting for requests for connections, while the former makes such a request.

When you surf the Web, say to **http://www.google.com**, your Web browser is a client. The program you contact at Google is a server. When a server is run, it sets up business at a certain port, say 80 in the Web case. It then waits for clients to contact it. When a client does so, the server will create a new socket, specifically for communication with that client, and then resume watching the original socket for new requests.

4.2 Our Example Client/Server Pair

As our main illustration of client/server programming in Python, we have modified a simple example in the Library Reference section of the Python documentation page, `http://www.python.org/doc/current/lib`. Here is the server, **tms.py**:

```
# simple illustration client/server pair; client program sends a string
# to server, which echoes it back to the client (in multiple copies),
# and the latter prints to the screen

# this is the server

import socket
import sys

# create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# associate the socket with a port
host = ’’  # can leave this blank on the server side
port = int(sys.argv[1])
s.bind((host, port))

# accept "call" from client
s.listen(1)
conn, addr = s.accept()
print ’client is at’, addr

# read string from client (assumed here to be so short that one call to
# recv() is enough), and make multiple copies (to show the need for the
# "while" loop on the client side)

data = conn.recv(1000000)
data = 10000 * data  # concatenate data with itself 999 times

# wait for the go-ahead signal from the keyboard (to demonstrate that
# recv() at the client will block until server sends)
```

[3] I’m speaking mainly of TCP here, but it mostly applies to UDP too.

```
z = raw_input()

# now send
conn.send(data)

# close the connection
conn.close()
```

And here is the client, **tmc.py**:

```
# simple illustration client/server pair; client program sends a string
# to server, which echoes it back to the client (in multiple copies),
# and the latter prints to the screen

# this is the client

import socket
import sys

# create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# connect to server
host = sys.argv[1]  # server address
port = int(sys.argv[2])  # server port
s.connect((host, port))

s.send(sys.argv[3])  # send test string

# read echo
i = 0
while(1):
   data = s.recv(1000000)  # read up to 1000000 bytes
   i += 1
   if (i < 5):  # look only at the first part of the message
      print data
   if not data:  # if end of data, leave loop
      break
   print ’received’, len(data), ’bytes’

# close the connection
s.close()
```

This client/server pair doesn’t do much. The client sends a test string to the server, and the server sends back multiple copies of the string. The client then prints the earlier part of that echoed material to the user’s screen, to demonstrate that the echoing is working, and also prints the amount of data received on each read, to demonstrate the “chunky” nature of TCP discussed earlier.

You should run this client/server pair before reading further.[4] Start up the server on one machine, by typing

[4] The source file from which this document is created, **PyNet.tex**, should be available wherever you downloaded the PDF file. You can get the client and server programs from the source file, rather than having to type them up yourself.

```
python tms.py 2000
```

and then start the client at another machine, by typing

```
python tmc.py server_machine_name 2000 abc
```

(Make sure to start the server before you start the client!)

The two main points to note when you run the programs are that (a) the client will block until you provide some keyboard input at the server machine, and (b) the client will receive data from the server in rather random-sized chunks.

4.2.1 Analysis of the Server Program

Let's look at the details of the server.

Line 7: We import the **socket** class from Python's library; this contains all the communication methods we need.

Line 11: We create a socket. This is very much analogous to a file handle or file descriptor in applications involving files. Internally it is a pointer to information about our connection (not yet made) to an application on another machine. Again, at this point, it is merely a placeholder. We have not undertaken any network actions yet.

The two arguments state that we wish to the socket to be an Internet socket (**socket.AF_INET**), and that it will use TCP (**socket.SOCK_STREAM**), rather than UDP (**socket.SOCK_DGRAM**). Note that the constants used in the arguments are attributes of the module **socket**, so they are preceded by 'socket.'; in C/C++, the analog is the **#include** file.

Line 16: We invoke the **socket** class' **bind()** method. Say for example we specify port 2000 on the command line when we run the server (obtained on Line 15).

When we call **bind()**, the operating system will first check to see whether port 2000 is already in use by some other process.[5] If so, an exception will be raised, but otherwise the OS will reserve port 2000 for the server. What that means is that from now on, whenever TCP data reaches this machine and specifies port 2000, that data will be copied to our server program. Note that **bind()** takes a single argument consisting of a two-element tuple, rather than two scalar arguments.

Line 19: The **listen()** method tells the OS that if any messages come in from the Internet specifying port 2000, then they should be considered to be requesting a connection to this socket.

[5] This could be another invocation of our server program, or a different program entirely. You could check this "by hand," by running the UNIX **netstat** command (Windows has something similar), but it would be better to have your program do it, using a Python **try/except** construct.

The method's argument tells the OS how many connection requests from remote clients to allow to be pending at any give time for port 2000. The argument 1 here tells the OS to allow only 1 pending connection request at a time.

We only care about one connection in this application, so we set the argument to 1. If we had set it to, say 5 (which is common), the OS would allow one active connection for this port, and four other pending connections for it. If a fifth pending request were to come it, it would be rejected, with a "connection refused" error.

That is about all **listen()** really does.

We term this socket a **listening socket**. That means its sole purpose is to accept connections with clients; it is usually not used for the actual transfer of data back and forth between clients and the server.[6]

Line 20: The **accept()** method tells the OS to wait for a connection request. It will block until a request comes in from a client at a remote machine.[7] That will occur when the client executes a **connect()** call (Line 16 of **tmc.py**). In that call, the OS at the client machine sends a connection request to the server machine, informing the latter as to (a) the Internet address of the client machine and (b) the **ephemeral** port of the client.[8]

At that point, the connection has been established. The OS on the server machine sets up a new socket, termed a **connected socket**, which will be used in the server's communication with the remote client.

You might wonder why there are separate listening and connected sockets. Typically a server will simultaneously be connected to many clients. So it needs a separate socket for communication with each client. (It then must either set up a separate thread for each one, or use nonblocking I/O. More on the latter below.)

All this releases **accept()** from its blocking status, and it returns a two-element tuple. The first element of that tuple, assigned here to **conn**, is the connected socket. Again, this is what will be used to communicate with the client (e.g. on Line 35).

The second item returned by **accept()** tells us who the client is, i.e. the Internet address of the client, in case we need to know that.[9]

Line 27: The **recv()** method reads data from the given socket. The argument states the maximum number of bytes we are willing to receive. This depends on how much memory we are willing to have our server use. It is traditionally set at 1024.

It is absolutely crucial, though, to keep in mind how TCP works in this regard. To review, consider a

[6] It could be used for that purpose, if our server only handles one client at a time.

[7] Which, technically, could be the same machine.

[8] The term here alludes to the temporary nature of this port, compared to the server port, which will extend over the span of all the clients the server deals with.

[9] When I say "we," I mean "we, the authors of this server program." That information may be optional for us, though obviously vital to the OS on the machine where the server is running. The OS also needs to know the client's ephemeral port, while "we" would almost never have a need for that.

connection set up between a client X and server Y. The entirety of data that X sends to Y is considered one giant message. If for example X sends text data in the form of 27 lines totalling 619 characters, TCP makes no distinction between one line and another; TCP simply considers it to be one 619-byte message.

Yet, that 619-byte message might not arrive all at once. It might, for instance, come into two pieces, one of 402 bytes and the other of 217 bytes. And that 402-byte piece may not consist of an integer number of lines. It may, and probably would, end somewhere in the middle of a line. For that reason, we seldom see a one-time call to **recv()** in real production code, as we see here on Line 27. Instead, the call is typically part of a loop, as can be seen starting on Line 22 of the client, **tmc.py**. In other words, here on Line 27 of the server, we have been rather sloppy, going on the assumption that the data from the client will be so short that it will arrive in just one piece. In a production program, we would use a loop.

Line 28: In order to show the need for such a loop in general, I have modified the original example by making the data really long. Recall that in Python, "multiplying" a string means duplicating it. For example:

```
>>> 3*'abc'
'abcabcabc'
```

Again, I put this in deliberately, so as to necessitate using a loop in the client, as we will see below.

Line 32: This too is inserted for the purpose of illustrating a principle later in the client. It takes some keyboard input at the server machine. The input is not actually used; it is merely a stalling mechanism.

Line 35: The server finally sends its data to the client.

Line 38: The server closes the connection. At this point, the sending of the giant message to the client is complete.[10] The closing of the connection will be sensed by the client, as discussed below.

4.2.2 Analysis of the Client Program

Now, what about the client code?

Line 16: The client makes the connection with the server. Note that both the server's Internet machine address and the server's port number are needed. As soon as this line is executed, Line 20 on the server side, which had been waiting, will finally execute.

Again, the connection itself is not physical. It merely is an agreement made between the server and client to exchange data, in agreed-upon chunk sizes, etc.

Line 18: The client sends its data to the server.

[10] However, that doesn't necessarily mean that the message has arrived at the client yet, nor even that the message has even left the server's machine yet. See below.

Lines 22ff: The client reads the message from the server. As explained earlier, this is done in a loop, because the message is likely to come in chunks. Again, even though Line 35 of the server gave the data to its OS in one piece, the OS may not send it out to the network in one piece, and thus the client must loop, repeatedly calling **recv**().

That raises the question of how the client will know that it has received the entire message sent by the server. The answer is that **recv**() will return an empty string when that occurs. And in turn, that will occur when the server executes **close()** on Line 38.[11]

Note:

- In Line 23, the client program is basically saying to the OS, "Give me whatever characters you've received so far." If the OS hasn't received any characters yet (and if the connection has not been closed), **recv**() will block until something does come in.[12]
- When the server finally does close the connection **recv()** will return an empty string.

4.3 Role of the OS

4.3.1 Basic Operation

As is the case with file functions, e.g. **os.open**(), the functions **socket.socket**(), **socket.bind**(), etc. are all wrappers to OS system calls.

The Python **socket.send()** calls the OS **send**(). The latter copies the data (which is an argument to the function) to the OS' buffer. Again, assuming we are using TCP, the OS will break the message into pieces before putting the data in the buffer. Characters from the latter are at various times picked up by the Ethernet card's device driver and sent out onto the network.

When a call to **send()** returns, that simply means that at least part of the given data has been copied from the application program to the OS' buffer. It does not mean that ALL of the data has been copied to the buffer, let alone saying that the characters have actually gotten onto the network yet, let alone saying they have reached the receiving end's OS, let alone saying they have reached the receiving end's application program. The OS will tell us how many bytes it accepted from us to send out onto the network, via the return value from the call to **send**(). (So, technically even **send()** should be in a loop, which iterates until all of our bytes have been accepted. See below.)

The OS at the receiving end will receive the data, check for errors and ask the sending side to retransmit an erroneous chunk, piece the data back to together and place it in the OS' buffer. Each call to **recv()** by the

[11] This would also occur if **conn** were to be garbage-collected when its scope ended, including the situation in which the server exits altogether.

[12] This will not be the case if the socket is **nonblocking**. More on this in Section 4.6.1.

application program on the receiving end will pick up whatever characters are currently in the buffer (up to the number specified in the argument to **recv**()).

4.3.2 How the OS Distinguishes Between Multiple Connections

When the server accepts a connection from a client, the connected socket will be given the same port as the listening socket. So, we'll have two different sockets, both using the same port. If the server has connections open with several clients, and the associated connected sockets all use the same port, how does the OS at the server machine decide which connected socket to give incoming data to for that port?

The answer lies in the fact that a connection is defined by five numbers: The server port; the client (ephemeral) port; the server IP address; the client IP address; and the protocol (TCP or UDP). Different clients will usually have different Internet addresses, so that is a distinguishing aspect. But even more importantly, two clients could be on the same machine, and thus have the same Internet address, yet still be distinguished from each other by the OS at the server machine, because the two clients would have different ephemeral addresses. So it all works out.

4.4 The `sendall()` Function

We emphasized earlier why a call to **recv()** should be put in a loop. One might also ask whether **send()** should be put in a loop too.

Unless the socket is nonblocking, **send()** will block until the OS on our machine has enough buffer space to accept at least some of the data given to it by the application program via **send()**. When it does so, the OS will tell us how many bytes it accepted, via the return value from the call to **send()**. The question is, is it possible that this will not be all of the bytes we wanted to send? If so, we need to put **send()** in a loop, e.g. something like this, where we send a string **w** via a socket **s**:

```
while(len(w) > 0):
   ns = s.send(w)  # ns will be the number of bytes sent
   w = w[ns:]  # still need to send the rest
```

The best reference on TCP/IP programming (*UNIX Network Programming*, by Richard Stevens, pub. Prentice-Hall, vol. 1, 2nd ed., p.77) says that this problem is "normally" seen only if the socket is nonblocking. However, that was for UNIX, and in any case, the best he seemed to be able to say was "normally." To be fully safe, one should put one's call to **send()** inside a loop, as shown above.

But as is often the case, Python recognizes that this is such a common operation that it should be automated. Thus Python provides the **sendall()** function. This function will not return until the entire string has been sent (in the sense stated above, i.e. completely copied to the OS' buffer).

The function **sendall()** should be used only with blocking sockets.

4.5 Sending Lines of Text

4.5.1 Remember, It's Just One Big Byte Stream, Not "Lines"

As discussed earlier, TCP regards all the data sent by a client or server as one giant message. If the data consists of lines of text, TCP will not pay attention to the demarcations between lines. This means that if your application is text/line-oriented, you must handle such demarcation yourself. If for example the client sends lines of text to the server, your server code must look for newline characters and separate lines on its own. Note too that in one call to **recv()** we might receive, say, all of one line and part of the next line, in which case we must keep the latter for piecing together with the bytes we get from our next call to **recv()**. This becomes a nuisance for the programmer.

In our example above, the client and server each execute **send()** only once, but in many applications they will alternate. The client will send something to the server, then the server will send something to the client, then the client will send something to the server, and so on. In such a situation, it will still be the case that the totality of all bytes sent by the client will be considered one single message by the TCP/IP system, and the same will be true for the server.

4.5.2 The Wonderful `makefile()` Function

As mentioned, if you are transferring text data between the client and server (in either direction), you've got to piece together each line on your own, a real pain. Not only might you get only part of a line during a receive, you might get part of the *next* line, which you would have to save for later use with reading the next line. [13] But Python allows you to avoid this work, by using the method **socket.makefile()**.

Python has the notion of a *file-like object*. This is a byte stream that you can treat as a "file," thinking of it as consisting of "lines." For example, we can invoke **readlines()**, a file function, on the standard input:

```
>>> import sys
>>> w = sys.stdin.readlines()  # ctrl-d to end input
moon, sun
and
stars
>>> w
['moon, sun\n', 'and\n', 'stars\n']
```

Well, **socket.makefile()** allows you to do this with sockets, as seen in the following example.

[13] One way around this problem would be to read one byte at a time, i.e. call **recv()** with argument 1. But this would be very inefficient, as system calls have heavy overhead.

Here we have a server which will allow anyone on the Internet to find out which processes are running on the host machine—even if they don’t have an account on that machine.[14] See the comments at the beginning of the programs for usage.

Here is the client:

```
# wps.py

# client for the server for remote versions of the w and ps commands

# user can check load on machine without logging in (or even without
# having an account on the remote machine)

# usage:

#   python wps.py remotehostname port_num {w,ps}

# e.g. python wps.py nimbus.org 8888 w would cause the server at
# nimbus.org on port 8888 to run the UNIX w command there, and send the
# output of the command back to the client here

import socket,sys

def main():

   s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
   host = sys.argv[1]
   port = int(sys.argv[2])
   s.connect((host,port))

   # send w or ps command to server
   s.send(sys.argv[3])

   # create "file-like object" flo
   flo = s.makefile(’r’,0)  # read-only, unbuffered
   # now can call readlines() on flo, and also use the fact that
   # that stdout is a file-like object too
   sys.stdout.writelines(flo.readlines())

if __name__ == ’__main__’:
   main()
```

And here is the server:

```
# svr.py

# server for remote versions of the w and ps commands

# user can check load on machine without logging in (or even without
```

[14] Some people have trouble believing this. How could you access such information without even having an account on that machine? Well, the server is willing to give it to you. Imagine what terrible security problems we’d have if the server allowed one to run any command from the client, rather than just **w** and **ps**.

```
# having an account on the remote machine)

# usage:

#   python svr.py port_num

import socket,sys,os

def main():

   ls = socket.socket(socket.AF_INET,socket.SOCK_STREAM);
   port = int(sys.argv[1])
   ls.bind(('', port))

   while (1):
      ls.listen(1)
      (conn, addr) = ls.accept()
      print 'client is at', addr
      # get w or ps command from client
      rc = conn.recv(2)
      # run the command in a Unix-style pipe
      ppn = os.popen(rc)  # do ppn.close() to close
      # ppn is a "file-like object," so can apply readlines()
      rl = ppn.readlines()
      # create a file-like object from the connection socket
      flo = conn.makefile('w',0)  # write-only, unbuffered
      # write the lines to the network
      flo.writelines(rl[:-1])
      # clean up
      # must close both the socket and the wrapper
      flo.close()
      conn.close()

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

Note that there is no explicit call to **recv()** in the client code. The call to **readlines()** basically triggers such a call. And the reason we're even allowed to call **readlines()** is that we called **makefile()** and created **flo**.

4.5.3 Getting the Tail End of the Data

A common bug in network programming is that most of a data transfer works fine, but the receiving program hangs at the end, waiting to receive the very last portion of the data. The cause of this is that, for efficiency reasons, the TCP layer at the sending end generally waits until it accumulates a "large enough" chunk of data before it sending out. As long as the sending side has not closed the connection, the TCP layer there assumes that the program may be sending more data, and TCP will wait for it.

The simplest way to deal with this is for the sending side to close the connection. Its TCP layer then knows that it will be given no more data by the sending program, and thus TCP needs to send out whatever it has. This means that both client and server programs must do a lot of opening and closing of socket connections, which is inefficient.[15]

Also, if you are using **makefile()**, it would be best to not use **readlines()** directly, as this may cause the receiving end to wait until the sender closes the connection, so that the "file" is complete. It may be safer to do something like this:

```
# flo is an already-created file-like object from calling makefile()
for line in flo:
   ...
```

This way we are using **flo** as an iterator, which will not require all lines to be received before the loop is started.

An alternative is to use UDP instead of TCP. With UDP, if you send an n-byte message, it will all be send immediately, in one piece, and received as such. However, you then lose the reliability of TCP.[16]

A related problem with **popen()** is discussed at `http://www.popekim.com/2008/12/never-use-pipe-with-p`
`html`.

4.6 Dealing with Asynchronous Inputs

In many applications a machine, typically a server, will be in a position in which input could come from several network sources, without knowing which one will come next. One way of dealing with that is to take a threaded approach, with the server having a separate thread for each possible client. Another way is to use nonblocking sockets.

4.6.1 Nonblocking Sockets

Note our statement that **recv()** blocks until either there is data available to be read or the sender has closed the connection holds only if the socket is in blocking mode. That mode is the default, but we can change a socket to nonblocking mode by calling **setblocking()** with argument 0.[17]

[15] Yet such inefficiency is common, and is used for example in the HTTP (i.e. Web access) protocol. Each time you click the mouse on a given Web page, for instance, there is a new connection made. It is this lack of **state** in the protocol that necessitates the use of **cookies**. Since each Web action involves a separate connection, there is no "memory" between the actions. This would mean, for example, that if the Web site were password-protected, the server would have to ask you for your password at every single action, quite a nuisance. The workaround is to have your Web browser write a file to your local disk, recording that you have already passed the password test.

[16] TCP does error checking, including checking for lost packets. If the client and server are multiple hops apart in the Internet, it's possible that some packets will be lost when an intermediate router has buffer overflow. If TCP at the receiving end doesn't receive a packet by a certain **timeout** period, it will ask TCP at the sending end to retransmit the packet. Of course, we could do all this ourselves in UDP by adding complexity to our code, but that would defeat the purpose.

[17] As usual, this is done in a much cleaner, easier manner than in C/C++. In Python, one simple function call does it.

One calls **recv()** in nonblocking mode as part of a **try/except** pair. If data is available to be read from that socket, **recv()** works as usual, but if no data is available, an exception is raised. While one normally thinks of exceptions as cases in which a drastic execution error has occurred, in this setting it simply means that no data is yet available on this socket, and we can go to try another socket or do something else.

Why would we want to do this? Consider a server program which is connected to multiple clients simultaneously. Data will come in from the various clients at unpredictable times. The problem is that if we were to simply read from each one in order, it could easily happen that **read()** would block while reading from one client while data from another client is ready for reading. Imagine that you are the latter client, while the former client is out taking a long lunch! You'd be unable to use the server all that time!

One way to handle this is to use threads, setting up one thread for each client. Indeed, I have an example of this in Chapter 5. But threads programming can be tricky, so one may turn to the alternative, which is to make the client sockets nonblocking.

The example below does nothing useful, but is a simple illustration of the principles. Each client keeps sending letters to the server; the server concatenates all the letters it receives, and sends the concatenated string back to a client whenever the client sends a character.

In a sample run of these programs, I started the server, then one client in one window, then another client is another window. (For convenience, I was doing all of this on the same machine, but a better illustration would be to use three different machines.) In the first window, I typed ’a’, then ’b’, then ’c’. Then I moved to the other window, and typed ’u’, ’u’, ’v’ and ’v’. I then went back to the first window and typed ’d’, etc. I ended each client session by simply hitting Enter instead of typing a letter.

Here is what happened at the terminal for the first client:

```
% python strclnt.py localhost 2000
enter a letter:a
a
enter a letter:b
ab
enter a letter:c
abc
enter a letter:d
abcuuvvd
enter a letter:e
abcuuvvdwwe
enter a letter:
```

Here is what happened at the terminal for the second client:

```
% python strclnt.py localhost 2000
enter a letter:u
abcu
enter a letter:u
abcuu
```

```
enter a letter:v
abcuuv
enter a letter:v
abcuuvv
enter a letter:w
abcuuvvdw
enter a letter:w
abcuuvvdww
enter a letter:
```

Here is what happened at the terminal for the server:

```
% python strsvr.py 2000
the final value of v is abcuuvvdwwe
```

Note that I first typed at the first client, but after typing ’c’, I switched to the second client. After hitting the second ’v’, I switched back to the first, etc.

Now, let’s see the code. First, the client:

```
# strclnt.py:  simple illustration of nonblocking sockets

# two clients connect to server; each client repeatedly sends a letter k,
# which the server appends to a global string v and reports it to the
# client; k = ’’ means the client is dropping out; when all clients are
# gone, server prints the final string v

# this is the client; usage is
#    python clnt.py server_address server_port_number

import socket, sys

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = sys.argv[1]  # server address
port = int(sys.argv[2])  # server port
s.connect((host, port))

while 1:
   # get letter
   k = raw_input(’enter a letter:’)
   s.send(k)  # send k to server
   # if stop signal, then leave loop
   if k == ’’: break
   v = s.recv(1024)  # receive v from server (up to 1024 bytes, assumed
                     # to come in one chunk)
   print v

s.close() # close socket
```

And here is the server:

```
# strsvr.py:  simple illustration of nonblocking sockets

# multiple clients connect to server; each client repeatedly sends a
# letter k, which the server adds to a global string v and echos back
# to the client; k = ’’ means the client is dropping out; when all
# clients are gone, server prints final value of v

# this is the server; usage is
#    python strsvr.py server_port_number

import socket, sys

# set up listening socket
lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

port = int(sys.argv[1])
# bind lstn socket to this port
lstn.bind((’’, port))
lstn.listen(5)

# initialize concatenate string, v
v = ’’

# initialize client socket list
cs = []

# in this example, a fixed number of clients, nc
nc = 2

# accept connections from the clients
for i in range(nc):
   (clnt,ap) = lstn.accept()
   # set this new socket to nonblocking mode
   clnt.setblocking(0)
   cs.append(clnt)

# now loop, always accepting input from whoever is ready, if any, until
# no clients are left
while len(cs) > 0:
   # get next client, with effect of a circular queue
   clnt = cs.pop(0)
   cs.append(clnt)
   # something ready to read from clnt? clnt closed connection?
   try:
      k = clnt.recv(1)  # try to receive one byte; if none is ready
                        # yet, that is the "exception"
      if k == ’’:
         clnt.close()
         cs.remove(clnt)
      v += k  # update the concatenated list
      clnt.send(v)
   except:  pass

lstn.close()
print ’the final value of v is’, v
```

Here we only set the client sockets in nonblocking mode, not the listening socket. However, if we wished to allow the number of clients to be unknown at the time execution starts, rather than a fixed number known ahead of time as in our example above, we would be forced to either make the listening socket nonblocking, or use threads.

Note once again that the actual setting of blocking/nonblocking mode is done by the OS. Python's **setblocking()** function merely makes system calls to make this happen. It should be said, though, that this Python function is far simpler to use than what must be done in C/C++ to set the mode.

4.6.2 Advanced Methods of Polling

In our example of nonblocking sockets above, we had to "manually" check whether a socket was ready to read. Today most OSs can automate that process for you. The "traditional" way to do this was via a UNIX system call named **select()**, which later was adopted by Windows as well. The more modern way for UNIX is another call, **poll()** (not yet available on Windows). Python has interfaces to both of these system calls, in the **select** module. Python also has modules **asyncore** and **asynchat** for similar purposes. I will not give the details here.

4.7 Troubleshooting

Network program debugging can be tricky. Here are some tips:

- As always, be sure to use a debugging tool!
- The **socket.getsockname()** can be used to determine the IP address and port number of a given socket, and **socket.getpeername()** will give you the same information for this socket's **peer**, i.e. the machine on the other end of the TCP connection.
- The following Unix commands list current connections:

  ```
  netstat -A inet
  lsof -i
  ```

A program which has an open socket **s** can determine its own IP address and port number by the call **s.getsockname()**.

4.8 Other Libraries

Python has libraries for FTP, HTML processing and so on. In addition, a higher-level library which is quite popular is Twisted.

4.9 Web Operations

There is a wealth of library code available for Web operations. Here we look at only a short introductory program covering a couple of the many aspects of this field:

```
# getlinks.py:  illustrates HTML parsing in Python

import urllib  # library for accessing Web sites
import HTMLParser  # library for parsing HTML

# basic idea:  Python provides the HTMLParser class within the
# HTMLParser module; the idea is to subclass it, overriding some of the
# member functions, in this case handle_starttag()
class GetLinks(HTMLParser.HTMLParser):
   # arguments to handle_starttag(), supplied by the class during its
   # parsing actions:
   # tag:  an HTML tag, e.g. 'a' arising from '<a href>'
   # attrs:  a two-tuples, again supplied by the class during
   # parsing, that give the value for the given HTML name;
   # e.g. when the class encounters
   # '<a href="http://heather.cs.ucdavis.edu/search.html">'
   # tag will be 'a', name will be 'href' and value will be
   # 'http://heather.cs.ucdavis.edu/search.html'
   def handle_starttag(self,tag,attrs):
      if tag == 'a':
         for name,value in attrs:
            if name == 'href': print value  # print URL link

gl = GetLinks()  # construct a class instance
url = 'http://www.ucdavis.edu/index.html'  # target URL
# open network connection; returns a file-like object, so can be read
urlconn = urllib.urlopen(url)
# read, and put the downloaded HTML code into urlcontents
urlcontents = urlconn.read()
# input the downloaded material into HTMLParser's member function feed
# for parsing
gl.feed(urlcontents)
```

The program reads the raw HTML code from a Web page, and then extracts the URL links. The comments explain the details.

Chapter 5

Parallel Python: Threads and Multiprocessing Modules

(Francis Hsu contributed sections of this chapter.)

There are a number of ways to write parallel Python code.[1]

5.1 The Python Threads and Multiprocessing Modules

Python's thread system builds on the underlying OS threads. They are thus pre-emptible. Note, though, that Python adds its own threads manager on top of the OS thread system; see Section 5.1.3.

5.1.1 Python Threads Modules

Python threads are accessible via two modules, **thread.py** and **threading.py**. The former is more primitive, thus easier to learn from, and we will start with it.

[1] This chapter is shared by two of my open source books: `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf` and `http://heather.cs.ucdavis.edu/~matloff/Python/PLN/FastLanePython.pdf`. If you wish to more about the topics covered in the book other than the one you are now reading, please check the other!

5.1.1.1 The `thread` Module

The example here involves a client/server pair.[2] As you'll see from reading the comments at the start of the files, the program does nothing useful, but is a simple illustration of the principles. We set up two invocations of the client; they keep sending letters to the server; the server concatenates all the letters it receives.

Only the server needs to be threaded. It will have one thread for each client.

Here is the client code, **clnt.py**:

```
# simple illustration of thread module

# two clients connect to server; each client repeatedly sends a letter,
# stored in the variable k, which the server appends to a global string
# v, and reports v to the client; k = ’’ means the client is dropping
# out; when all clients are gone, server prints the final string v

# this is the client; usage is

#    python clnt.py server_address port_number

import socket  # networking module
import sys

# create Internet TCP socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = sys.argv[1]  # server address
port = int(sys.argv[2])  # server port

# connect to server
s.connect((host, port))

while(1):
   # get letter
   k = raw_input(’enter a letter:’)
   s.send(k)  # send k to server
   # if stop signal, then leave loop
   if k == ’’: break
   v = s.recv(1024)  # receive v from server (up to 1024 bytes)
   print v

s.close() # close socket
```

And here is the server, **srvr.py**:

```
# simple illustration of thread module

```

[2] It is preferable here that the reader be familiar with basic network programming. See my tutorial at `http://heather.cs.ucdavis.edu/~matloff/Python/PLN/FastLanePython.pdf`. However, the comments preceding the various network calls would probably be enough for a reader without background in networks to follow what is going on.

```
# multiple clients connect to server; each client repeatedly sends a
# letter k, which the server adds to a global string v and echos back
# to the client; k = ’’ means the client is dropping out; when all
# clients are gone, server prints final value of v

# this is the server

import socket  # networking module
import sys

import thread

# note the globals v and nclnt, and their supporting locks, which are
#    also global; the standard method of communication between threads is
#    via globals

# function for thread to serve a particular client, c
def serveclient(c):
   global v,nclnt,vlock,nclntlock
   while 1:
      # receive letter from c, if it is still connected
      k = c.recv(1)
      if k == ’’: break
      # concatenate v with k in an atomic manner, i.e. with protection
      #    by locks
      vlock.acquire()
      v += k
      vlock.release()
      # send new v back to client
      c.send(v)
   c.close()
   nclntlock.acquire()
   nclnt -= 1
   nclntlock.release()

# set up Internet TCP socket
lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

port = int(sys.argv[1])  # server port number
# bind lstn socket to this port
lstn.bind((’’, port))
# start listening for contacts from clients (at most 2 at a time)
lstn.listen(5)

# initialize concatenated string, v
v = ’’
# set up a lock to guard v
vlock = thread.allocate_lock()

# nclnt will be the number of clients still connected
nclnt = 2
# set up a lock to guard nclnt
nclntlock = thread.allocate_lock()

# accept calls from the clients
for i in range(nclnt):
   # wait for call, then get a new socket to use for this client,
   #    and get the client’s address/port tuple (though not used)
   (clnt,ap) = lstn.accept()
   # start thread for this client, with serveclient() as the thread's
   #    function, with parameter clnt; note that parameter set must be
   #    a tuple; in this case, the tuple is of length 1, so a comma is
   #    needed
   thread.start_new_thread(serveclient,(clnt,))

# shut down the server socket, since it's not needed anymore
lstn.close()

# wait for both threads to finish
while nclnt > 0: pass

print 'the final value of v is', v
```

Make absolutely sure to run the programs before proceeding further.[3] Here is how to do this:

I'll refer to the machine on which you run the server as **a.b.c**, and the two client machines as **u.v.w** and **x.y.z.**[4] First, on the server machine, type

```
python srvr.py 2000
```

and then on each of the client machines type

```
python clnt.py a.b.c 2000
```

(You may need to try another port than 2000, anything above 1023.)

Input letters into both clients, in a rather random pattern, typing some on one client, then on the other, then on the first, etc. Then finally hit Enter without typing a letter to one of the clients to end the session for that client, type a few more characters in the other client, and then end that session too.

The reason for threading the server is that the inputs from the clients will come in at unpredictable times. At any given time, the server doesn't know which client will send input next, and thus doesn't know on which client to call **recv()**. One way to solve this problem is by having threads, which run "simultaneously" and thus give the server the ability to read from whichever client has sent data.[5].

So, let's see the technical details. We start with the "main" program.[6]

```
vlock = thread.allocate_lock()
```

[3] You can get them from the **.tex** source file for this tutorial, located wherever your picked up the **.pdf** version.

[4] You could in fact run all of them on the same machine, with address name **localhost** or something like that, but it would be better on separate machines.

[5] Another solution is to use nonblocking I/O. See this example in that context in `http://heather.cs.ucdavis.edu/~matloff/Python/PyNet.pdf`

[6] Just as you should write the main program first, you should read it first too, for the same reasons.

Here we set up a **lock variable** which guards **v**. We will explain later why this is needed. Note that in order to use this function and others we needed to import the **thread** module.

```
nclnt = 2
nclntlock = thread.allocate_lock()
```

We will need a mechanism to insure that the "main" program, which also counts as a thread, will be passive until both application threads have finished. The variable **nclnt** will serve this purpose. It will be a count of how many clients are still connected. The "main" program will monitor this, and wrap things up later when the count reaches 0.

```
thread.start_new_thread(serveclient,(clnt,))
```

Having accepted a a client connection, the server sets up a thread for serving it, via **thread.start_new_thread()**. The first argument is the name of the application function which the thread will run, in this case **serveclient()**. The second argument is a tuple consisting of the set of arguments for that application function. As noted in the comment, this set is expressed as a tuple, and since in this case our tuple has only one component, we use a comma to signal the Python interpreter that this is a tuple.

So, here we are telling Python's threads system to call our function **serveclient()**, supplying that function with the argument **clnt**. The thread becomes "active" immediately, but this does not mean that it starts executing right away. All that happens is that the threads manager adds this new thread to its list of threads, and marks its current state as Run, as opposed to being in a Sleep state, waiting for some event.

By the way, this gives us a chance to show how clean and elegant Python's threads interface is compared to what one would need in C/C++. For example, in **pthreads**, the function analogous to **thread.start_new_thread()** has the signature

```
pthread_create (pthread_t *thread_id, const pthread_attr_t *attributes,
   void *(*thread_function)(void *), void *arguments);
```

What a mess! For instance, look at the types in that third argument: A pointer to a function whose argument is pointer to `void` and whose value is a pointer to `void` (all of which would have to be `cast` when called). It's such a pleasure to work in Python, where we don't have to be bothered by low-level things like that.

Now consider our statement

```
while nclnt > 0: pass
```

The statement says that as long as at least one client is still active, do nothing. Sounds simple, and it is, but you should consider what is really happening here.

Remember, the three threads—the two client threads, and the “main” one—will take turns executing, with each turn lasting a brief period of time. Each time “main” gets a turn, it will loop repeatedly on this line. But all that empty looping in “main” is wasted time. What we would really like is a way to prevent the “main” function from getting a turn at all until the two clients are gone. There are ways to do this which you will see later, but we have chosen to remain simple for now.

Now consider the function **serveclient()**. Any thread executing this function will deal with only one particular client, the one corresponding to the connection **c** (an argument to the function). So this **while** loop does nothing but read from that particular client. If the client has not sent anything, the thread will block on the line

```
k = c.recv(1)
```

This thread will then be marked as being in Sleep state by the thread manager, thus allowing the other client thread a chance to run. If neither client thread can run, then the “main” thread keeps getting turns. When a user at one of the clients finally types a letter, the corresponding thread unblocks, i.e. the threads manager changes its state to Run, so that it will soon resume execution.

Next comes the most important code for the purpose of this tutorial:

```
vlock.acquire()
v += k
vlock.release()
```

Here we are worried about a **race condition**. Suppose for example **v** is currently ’abx’, and Client 0 sends **k** equal to ’g’. The concern is that this thread’s turn might end in the middle of that addition to **v**, say right after the Python interpreter had formed ’abxg’ but before that value was written back to **v**. This could be a big problem. The next thread might get to the same statement, take **v**, still equal to ’abx’, and append, say, ’w’, making **v** equal to ’abxw’. Then when the first thread gets its next turn, it would finish its interrupted action, and set **v** to ’abxg’—which would mean that the ’w’ from the other thread would be lost.

All of this hinges on whether the operation

```
v += k
```

is interruptible. Could a thread’s turn end somewhere in the midst of the execution of this statement? If not, we say that the operation is **atomic**. If the operation were atomic, we would not need the lock/unlock operations surrounding the above statement. I did this, using the methods described in Section 5.1.3.5, and it appears to me that the above statement is *not* atomic.

Moreover, it’s safer not to take a chance, especially since Python compilers could vary or the virtual machine could change; after all, we would like our Python source code to work even if the machine changes.

So, we need the lock/unlock operations:

```
vlock.acquire()
v += k
vlock.release()
```

The lock, **vlock** here, can only be held by one thread at a time. When a thread executes this statement, the Python interpreter will check to see whether the lock is locked or unlocked right now. In the latter case, the interpreter will lock the lock and the thread will continue, and will execute the statement which updates **v**. It will then release the lock, i.e. the lock will go back to unlocked state.

If on the other hand, when a thread executes **acquire()** on this lock when it is locked, i.e. held by some other thread, its turn will end and the interpreter will mark this thread as being in Sleep state, waiting for the lock to be unlocked. When whichever thread currently holds the lock unlocks it, the interpreter will change the blocked thread from Sleep state to Run state.

Note that if our threads were non-preemptive, we would not need these locks.

Note also the crucial role being played by the global nature of **v**. Global variables are used to communicate between threads. In fact, recall that this is one of the reasons that threads are so popular—easy access to global variables. Thus the dogma so often taught in beginning programming courses that global variables must be avoided is wrong; on the contrary, there are many situations in which globals are necessary and natural.[7]

The same race-condition issues apply to the code

```
nclntlock.acquire()
nclnt -= 1
nclntlock.release()
```

Following is a Python program that finds prime numbers using threads. Note carefully that it is not claimed to be efficient at all (it may well run more slowly than a serial version); it is merely an illustration of the concepts. Note too that we are again using the simple **thread** module, rather than **threading**.

```
#!/usr/bin/env python

import sys
import math
import thread

def dowork(tn):  # thread number tn
   global n,prime,nexti,nextilock,nstarted,nstartedlock,donelock
   donelock[tn].acquire()
   nstartedlock.acquire()
   nstarted += 1
   nstartedlock.release()
```

[7] I think that dogma is presented in a far too extreme manner anyway. See `http://heather.cs.ucdavis.edu/~matloff/globals.html`.

```
13    lim = math.sqrt(n)
14    nk = 0
15    while 1:
16       nextilock.acquire()
17       k = nexti
18       nexti += 1
19       nextilock.release()
20       if k > lim: break
21       nk += 1
22       if prime[k]:
23          r = n / k
24          for i in range(2,r+1):
25             prime[i*k] = 0
26    print ’thread’, tn, ’exiting; processed’, nk, ’values of k’
27    donelock[tn].release()
28
29 def main():
30    global n,prime,nexti,nextilock,nstarted,nstartedlock,donelock
31    n = int(sys.argv[1])
32    prime = (n+1) * [1]
33    nthreads = int(sys.argv[2])
34    nstarted = 0
35    nexti = 2
36    nextilock = thread.allocate_lock()
37    nstartedlock = thread.allocate_lock()
38    donelock = []
39    for i in range(nthreads):
40       d = thread.allocate_lock()
41       donelock.append(d)
42       thread.start_new_thread(dowork,(i,))
43    while nstarted < nthreads: pass
44    for i in range(nthreads):
45       donelock[i].acquire()
46    print ’there are’, reduce(lambda x,y: x+y, prime) - 2, ’primes’
47
48 if __name__ == ’__main__’:
49     main()
```

So, let’s see how the code works.

The algorithm is the famous Sieve of Erathosthenes: We list all the numbers from 2 to **n**, then cross out all multiples of 2 (except 2), then cross out all multiples of 3 (except 3), and so on. The numbers which get crossed out are composite, so the ones which remain at the end are prime.

Line 32: We set up an array **prime**, which is what we will be “crossing out.” The value 1 means “not crossed out,” so we start everything at 1. (Note how Python makes this easy to do, using list “multiplication.”)

Line 33: Here we get the number of desired threads from the command line.

Line 34: The variable **nstarted** will show how many threads have already started. This will be used later, in Lines 43-45, in determining when the **main()** thread exits. Since the various threads will be writing this variable, we need to protect it with a lock, on Line 37.

Lines 35-36: The variable **nexti** will say which value we should do “crossing out” by next. If this is, say, 17, then it means our next task is to cross out all multiples of 17 (except 17). Again we need to protect it with a lock.

Lines 39-42: We create the threads here. The function executed by the threads is named **dowork()**. We also create locks in an array **donelock**, which again will be used later on as a mechanism for determining when **main()** exits (Line 44-45).

Lines 43-45: There is a lot to discuss here.

To start, recall that in **srvr.py**, our example in Section 5.1.1.1, we didn't want the main thread to exit until the child threads were done.[8] So, Line 50 was a **busy wait**, repeatedly doing nothing (**pass**). That's a waste of time—each time the main thread gets a turn to run, it repeatedly executes **pass** until its turn is over.

Here in our primes program, a premature exit by **main()** result in printing out wrong answers. On the other hand, we don't want **main()** to engage in a wasteful busy wait. We could use **join()** from **threading.Thread** for this purpose, to be discussed later, but here we take a different tack: We set up a list of locks, one for each thread, in a list **donelock**. Each thread initially acquires its lock (Line 9), and releases it when the thread finishes its work (Lin 27). Meanwhile, **main()** has been waiting to acquire those locks (Line 45). So, when the threads finish, **main()** will move on to Line 46 and print out the program's results.

But there is a subtle problem (threaded programming is notorious for subtle problems), in that there is no guarantee that a thread will execute Line 9 before **main()** executes Line 45. That's why we have a busy wait in Line 43, to make sure all the threads acquire their locks before **main()** does. Of course, we're trying to avoid busy waits, but this one is quick.

Line 13: We need not check any "crosser-outers" that are larger than $\sqrt{n}$.

Lines 15-25: We keep trying "crosser-outers" until we reach that limit (Line 20). Note the need to use the lock in Lines 16-19. In Line 22, we check the potential "crosser-outer" for primeness; if we have previously crossed it out, we would just be doing duplicate work if we used this **k** as a "crosser-outer."

Here's one more example, a type of Web crawler. This one continually monitors the access time of the Web, by repeatedly accessing a list of "representative" Web sites, say the top 100. What's really different about this program, though, is that we've reserved one thread for human interaction. The person can, whenever he/she desires, find for instance the mean of recent access times.

```
import sys
import time
import os
import thread

class glbls:
   acctimes = []  # access times
   acclock = thread.allocate_lock()  # lock to guard access time data
```

[8] The effect of the main thread ending earlier would depend on the underlying OS. On some platforms, exit of the parent may terminate the child threads, but on other platforms the children continue on their own.

```
   nextprobe = 0  # index of next site to probe
   nextprobelock = thread.allocate_lock()  # lock to guard access time data
   sites = open(sys.argv[1]).readlines()  # the sites to monitor
   ww = int(sys.argv[2])  # window width

def probe(me):
   if me > 0:
      while 1:
         # determine what site to probe next
         glbls.nextprobelock.acquire()
         i = glbls.nextprobe
         i1 = i + 1
         if i1 >= len(glbls.sites): i1 = 0
         glbls.nextprobe = i1
         glbls.nextprobelock.release()
         # do probe
         t1 = time.time()
         os.system(’wget --spider -q ’+glbls.sites[i1])
         t2 = time.time()
         accesstime = t2 - t1
         glbls.acclock.acquire()
         # list full yet?
         if len(glbls.acctimes) < glbls.ww:
            glbls.acctimes.append(accesstime)
         else:
            glbls.acctimes = glbls.acctimes[1:] + [accesstime]
         glbls.acclock.release()
   else:
      while 1:
         rsp = raw_input(’monitor: ’)
         if rsp == ’mean’: print mean(glbls.acctimes)
         elif rsp == ’median’: print median(glbls.acctimes)
         elif rsp == ’all’: print all(glbls.acctimes)

def mean(x):
   return sum(x)/len(x)

def median(x):
   y = x
   y.sort()
   return y[len(y)/2]  # a little sloppy

def all(x):
   return x

def main():
   nthr = int(sys.argv[3])  # number of threads
   for thr in range(nthr):
      thread.start_new_thread(probe,(thr,))
   while 1: continue

if __name__ == ’__main__’:
   main()

```

5.1.1.2 The `threading` Module

The program below treats the same network client/server application considered in Section 5.1.1.1, but with the more sophisticated **threading** module. The client program stays the same, since it didn’t involve threads in the first place. Here is the new server code:

```
# simple illustration of threading module

# multiple clients connect to server; each client repeatedly sends a
# value k, which the server adds to a global string v and echos back
# to the client; k = ’’ means the client is dropping out; when all
# clients are gone, server prints final value of v

# this is the server

import socket  # networking module
import sys
import threading

# class for threads, subclassed from threading.Thread class
class srvr(threading.Thread):
   # v and vlock are now class variables
   v = ’’
   vlock = threading.Lock()
   id = 0  # I want to give an ID number to each thread, starting at 0
   def __init__(self,clntsock):
      # invoke constructor of parent class
      threading.Thread.__init__(self)
      # add instance variables
      self.myid = srvr.id
      srvr.id += 1
      self.myclntsock = clntsock
   # this function is what the thread actually runs; the required name
   #    is run(); threading.Thread.start() calls threading.Thread.run(),
   #    which is always overridden, as we are doing here
   def run(self):
      while 1:
         # receive letter from client, if it is still connected
         k = self.myclntsock.recv(1)
         if k == ’’: break
         # update v in an atomic manner
         srvr.vlock.acquire()
         srvr.v += k
         srvr.vlock.release()
         # send new v back to client
         self.myclntsock.send(srvr.v)
      self.myclntsock.close()

# set up Internet TCP socket
lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
port = int(sys.argv[1])  # server port number
# bind lstn socket to this port
lstn.bind((’’, port))
# start listening for contacts from clients (at most 2 at a time)
lstn.listen(5)
```

```

nclnt = int(sys.argv[2])  # number of clients

mythreads = []  # list of all the threads
# accept calls from the clients
for i in range(nclnt):
   # wait for call, then get a new socket to use for this client,
   #    and get the client’s address/port tuple (though not used)
   (clnt,ap) = lstn.accept()
   # make a new instance of the class srvr
   s = srvr(clnt)
   # keep a list all threads
   mythreads.append(s)
   # threading.Thread.start calls threading.Thread.run(), which we
   #    overrode in our definition of the class srvr
   s.start()

# shut down the server socket, since it’s not needed anymore
lstn.close()

# wait for all threads to finish
for s in mythreads:
   s.join()

print ’the final value of v is’, srvr.v
```

Again, let’s look at the main data structure first:

```
class srvr(threading.Thread):
```

The **threading** module contains a class **Thread**, any instance of which represents one thread. A typical application will subclass this class, for two reasons. First, we will probably have some application-specific variables or methods to be used. Second, the class **Thread** has a member method **run()** which is meant to be overridden, as you will see below.

Consistent with OOP philosophy, we might as well put the old globals in as class variables:

```
v = ’’
vlock = threading.Lock()
```

Note that class variable code is executed immediately upon execution of the program, as opposed to when the first object of this class is created. So, the lock is created right away.

```
id = 0
```

This is to set up ID numbers for each of the threads. We don’t use them here, but they might be useful in debugging or in future enhancement of the code.

```
def __init__(self,clntsock):
   ...
   self.myclntsock = clntsock

# ``main'' program
...
   (clnt,ap) = lstn.accept()
   s = srvr(clnt)
```

The "main" program, in creating an object of this class for the client, will pass as an argument the socket for that client. We then store it as a member variable for the object.

```
def run(self):
   ...
```

As noted earlier, the **Thread** class contains a member method **run()**. This is a dummy, to be overridden with the application-specific function to be run by the thread. It is invoked by the method **Thread.start()**, called here in the main program. As you can see above, it is pretty much the same as the previous code in Section 5.1.1.1 which used the **thread** module, adapted to the class environment.

One thing that is quite different in this program is the way we end it:

```
for s in mythreads:
   s.join()
```

The **join()** method in the class **Thread** blocks until the given thread exits. (The threads manager puts the main thread in Sleep state, and when the given thread exits, the manager changes that state to Run.) The overall effect of this loop, then, is that the main program will wait at that point until all the threads are done. They "join" the main program. This is a much cleaner approach than what we used earlier, and it is also more efficient, since the main program will not be given any turns in which it wastes time looping around doing nothing, as in the program in Section 5.1.1.1 in the line

```
while nclnt > 0: pass
```

Here we maintained our own list of threads. However, we could also get one via the call **threading.enumerate()**. If placed after the **for** loop in our server code above, for instance as

```
print threading.enumerate()
```

we would get output like

```
[<_MainThread(MainThread, started)>, <srvr(Thread-1, started)>,
<srvr(Thread-2, started)>]
```

Here's another example, which finds and counts prime numbers, again not assumed to be efficient:

```
#!/usr/bin/env python

# prime number counter, based on Python threading class

# usage:  python PrimeThreading.py n nthreads
#   where we wish the count of the number of primes from 2 to n, and to
#   use nthreads to do the work

# uses Sieve of Erathosthenes:  write out all numbers from 2 to n, then
# cross out all the multiples of 2, then of 3, then of 5, etc., up to
# sqrt(n); what's left at the end are the primes

import sys
import math
import threading

class prmfinder(threading.Thread):
   n = int(sys.argv[1])
   nthreads = int(sys.argv[2])
   thrdlist = []  # list of all instances of this class
   prime = (n+1) * [1]  # 1 means assumed prime, until find otherwise
   nextk = 2  # next value to try crossing out with
   nextklock = threading.Lock()
   def __init__(self,id):
      threading.Thread.__init__(self)
      self.myid = id
   def run(self):
      lim = math.sqrt(prmfinder.n)
      nk = 0  # count of k's done by this thread, to assess load balance
      while 1:
         # find next value to cross out with
         prmfinder.nextklock.acquire()
         k = prmfinder.nextk
         prmfinder.nextk += 1
         prmfinder.nextklock.release()
         if k > lim: break
         nk += 1  # increment workload data
         if prmfinder.prime[k]:  # now cross out
            r = prmfinder.n / k
            for i in range(2,r+1):
               prmfinder.prime[i*k] = 0
      print 'thread', self.myid, 'exiting; processed', nk, 'values of k'

def main():
   for i in range(prmfinder.nthreads):
      pf = prmfinder(i)  # create thread i
      prmfinder.thrdlist.append(pf)
      pf.start()
   for thrd in prmfinder.thrdlist: thrd.join()
   print 'there are', reduce(lambda x,y: x+y, prmfinder.prime) - 2, 'primes'

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

5.1.2 Condition Variables

5.1.2.1 General Ideas

We saw in the last section that **threading.Thread.join()** avoids the need for wasteful looping in **main()**, while the latter is waiting for the other threads to finish. In fact, it is very common in threaded programs to have situations in which one thread needs to wait for something to occur in another thread. Again, in such situations we would not want the waiting thread to engage in wasteful looping.

The solution to this problem is **condition variables**. As the name implies, these are variables used by code to wait for a certain condition to occur. Most threads systems include provisions for these, and Python's **threading** package is no exception.

The **pthreads** package, for instance, has a type **pthread_cond** for such variables, and has functions such as **pthread_cond_wait()**, which a thread calls to wait for an event to occur, and **pthread_cond_signal()**, which another thread calls to announce that the event now has occurred.

But as is typical with Python in so many things, it is easier for us to use condition variables in Python than in C. At the first level, there is the class **threading.Condition**, which corresponds well to the condition variables available in most threads systems. However, at this level condition variables are rather cumbersome to use, as not only do we need to set up condition variables but we also need to set up extra locks to guard them. This is necessary in any threading system, but it is a nuisance to deal with.

So, Python offers a higher-level class, **threading.Event**, which is just a wrapper for **threading.Condition**, but which does all the condition lock operations behind the scenes, alleviating the programmer of having to do this work.

5.1.2.2 Other `threading` Classes

The function **Event.set()** "wakes" all threads that are waiting for the given event. That didn't matter in our example above, since only one thread (**main()**) would ever be waiting at a time in that example. But in more general applications, we sometimes want to wake only one thread instead of all of them. For this, we can revert to working at the level of **threading.Condition** instead of **threading.Event**. There we have a choice between using **notify()** or **notifyAll()**.

The latter is actually what is called internally by **Event.set()**. But **notify()** instructs the threads manager to wake just one of the waiting threads (we don't know which one).

The class **threading.Semaphore** offers semaphore operations. Other classes of advanced interest are **threading.RLock** and **threading.Timer**.

5.1.3 Threads Internals

The thread manager acts like a "mini-operating system." Just like a real OS maintains a table of processes, a thread system's thread manager maintains a table of threads. When one thread gives up the CPU, or has its turn pre-empted (see below), the thread manager looks in the table for another thread to activate. Whichever thread is activated will then resume execution where it had left off, i.e. where its last turn ended.

Just as a process is either in Run state or Sleep state, the same is true for a thread. A thread is either ready to be given a turn to run, or is waiting for some event. The thread manager will keep track of these states, decide which thread to run when another has lost its turn, etc.

5.1.3.1 Kernel-Level Thread Managers

Here each thread really is a process, and for example will show up on Unix systems when one runs the appropriate **ps** process-list command, say **ps axH**. The threads manager is then the OS.

The different threads set up by a given application program take turns running, among all the other processes.

This kind of thread system is is used in the Unix **pthreads** system, as well as in Windows threads.

5.1.3.2 User-Level Thread Managers

User-level thread systems are "private" to the application. Running the **ps** command on a Unix system will show only the original application running, not all the threads it creates. Here the threads are not pre-empted; on the contrary, a given thread will continue to run until it voluntarily gives up control of the CPU, either by calling some "yield" function or by calling a function by which it requests a wait for some event to occur.[9]

A typical example of a user-level thread system is **pth**.

5.1.3.3 Comparison

Kernel-level threads have the advantage that they can be used on multiprocessor systems, thus achieving true parallelism between threads. This is a major advantage.

On the other hand, in my opinion user-level threads also have a major advantage in that they allow one to produce code which is much easier to write, is easier to debug, and is cleaner and clearer. This in turn stems from the non-preemptive nature of user-level threads; application programs written in this manner

[9] In typical user-level thread systems, an external event, such as an I/O operation or a signal, will also also cause the current thread to relinquish the CPU.

typically are not cluttered up with lots of lock/unlock calls (details on these below), which are needed in the pre-emptive case.

5.1.3.4 The Python Thread Manager

Python “piggybacks” on top of the OS’ underlying threads system. A Python thread is a real OS thread. If a Python program has three threads, for instance, there will be three entries in the **ps** output.

However, Python’s thread manager imposes further structure on top of the OS threads. It keeps track of how long a thread has been executing, in terms of the number of Python **byte code** instructions that have executed.[10] When that reaches a certain number, by default 100, the thread’s turn ends. In other words, the turn can be pre-empted either by the hardware timer and the OS, or when the interpreter sees that the thread has executed 100 byte code instructions.[11]

5.1.3.5 The GIL

In the case of CPython (but not Jython or Iron Python), there is a global interpreter lock, the famous (or infamous) GIL. It is set up to ensure that only one thread runs at a time, in order to facilitate easy garbage collection.

Suppose we have a C program with three threads, which I’ll call X, Y and Z. Say currently Y is running. After 30 milliseconds (or whatever the quantum size has been set to by the OS), Y will be interrupted by the timer, and the OS will start some other process. Say the latter, which I’ll call Q, is a different, unrelated program. Eventually Q’s turn will end too, and let’s say that the OS then gives X a turn. From the point of view of our X/Y/Z program, i.e. ignoring Q, control has passed from Y to X. The key point is that the point within Y at which that event occurs is random (with respect to where Y is at the time), based on the time of the hardware interrupt.

By contrast, say my Python program has three threads, U, V and W. Say V is running. The hardware timer will go off at a random time, and again Q might be given a turn, *but* definitely neither U nor W will be given a turn, because the Python interpreter had earlier made a call to the OS which makes U and W wait for the GIL to become unlocked.

Let’s look at this a little closer. The key point to note is that the Python interpreter itself is threaded, say using **pthreads**. For instance, in our X/Y/Z example above, when you ran **ps axH**, you would see three Python processes/threads. I just tried that on my program **thsvr.py**, which creates two threads, with a command-line argument of 2000 for that program. Here is the relevant portion of the output of **ps axH**:

```
28145 pts/5    Rl     0:09 python thsvr.py 2000
```

[10] This is the “machine language” for the Python virtual machine.

[11] The author thanks Alex Martelli for a helpful clarification.

```
28145 pts/5    Sl     0:00 python thsvr.py 2000
28145 pts/5    Sl     0:00 python thsvr.py 2000
```

What has happened is the Python interpreter has spawned two child threads, one for each of my threads in **thsvr.py**, in addition to the interpreter's original thread, which runs my **main()**. Let's call those threads UP, VP and WP. Again, these are the threads that the OS sees, while U, V and W are the threads that I see—or think I see, since they are just virtual.

The GIL is a **pthreads** lock. Say V is now running. Again, what that actually means on my real machine is that VP is running. VP keeps track of how long V has been executing, in terms of the number of Python **byte code** instructions that have executed. When that reaches a certain number, by default 100, VP will release the GIL by calling **pthread_mutex_unlock()** or something similar.

The OS then says, "Oh, were any threads waiting for that lock?" It then basically gives a turn to UP or WP (we can't predict which), which then means that from my point of view U or W starts, say U. Then VP and WP are still in Sleep state, and thus so are my V and W.

So you can see that it is the Python interpreter, not the hardware timer, that is determining how long a thread's turn runs, relative to the other threads in my program. Again, Q might run too, but within this Python program there will be no control passing from V to U or W simply because the timer went off; such a control change will only occur when the Python interpreter wants it to. This will be either after the 100 byte code instructions or when U reaches an I/O operation or other wait-event operation.

So, the bottom line is that while Python uses the underlying OS threads system as its base, it superimposes further structure in terms of transfer of control between threads.

Most importantly, the presence of the GIL means that two Python threads (spawned from the same program) cannot run at the same time—*even on a multicore machine*. This has been the subject of great controversy.

5.1.3.6 Implications for Randomness and Need for Locks

I mentioned in Section 5.1.3.2 that non-pre-emptive threading is nice because one can avoid the code clutter of locking and unlocking (details of lock/unlock below). Since, barring I/O issues, a thread working on the same data would seem to always yield control at exactly the same point (i.e. at 100 byte code instruction boundaries), Python would seem to be deterministic and non-pre-emptive. However, it will not quite be so simple.

First of all, there is the issue of I/O, which adds randomness. There may also be randomness in how the OS chooses the first thread to be run, which could affect computation order and so on.

Finally, there is the question of atomicity in Python operations: The interpreter will treat any Python virtual machine instruction as indivisible, thus not needing locks in that case. But the bottom line will be that unless you know the virtual machine well, you should use locks at all times.

5.1.4 The `multiprocessing` Module

CPython's GIL is the subject of much controversy. As we saw in Section 5.1.3.5, it prevents running true parallel applications when using the **thread** or **threading** modules.

That might not seem to be too severe a restriction—after all if you really need the speed, you probably won't use a scripting language in the first place. But a number of people took the point of view that, given that they have decided to use Python no matter what, they would like to get the best speed subject to that restriction. So, there was much grumbling about the GIL.

Thus, later the **multiprocessing** module was developed, which enables true parallel processing with Python on a multiprocore machine, with an interface very close to that of the **threading** module.

Moreover, one can run a program across machines! In other words, the **multiprocessing** module allows to run several threads not only on the different cores of one machine, but on many machines at once, in cooperation in the same manner that threads cooperate on one machine. By the way, this idea is similar to something I did for Perl some years ago (PerlDSM: A Distributed Shared Memory System for Perl. *Proceedings of PDPTA 2002*, 63-68), and for which I did in R as a package **Rdsm** some time later. We will not cover the cross-machine case here.

So, let's go to our first example, a simulation application that will find the probability of getting a total of exactly k dots when we roll n dice:

```
# dice probability finder, based on Python multiprocessing class

# usage:  python DiceProb.py n k nreps nthreads
#   where we wish to find the probability of getting a total of k dots
#   when we roll n dice; we'll use nreps total repetitions of the
#   simulation, dividing those repetitions among nthreads threads

import sys
import random
from multiprocessing import Process, Lock, Value

class glbls:  # globals, other than shared
   n = int(sys.argv[1])
   k = int(sys.argv[2])
   nreps = int(sys.argv[3])
   nthreads = int(sys.argv[4])
   thrdlist = []  # list of all instances of this class

def worker(id,tot,totlock):
   mynreps = glbls.nreps/glbls.nthreads
   r = random.Random()  # set up random number generator
   count = 0  # number of times get total of k
   for i in range(mynreps):
      if rolldice(r) == glbls.k:
         count += 1
   totlock.acquire()
   tot.value += count
   totlock.release()
   # check for load balance
   print 'thread', id, 'exiting; total was', count

def rolldice(r):
   ndots = 0
   for roll in range(glbls.n):
      dots = r.randint(1,6)
      ndots += dots
   return ndots

def main():
   tot = Value('i',0)
   totlock = Lock()
   for i in range(glbls.nthreads):
      pr = Process(target=worker, args=(i,tot,totlock))
      glbls.thrdlist.append(pr)
      pr.start()
   for thrd in glbls.thrdlist: thrd.join()
   # adjust for truncation, in case nthreads doesn't divide nreps evenly
   actualnreps = glbls.nreps/glbls.nthreads * glbls.nthreads
   print 'the probability is',float(tot.value)/actualnreps

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

As in any simulation, the longer one runs it, the better the accuracy is likely to be. Here we run the simulation **nreps** times, but divide those repetitions among the threads. This is an example of an "embarrassingly parallel" application, so we should get a good speedup (not shown here).

So, how does it work? The general structure looks similar to that of the Python **threading** module, using **Process()** to create a create a thread, **start()** to get it running, **Lock()** to create a lock, **acquire()** and **release()** to lock and unlock a lock, and so on.

The main difference, though, is that globals are not automatically shared. Instead, shared variables must be created using **Value** for a scalar and **Array** for an array. Unlike Python in general, here one must specify a data type, 'i' for integer and 'd' for double (floating-point). (One can use **Namespace** to create more complex types, at some cost in performance.) One also specifies the initial value of the variable. One must pass these variables explicitly to the functions to be run by the threads, in our case above the function **worker()**. Note carefully that the shared variables are still accessed syntactically as if they were globals.

Here's the prime number-finding program from before, now using **multiprocessing**:

```
#!/usr/bin/env python

# prime number counter, based on Python multiprocessing class

# usage:  python PrimeThreading.py n nthreads
#   where we wish the count of the number of primes from 2 to n, and to
#   use nthreads to do the work

# uses Sieve of Erathosthenes:  write out all numbers from 2 to n, then
# cross out all the multiples of 2, then of 3, then of 5, etc., up to
# sqrt(n); what’s left at the end are the primes

import sys
import math
from multiprocessing import Process, Lock, Array, Value

class glbls:  # globals, other than shared
   n = int(sys.argv[1])
   nthreads = int(sys.argv[2])
   thrdlist = []  # list of all instances of this class

def prmfinder(id,prm,nxtk,nxtklock):
   lim = math.sqrt(glbls.n)
   nk = 0  # count of k’s done by this thread, to assess load balance
   while 1:
      # find next value to cross out with
      nxtklock.acquire()
      k = nxtk.value
      nxtk.value = nxtk.value + 1
      nxtklock.release()
      if k > lim: break
      nk += 1  # increment workload data
      if prm[k]:  # now cross out
         r = glbls.n / k
         for i in range(2,r+1):
            prm[i*k] = 0
   print ’thread’, id, ’exiting; processed’, nk, ’values of k’

def main():
   prime = Array(’i’,(glbls.n+1) * [1])  # 1 means prime, until find otherwise
   nextk = Value(’i’,2)  # next value to try crossing out with
   nextklock = Lock()
   for i in range(glbls.nthreads):
      pf = Process(target=prmfinder, args=(i,prime,nextk,nextklock))
      glbls.thrdlist.append(pf)
      pf.start()
   for thrd in glbls.thrdlist: thrd.join()
   print ’there are’, reduce(lambda x,y: x+y, prime) - 2, ’primes’

if __name__ == ’__main__’:
    main()
```

The main new item in this example is use of **Array()**.

One can use the **Pool** class to create a set of threads, rather than doing so “by hand” in a loop as above. You can start them with various initial values for the threads using **Pool.map()**, which works similarly to Python’s ordinary **map()** function.

The **multiprocessing** documentation warns that shared items may be costly, and suggests using **Queue** and **Pipe** where possible. We will cover the former in the next section. Note, though, that in general it’s difficult to get much speedup (or difficult even to avoid slowdown!) with non-“embarrassingly parallel” applications.

5.1.5 The Queue Module for Threads and Multiprocessing

Threaded applications often have some sort of work queue data structure. When a thread becomes free, it will pick up work to do from the queue. When a thread creates a task, it will add that task to the queue.

Clearly one needs to guard the queue with locks. But Python provides the **Queue** module to take care of all the lock creation, locking and unlocking, and so on. This means we don't have to bother with it, and the code will probably be faster.

Queue is implemented for both **threading** and **multiprocessing**, in almost identical forms. This is good, because the documentation for **multiprocessing** is rather sketchy, so you can turn to the docs for `threading` for more details.

The function **put()** in Queue adds an element to the end of the queue, while **get()** will remove the head of the queue, again without the programmer having to worry about race conditions.

Note that **get()** will block if the queue is currently empty. An alternative is to call it with **block=False**, within a **try/except** construct. One can also set timeout periods.

Here once again is the prime number example, this time done with **Queue**:

```
#!/usr/bin/env python

# prime number counter, based on Python multiprocessing class with
# Queue

# usage:  python PrimeThreading.py n nthreads
#   where we wish the count of the number of primes from 2 to n, and to
#   use nthreads to do the work

# uses Sieve of Erathosthenes:  write out all numbers from 2 to n, then
# cross out all the multiples of 2, then of 3, then of 5, etc., up to
# sqrt(n); what's left at the end are the primes

import sys
import math
from multiprocessing import Process, Array, Queue

class glbls:  # globals, other than shared
   n = int(sys.argv[1])
   nthreads = int(sys.argv[2])
   thrdlist = []  # list of all instances of this class

def prmfinder(id,prm,nxtk):
   nk = 0  # count of k's done by this thread, to assess load balance
   while 1:
      # find next value to cross out with
      try: k = nxtk.get(False)
      except: break
      nk += 1  # increment workload data
      if prm[k]:  # now cross out
         r = glbls.n / k
         for i in range(2,r+1):
            prm[i*k] = 0
   print ’thread’, id, ’exiting; processed’, nk, ’values of k’

def main():
   prime = Array(’i’,(glbls.n+1) * [1])  # 1 means prime, until find otherwise
   nextk = Queue()  # next value to try crossing out with
   lim = int(math.sqrt(glbls.n)) + 1  # fill the queue with 2...sqrt(n)
   for i in range(2,lim): nextk.put(i)
   for i in range(glbls.nthreads):
      pf = Process(target=prmfinder, args=(i,prime,nextk))
      glbls.thrdlist.append(pf)
      pf.start()
   for thrd in glbls.thrdlist: thrd.join()
   print ’there are’, reduce(lambda x,y: x+y, prime) - 2, ’primes’

if __name__ == ’__main__’:
    main()
```

The way **Queue** is used here is to put all the possible “crosser-outers,” obtained in the variable **nextk** in the previous versions of this code, into a queue at the outset. One then uses `get()` to pick up work from the queue. Look Ma, no locks!

Below is an example of queues in an in-place quicksort. (Again, the reader is warned that this is just an example, not claimed to be efficient.)

The work items in the queue are a bit more involved here. They have the form **(i,j,k)**, with the first two elements of this tuple meaning that the given array chunk corresponds to indices **i** through **j** of **x**, the original array to be sorted. In other words, whichever thread picks up this chunk of work will have the responsibility of handling that particular section of **x**.

Quicksort, of course, works by repeatedly splitting the original array into smaller and more numerous chunks. Here a thread will split its chunk, taking the lower half for itself to sort, but placing the upper half into the queue, to be available for other chunks that have not been assigned any work yet. I’ve written the algorithm so that as soon as all threads have gotten some work to do, no more splitting will occur. That’s where the value of **k** comes in. It tells us the split number of this chunk. If it’s equal to **nthreads-1**, this thread won’t split the chunk.

```
# Quicksort and test code, based on Python multiprocessing class and
# Queue

# code is incomplete, as some special cases such as empty subarrays
# need to be accounted for

# usage:  python QSort.py n nthreads
#   where we wish to test the sort on a random list of n items,
#   using nthreads to do the work

import sys
import random
```

```
from multiprocessing import Process, Array, Queue

class glbls:  # globals, other than shared
   nthreads = int(sys.argv[2])
   thrdlist = []  # list of all instances of this class
   r = random.Random(9876543)

def sortworker(id,x,q):
   chunkinfo = q.get()
   i = chunkinfo[0]
   j = chunkinfo[1]
   k = chunkinfo[2]
   if k < glbls.nthreads - 1:  # need more splitting?
      splitpt = separate(x,i,j)
      q.put((splitpt+1,j,k+1))
      # now, what do I sort?
      rightend = splitpt + 1
   else: rightend = j
   tmp = x[i:(rightend+1)]  # need copy, as Array type has no sort() method
   tmp.sort()
   x[i:(rightend+1)] = tmp

def separate(xc, low, high):  # common algorithm; see Wikipedia
   pivot = xc[low]  # would be better to take, e.g., median of 1st 3 elts
   (xc[low],xc[high]) = (xc[high],xc[low])
   last = low
   for i in range(low,high):
      if xc[i] <= pivot:
         (xc[last],xc[i]) = (xc[i],xc[last])
         last += 1
   (xc[last],xc[high]) = (xc[high],xc[last])
   return last

def main():
   tmp = []
   n = int(sys.argv[1])
   for i in range(n): tmp.append(glbls.r.uniform(0,1))
   x = Array(’d’,tmp)
   # work items have form (i,j,k), meaning that the given array chunk
   # corresponds to indices i through j of x, and that this is the kth
   # chunk that has been created, x being the 0th
   q = Queue()  # work queue
   q.put((0,n-1,0))
   for i in range(glbls.nthreads):
      p = Process(target=sortworker, args=(i,x,q))
      glbls.thrdlist.append(p)
      p.start()
   for thrd in glbls.thrdlist: thrd.join()
   if n < 25: print x[:]

if __name__ == ’__main__’:
    main()
```

5.1.6 Debugging Threaded and Multiprocessing Python Programs

Debugging is always tough with parallel programs, including threads programs. It’s especially difficult with pre-emptive threads; those accustomed to debugging non-threads programs find it rather jarring to see sudden changes of context while single-stepping through code. Tracking down the cause of deadlocks can be very hard. (Often just getting a threads program to end properly is a challenge.)

Another problem which sometimes occurs is that if you issue a “next” command in your debugging tool, you may end up inside the internal threads code. In such cases, use a “continue” command or something like that to extricate yourself.

Unfortunately, as of April 2010, I know of no debugging tool that works with **multiprocessing**. However, one can do well with **thread** and **threading**.

5.2 Using Python with MPI

(**Important note**: As of April 2010, a much more widely used Python/MPI interface is MPI4Py. It works similarly to what is described here.)

A number of interfaces of Python to MPI have been developed.[12] A well-known example is pyMPI, developed by a PhD graduate in computer science in UCD, Patrick Miller.

One writes one’s pyMPI code, say in **x.py**, by calling pyMPI versions of the usual MPI routines. To run the code, one then runs MPI on the program **pyMPI** with **x.py** as a command-line argument.

Python is a very elegant language, and pyMPI does a nice job of elegantly interfacing to MPI. Following is a rendition of Quicksort in pyMPI. Don’t worry if you haven’t worked in Python before; the “non-C-like” Python constructs are explained in comments at the end of the code.

```
# a type of quicksort; break array x (actually a Python "list") into
# p quicksort-style piles, based # on comparison with the first p-1
# elements of x, where p is the number # of MPI nodes; the nodes sort
# their piles, then return them to node 0, # which strings them all
# together into the final sorted array

import mpi  # load pyMPI module

# makes npls quicksort-style piles
def makepiles(x,npls):
   pivot = x[:npls]  # we’ll use the first npls elements of x as pivots,
                     # i.e. we’ll compare all other elements of x to these
   pivot.sort()  # sort() is a member function of the Python list class
   pls = []  # initialize piles list to empty
```

[12] If you are not familiar with Python, I have a quick tutorial at http://heather.cs.ucdavis.edu/~matloff/python.html.

```
   lp = len(pivot)  # length of the pivot array
   # pls will be a list of lists, with the i-th list in pls storing the
   # i-th pile; the i-th pile will start with ID i (to enable
   # identification later on) and pivot[i]
   for i in range(lp):  # i = 0,1,...lp-1
      pls.append([i,pivot[i]])  # build up array via append() member function
   pls.append([lp])
   for xi in x[npls:]:  # now place each element in the rest of x into
                        # its proper pile
      for j in range(lp):  # j = 0,1,...,lp-1
         if xi <= pivot[j]:
            pls[j].append(xi)
            break
         elif j == lp-1: pls[lp].append(xi)
   return pls

def main():
   if mpi.rank == 0:  # analog of calling MPI_Rank()
      x = [12,5,13,61,9,6,20,1]  # small test case
      # divide x into piles to be disbursed to the various nodes
      pls = makepiles(x,mpi.size)
   else:  # all other nodes set their x and pls to empty
      x = []
      pls = []
   mychunk = mpi.scatter(pls)  # node 0 (not an explicit argument) disburses
                               # pls to the nodes, each of which receives
                               # its chunk in its mychunk
   newchunk = []  # will become sorted version of mychunk
   for pile in mychunk:
      # I need to sort my chunk but most remove the ID first
      plnum = pile.pop(0)  # ID
      pile.sort()
      # restore ID
      newchunk.append([plnum]+pile)  # the + is array concatenation
   # now everyone sends their newchunk lists, which node 0 (again an
   # implied argument) gathers together into haveitall
   haveitall = mpi.gather(newchunk)
   if mpi.rank == 0:
      haveitall.sort()
      # string all the piles together
      sortedx = [z for q in haveitall for z in q[1:]]
      print sortedx

# common idiom for launching a Python program
if __name__ == ’__main__’: main()
```

Some examples of use of other MPI functions:

```
mpi.send(mesgstring,destnodenumber)
(message,status) = mpi.recv()  # receive from anyone
print message
(message,status) = mpi.recv(3)  # receive only from node 3
(message,status) = mpi.recv(3,ZMSG)  # receive only message type ZMSG,
                                     # only from node 3
(message,status) = mpi.recv(tag=ZMSG)  # receive from anyone, but
                                       # only message type ZMSG
```

5.2.1 Using PDB to Debug Threaded Programs

Using PDB is a bit more complex when threads are involved. One cannot, for instance, simply do something like this:

```
pdb.py buggyprog.py
```

because the child threads will not inherit the PDB process from the main thread. You can still run PDB in the latter, but will not be able to set breakpoints in threads.

What you can do, though, is invoke PDB from *within* the function which is run by the thread, by calling **pdb.set_trace()** at one or more points within the code:

```
import pdb
pdb.set_trace()
```

In essence, those become breakpoints.

For example, in our program **srvr.py** in Section 5.1.1.1, we could add a PDB call at the beginning of the loop in **serveclient()**:

```
while 1:
   import pdb
   pdb.set_trace()
   # receive letter from client, if it is still connected
   k = c.recv(1)
   if k == '': break
```

You then run the program directly through the Python interpreter as usual, NOT through PDB, but then the program suddenly moves into debugging mode on its own. At that point, one can then step through the code using the **n** or **s** commands, query the values of variables, etc.

PDB's **c** ("continue") command still works. Can one still use the **b** command to set additional breakpoints? Yes, but it might be only on a one-time basis, depending on the context. A breakpoint might work only once, due to a scope problem. Leaving the scope where we invoked PDB causes removal of the trace object. Thus I suggested setting up the trace inside the loop above.

Of course, you can get fancier, e.g. setting up "conditional breakpoints," something like:

```
debugflag = int(sys.argv[1])
...
if debugflag == 1:
   import pdb
   pdb.set_trace()
```

Then, the debugger would run only if you asked for it on the command line. Or, you could have multiple **debugflag** variables, for activating/deactivating breakpoints at various places in the code.

Moreover, once you get the `(Pdb)` prompt, you could set/reset those flags, thus also activating/deactivating breakpoints.

Note that local variables which were set before invoking PDB, including parameters, are not accessible to PDB.

Make sure to insert code to maintain an ID number for each thread. This really helps when debugging.

5.2.2 RPDB2 and Winpdb

The Winpdb debugger (`www.digitalpeers.com/pythondebugger/`),[13] is very good. Among other things, it can be used to debug threaded code, curses-based code and so on, which many debuggers can't. Winpdb is a GUI front end to the text-based RPDB2, which is in the same package. I have a tutorial on both at `http://heather.cs.ucdavis.edu/~matloff/winpdb.html`.

Another very promising debugger that handles threads is PYDB, by Rocky Bernstein (not to be confused with an earlier debugger of the same name). You can obtain it from `http://code.google.com/p/pydbgr/` or the older version at `http://bashdb.sourceforge.net/pydb/`. Invoke it on your code **x.py** by typing

```
$ pydb --threading x.py your_command_line_args_for_x
```

[13] No, it's not just for Microsoft Windows machines, in spite of the name.

Chapter 6

Python Iterators and Generators

Here is where Python begins to acquire some abstraction. Though I generally am not a fan of having a lot of abstraction, the constructs here are both elegant and useful. In the case of generators in particular, it's more than just abstraction; it actually enables us to do some things that essentially could not otherwise be done.

6.1 Iterators

6.1.1 What Are Iterators? Why Use Them?

Let's start with an example we know from Chapter 2. Say we open a file and assign the result to **f**, e.g.

```
f = open('x')
```

Suppose we wish to print out the lengths of the lines of the file.

```
for l in f.readlines():
   print len(l)
```

But the same result can be obtained with

```
for l in f:
   print len(l)
```

The second method has two advantages:

(a) it's simpler and more elegant

(b) a line of the file is not read until it is actually needed

For point (b), note that typically this becomes a major issue if we are reading a huge file, one that either might not fit into even virtual memory space, or is big enough to cause performance issues, e.g. excessive paging.

Point (a) becomes even clearer if we take the functional programming approach. The code

```
print map(len,f.readlines())
```

is not as nice as

```
print map(len,f)
```

Point (b) would be of major importance if the file were really large. The first method above would have the entire file in memory, very undesirable. Here we read just one line of the file at a time. Of course, we also could do this by calling **readline()** instead of **readlines()**, but not as simply and elegantly, i.e. we could not use **map()**.

In our second method above,

```
for l in f:
   print len(l)
```

f is serving as an **iterator**. Let's look at the concept more generally.

Recall that a Python **sequence** is roughly like an array in most languages, and takes on two forms—lists and tuples.[1] Sequence operations in Python are much more flexible than in a language like C or C++. One can have a function return a sequence; one can **slice** sequences; one can concatenate sequences; etc.

In this context, an **iterator** looks like a sequence when you use it, but with some major differences. With an iterator,

(a) you usually must write a function which actually constructs that sequence-like object

(b) an element of the "sequence" is not actually produced until you need it

(c) unlike real sequences, an iterator "sequence" can be (in concept) infinitely long

[1] Recall also that strings are tuples, but with extra properties.

6.1.2 Example: Fibonacci Numbers

For simplicity, let's start with everyone's favorite computer science example, Fibonacci numbers, as defined by the recursion,

$$f_n = \begin{cases} 1, & \text{if } n = 1, 2 \\ f_{n-1} + f_{n-2}, & \text{if } n > 2 \end{cases} \tag{6.1}$$

It's easy to write a loop to compute these numbers. But let's try it as an iterator:

```
# iterator example; uses Fibonacci numbers, so common in computer
# science examples:  f_n = f_{n-1} + f_{n-2}, with f_0 = f_1 = 1

class fibnum:
   def __init__(self):
      self.fn2 = 1  # "f_{n-2}"
      self.fn1 = 1  # "f_{n-1}"
   def next(self):  # next() is the heart of any iterator
      # note the use of the following tuple to not only save lines of
      # code but also to insure that only the old values of self.fn1 and
      # self.fn2 are used in assigning the new values
      (self.fn1,self.fn2,oldfn2) = (self.fn1+self.fn2,self.fn1,self.fn2)
      return oldfn2
   def __iter__(self):
      return self
```

Now here is how we would use the iterator, e.g. to loop with it:

```
from fib import *

def main():
   f = fibnum()
   for i in f:
      print i
      if i > 20: break

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

By including the method **__iter__()** in our **fibnum** class, we informed the Python interpreter that we wish to use this class as an iterator.

We also had to include the method **next()**, which as its name implies, is the mechanism by which the "sequence" is formed. This enabled us to simply place an instance of the class in the **for** loop above. Knowing that **f** is an iterator, the Python interpreter will repeatedly call **f.next()**, assigning the values returned by that function to **i**. When there are no items left in the iterator, a call to next produces the **StopIteration** exception.

It returns itself (in this common but not exhaustive form), which is used for iteration in **for** loops.

6.1.3 The iter() Function

Some Python structures already have built-in iterator capabilities. For example,

```
>>> dir([])
[’__add__’, ’__class__’, ’__contains__’, ’__delattr__’, ’__delitem__’,
’__delslice__’, ’__doc__’, ’__eq__’, ’__ge__’, ’__getattribute__’,
’__getitem__’, ’__getslice__’, ’__gt__’, ’__hash__’, ’__iadd__’,
’__imul__’, ’__init__’, ’__iter__’, ’__le__’, ’__len__’, ’__lt__’,
’__mul__’, ’__ne__’, ’__new__’, ’__reduce__’, ’__reduce_ex__’,
’__repr__’, ’__reversed__’, ’__rmul__’, ’__setattr__’, ’__setitem__’,
’__setslice__’, ’__str__’, ’append’, ’count’, ’extend’, ’index’,
’insert’, ’pop’, ’remove’, ’reverse’, ’sort’]
```

As you can see, the Python’s **list** class includes a member function **__iter__()**, so we can make an iterator out of it. Python provides the function **iter()** for this purpose, e.g.:

```
>>> i = iter(range(5))
>>> i.next()
0
>>> i.next()
1
```

Though the **next()** function didn’t show up in the listing above, it is in fact present:

```
>>> i.next
<method-wrapper ’next’ of listiterator object at 0xb765f52c>
```

You can now understand what is happening internally in this innocent-looking **for** loop:

```
for i in range(8): print i
```

Internally, it’s doing

```
itr = iter(range(8))
while True:
   try:
      i = itr.next()
      print i
   except:
      raise StopIteration
```

Of course it is doing the same thing for iterators that we construct from our own classes.

You can apply **iter()** to any sequence, e.g.:

```
>>> s = iter('UC Davis')
>>> s.next()
'U'
>>> s.next()
'C'
```

and even to dictionaries, where the iterator returns the keys.

You can also use **iter()** on any callable object.

6.1.4 Applications to Situations with an Indefinite Number of Iterations

As stated above, the iterator approach often makes for more elegant code. But again, note the importance of not having to compute the entire sequence at once. Having the entire sequence in memory would waste memory and would be impossible in the case of an infinite sequence, as we have here. Our **for** loop above is iterating through an infinite number of iterations—and would do so, if we didn't stop it as we did. But each element of the "sequence" is computed only at the time it is needed.

Moreover, this may be necessary, not just a luxury, even in the finite case. Consider this simple client/server pair in the next section.

6.1.4.1 Client/Server Example

```
# x.py, server

import socket,sys,os

def main():
   ls = socket.socket(socket.AF_INET,socket.SOCK_STREAM);
   port = int(sys.argv[1])
   ls.bind(('', port))
   ls.listen(1)
   (conn, addr) = ls.accept()
   while 1:
      l = raw_input()
      conn.send(l)

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

```
# w.py, client

import socket,sys

def main():
   s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
   host = sys.argv[1]
   port = int(sys.argv[2])
   s.connect((host,port))
   flo = s.makefile(’r’,0)  # file-like object, thus iterable
   for l in flo:
      print l

if __name__ == ’__main__’:
   main()
```

The server reads lines from the keyboard. It sends each line to the client as soon as the line is typed. However, if on the client side we had written

```
   for l in flo.readlines:
      print l
```

instead of

```
   for l in flo:
      print l
```

then the client would print out nothing until all of **flo** is received, meaning that the user on the server end typed ctrl-d to end the keyboard input, thus closing the connection.

Rather than being thought of as an “accident,” one can use exceptions as an elegant way to end a loop involving an iterator, using the built-in exception type **StopIteration**. For example:

```
class fibnum20:
   def __init__(self):
      self.fn2 = 1  # "f_{n-2}"
      self.fn1 = 1  # "f_{n-1}"
   def next(self):
      (self.fn1,self.fn2,oldfn2) = (self.fn1+self.fn2,self.fn1,self.fn2)
      if oldfn2 > 20: raise StopIteration
      return oldfn2
   def __iter__(self):
      return self
```

```
>>> from fib20 import *
>>> g = fibnum20()
>>> for i in g:
...    i
...
1
1
2
3
5
8
13
```

What happens is that iterating in the loop

```
>>> for i in g:
```

catches the exception **StopIteration**, which makes the looping terminate, and our "sequence" is finite.

6.1.4.2 "Circular" Array Example

Here's an example of using iterators to make a "circular" array. In Chapter 4, we needed to continually cycle through a list **cs** of client sockets:[2]

```
while (1):
   # get next client, with effect of a circular queue
   clnt = cs.pop(0)
   ...
   cs.append(clnt)
   ...
```

Here's how to make an iterator out of it:[3]

```
# circular queue

class cq:  # the argument q is a list
   def __init__(self,q):
      self.q = q
   def __iter__(self):
      return self
   def next(self):
      self.q = self.q[1:] + [self.q[0]]
      return self.q[-1]
```

Let's test it:

```
>>> from cq import *
>>> x = cq([1,2,3])
>>> x.next()
1
>>> x.next()
2
>>> x.next()
3
>>> x.next()
1
>>> x.next()
2
```

[2] I am slightly modifying it here, by assuming a constant number of clients.

[3] I've also made the code more compact, independent of the change to an iterator.

With this, our **while** loop in the network program above would look like this:

```
cit = cq(cs)
for clnt in cit:
   # code using clnt
   ...
```

The code would iterate indefinitely.

Of course, we had to do a bit of work to set this up. But now that we have, we can reuse this code in lots of different applications in the future, and the nice, clear form such as that above,

```
for clnt in cs:
```

adds to ease of programming and readability of code.

6.1.5 Overriding the next() Function: File Subclass Example

As mentioned, one can use a file as an iterator. The **file** class does have member functions **__iter__()** and **next()**. The latter is what is called by **readline()** and **readlines()**, and can be overriden.

Suppose we often deal with text files whose only elements are ’0’ and ’1’, with the same number of elements per line. We can form a class **file01** as a subclass of **file**, and add some error checking:

```
import sys

class file01(file):
   def __init__(self,name,mode,ni):
      file.__init__(self,name,mode)
      self.ni = ni  # number of items per line
   def next(self):
      line = file.next(self)
      items = line.split()
      if len(items) != self.ni:
         print ’wrong number of items’
         print line
         raise StopIteration
      for itm in items:
         if itm != ’1’ and itm != ’0’:
            print ’non-0/1 item:’, itm
            raise StopIteration
      return line

def main():
   f = file01(sys.argv[1],’r’,int(sys.argv[2]))
   for l in f: print l[:-1]

if __name__ == ’__main__’: main()
```

Here are some simple examples:

```
% cat u
1 0 1
0 1 1
% python file01.py u 3
1 0 1
0 1 1
% python file01.py u 2
wrong number of items
1 0 1

% cat v
1 1 b
1 0 1
% python file01.py v 3
non-0/1 item: b
```

One point to note here that you can open any file (not just of this new special kind) by simply creating an instance of the **file** class. For example, this would open a file **x** for reading and print its lines:

```
f = file('x','r')
for l in f: print l
```

I've used that fact in **main()** above.

Note also that in overriding **file.next()**, we still need to call it, in order to to read a line:

```
line = file.next(self)
```

6.1.6 Iterator Functions

6.1.6.1 General Functions

You can also make a real sequence out of an iterator's "output" by using the **list()** function, though you of course do have to make sure the iterator produces finite output. For example:

```
>>> from fib20 import *
>>> g = fibnum20()
>>> g
<fib20.fibnum20 instance at 0xb7e6c50c>
>>> list(g)
[1, 1, 2, 3, 5, 8, 13]
```

The functions **sum()**, **max()** and **min()** are built-ins for iterators, e.g.

```
>>> from fib20 import *
>>> g = fibnum20()
>>> sum(g)
33
```

6.1.6.2 Example: Word Fetcher

We have a text file, and wish to fetch words in the file one at a time.

```
class wordfetch:
   def __init__(self,fl):
      self.fl = fl
      # words remaining in current line
      self.words = []
   def __iter__(self): return self
   def next(self):
      if self.words == []:
         line = self.fl.readline()
         # check for end-of-file
         if line == '': raise StopIteration
         # remove end-of-line char
         line = line[:-1]
         self.words = line.split()
      firstword = self.words[0]
      self.words = self.words[1:]
      return firstword

def main():
   f = wordfetch(open('x'))
   for word in f: print word
```

6.1.6.3 The `itertools` Module

Here you can really treat an infinite iterator like a “sequence,” using various tools in this module.

For instance, **iterators.islice()** is handy:

```
>>> from itertools import *
>>> g = fibnum()
>>> list(islice(g,6))  # slice out the first 6 elements
[1, 1, 2, 3, 5, 8]
```

The general form of **islice()** is

```
itertools.islice(iteratorname, [start], stop, [step])
```

Here we get elements *start*, *start* + **step**, and so on, but ending before element *stop*.

For instance:

```
>>> list(islice(g,3,9,2))
[3, 8, 21]
```

There are also analogs of the **map()** and **filter()** functions which operate on real sequences. The call

```
itertools.imap(f, iter1, iter2, ...)
```

returns the stream **f(iter1[0],iter2[0],...)**, which one can then apply **list()** to.

The call

```
itertools.ifilter(boolean expression, iter)
```

applies the boolean test to each elment of the iterator stream.

6.2 Generators

6.2.1 General Structures

Generators are entities which generate iterators! Hence the name.

There are two main goals:

- Generators provide a cleaner, clearer, more convenient ways to create iterators.
- Generators can be used to create **coroutines**, which are quite useful in certain applications, notably discrete-event simulation.

Roughly speaking, a generator is a function that we wish to call repeatedly, but which is unlike an ordinary function in that successive calls to a generator function don't start execution at the beginning of the function.

Instead, the current call to a generator function will resume execution right after the spot in the code at which the last call exited, i.e. we "pick up where we left off."

In other words, generators have some notion of "state," where the state consists of the line following the last one executed in the previous call, and the values of local variables and arguments.

The way this occurs is as follows. One calls the generator itself just once. That returns an iterator. This is a real iterator, with **__iter()__** and **next()** methods. The latter is essentially the function which implements our "pick up where we left off" goal. We can either call **next()** directly, or use the iterator in a loop.

Note that difference in approach:

- In the case of iterators, a class is recognized by the Python interpreter as an iterator by the presence of the **__iter()__** and **next()** methods.
- By contrast, with a generator we don't even need to set up a class. We simply write a plain function, with its only distinguishing feature for recognition by the Python interpreter being that we use **yield** instead of **return**.

Note, though, that **yield** and **return** work quite differently from each other. When a **yield** is executed, the Python interpreter records the line number of that statement (there may be several **yield** lines within the same generator), and the values of the local variables and arguments. Then, the next time we call the **.next()** function of this same iterator that we generated from the generator function, the function will resume execution at the line following the **yield**. Depending on your application, the net effect may be very different from the iterators we've seen so far, in a much more flexible way.

Here are the key points:

- A **yield** causes an exit from the function, but the next time the function is called, we start "where we left off," i.e. at the line following the **yield** rather than at the beginning of the function.
- All the values of the local variables which existed at the time of the **yield** action are now still intact when we resume.
- There may be several **yield** lines in the same generator.
- We can also have **return** statements, but execution of any such statement will result in a **StopIteration** exception being raised if the **next()** method is called again.
- The **yield** operation has one operand (or none), which is the return value. That one operand can be a tuple, though. As usual, if there is no ambiguity, you do not have to enclose the tuple in parentheses.

Read the following example carefully, keeping all of the above points in mind:

```
# yieldex.py example of yield, return in generator functions

def gy():
   x = 2
   y = 3
   yield x,y,x+y
   z = 12
   yield z/x
   print z/y
   return

def main():
   g = gy()
   print g.next()  # prints x,y,x+y
   print g.next()  # prints z/x
   print g.next()

if __name__ == ’__main__’:
   main()
```

```
% python yieldex.py
(2, 3, 5)
6
4
Traceback (most recent call last):
  File "yieldex.py", line 19, in ?
    main()
  File "yieldex.py", line 16, in main
    print g.next()
StopIteration
```

Note that execution of the actual code in the function **gy()**, i.e. the lines

```
x = 2
...
```

does not execute until the first **g.next()** is executed.

6.2.2 Example: Fibonacci Numbers

As another simple illustration, let’s look at the good ol’ Fibonacci numbers again:

```
# fibg,py, generator example; Fibonacci numbers
# f_n = f_{n-1} + f_{n-2}

def fib():
   fn2 = 1  # "f_{n-2}"
   fn1 = 1  # "f_{n-1}"
   while True:
      (fn1,fn2,oldfn2) = (fn1+fn2,fn1,fn2)
      yield oldfn2
```

```
>>> from fibg import *
>>> g = fib()
>>> g.next()
1
>>> g.next()
1
>>> g.next()
2
>>> g.next()
3
>>> g.next()
5
>>> g.next()
8
```

Note that the generator's trait of resuming execution "where we left off" is quite necessary here. We certainly don't want to execute

```
fn2 = 1
```

again, for instance. Indeed, a key point is that the local variables **fn1** and **fn2** retain their values between calls. This is what allowed us to get away with using just a function instead of a class. This is simpler and cleaner than the class-based approach. For instance, in the code here we refer to **fn1** instead of **self.fn1** as we did in our class-based version in Section 6.1.2. In more complicated functions, all these simplifications would add up to a major improvement in readability.

This property of retaining locals between calls is like that of locals declared as **static** in C. Note, though, that in Python we might set up several instances of a given generator, each instance maintaining different values for the locals. To do this in C, we need to have arrays of the locals, indexed by the instance number. It would be easier in C++, by using instance variables in a class.

6.2.3 Example: Word Fetcher

The following is a producer/consumer example. The producer, **getword()**, gets words from a text file, feeding them one at a time to the consumer (in this case **main()**).[4] In the test here, the consumer is **testgw.py**.

```
# getword.py

# the function getword() reads from the text file fl, returning one word
# at a time; will not return a word until an entire line has been read

def getword(fl):
   for line in fl:
      for word in line.split():
         yield word
   return
```

[4] I thank C. Osterwisch for this much improved version of the code I had here originally.

```
# testgw.py, test of getword; counts words and computes average length
# of words

# usage:  python testgw.py [filename]
# (stdin) is assumed if no file is specified)

from getword import *

def main():
   import sys
   # determine which file we’ll evaluate
   try:
      f = open(sys.argv[1])
   except:
      f = sys.stdin
   # generate the iterator
   w = getword(f)
   wcount = 0
   wltot = 0
   for wrd in w:
      wcount += 1
      wltot += len(wrd)
   print "%d words, average length %f" % (wcount,wltot/float(wcount))

if __name__ == ’__main__’:
   main()
```

Notice how much simpler the generator version is here than the iterator version back in Section 6.1.6.2. A great example of the power of generators!

6.2.4 Mutiple Iterators from the Same Generator

Note our phrasing earlier (emphasis added):

> ...the next time this generator function is called *with this same iterator*, the function will resume execution at the line following the **yield**

Suppose for instance that in the above word count example we have two sorted text files, one word per line, and we wish to merge them into a combined sorted file. We could use our **getword()** function above, setting up two iterators, one for each file. Note that we might reach the end of one file before the other. We would then continue with the other file by itself. To deal with this, we would have to test for the **StopIteration** exception to sense when we’ve come to the end of a file.

6.2.5 The os.path.walk() Function

The function **os.path.walk()** in Chapter 2 is a generator.

6.2.6 Don't Put yield in a Subfunction

If you have a generator **g()**, and it in turn calls a function **h()**, don't put a **yield** statement in the latter, as the Python interpreter won't know how to deal with it.

6.2.7 Coroutines

So far, our presentation on generators has concentrated on their ability to create iterators. Another usage for generators is the construction of **coroutines**.

This is a general computer science term, not restricted to Python, that refers to subroutines that alternate in execution. Subroutine A will run for a while, then subroutine B will run for a while, then A again, and so on. Each time a subroutine runs, it will resume execution right where it left off before. That behavior of course is just like Python generators, which is why one can use Python generators for that purpose.

Basically coroutines are threads, but of the nonpreemptive type. In other words, a coroutine will continue executing until it voluntarily relinquishes the CPU. (Of course, this doesn't count timesharing with other unrelated programs. We are only discussing flow of control among the threads of one program.) In "ordinary" threads, the timing of the passing of control from one thread to another is to various degrees random.

The major advantage of using nonpreemptive threads is that you do not need locks, due to the fact that you are guaranteed that a thread runs until it itself relinquishes control of the CPU. This makes your code a lot simpler and cleaner, and much easier to debug. (The randomness alone makes ordinary threads really tough to debug.)

The disadvantage of nonpreemptive threads is precisely its advantage: Because only one thread runs at a time, one cannot use nonpreemptive threads for parallel computation on a multiprocessor machine.

Some major applications of nonpreemptive threads are:

- servers
- GUI programs
- discrete-event simulation

In this section, we will see examples of coroutines, in SimPy, a well-known Python discrete-event simulation library written by Klaus Muller and Tony Vignaux.

6.2.7.1 The SimPy Discrete Event Simulation Library

In discrete event simulation (DES), we are modeling discontinuous changes in the system state. We may be simulating a queuing system, for example, and since the number of jobs in the queue is an integer, the number will be incremented by an integer value, typically 1 or -1.[5] By contrast, if we are modeling a weather system, variables such as temperature change continuously.

The goal of DES is typically to ask “What if?” types of questions. An HMO might have nurses staffing advice phone lines, so one might ask, how many nurses do we need in order to keep the mean waiting time of patients below some desired value?

SimPy is a widely used open-source Python library for DES. Following is an example of its use. The application is an analysis of a pair of machines that break down at random times, and need random time for repair. See the comments at the top of the file for details, including the quantities of interest to be computed.

```
#!/usr/bin/env python

# MachRep.py

# SimPy example:  Two machines, but sometimes break down.  Up time is
# exponentially distributed with mean 1.0, and repair time is
# exponentially distributed with mean 0.5.  In this example, there is
# only one repairperson, so the two machines cannot be repaired
# simultaneously if they are down at the same time.

# In addition to finding the long-run proportion of up time, let’s also
# find the long-run proportion of the time that a given machine does not
# have immediate access to the repairperson when the machine breaks
# down.  Output values should be about 0.6 and 0.67.

from SimPy.Simulation import *
from random import Random,expovariate,uniform

class G:  # globals
   Rnd = Random(12345)
   # create the repairperson
   RepairPerson = Resource(1)

class MachineClass(Process):
   TotalUpTime = 0.0  # total up time for all machines
   NRep = 0  # number of times the machines have broken down
   NImmedRep = 0  # number of breakdowns in which the machine
                  # started repair service right away
   UpRate = 1/1.0  # breakdown rate
   RepairRate = 1/0.5  # repair rate
   # the following two variables are not actually used, but are useful
   # for debugging purposes
   NextID = 0  # next available ID number for MachineClass objects
   NUp = 0  # number of machines currently up
   def __init__(self):
```

[5] **Batch queues** may take several jobs at a time, but the increment is still integer-valued.

```
      Process.__init__(self)
      self.StartUpTime = 0.0  # time the current up period stated
      self.ID = MachineClass.NextID   # ID for this MachineClass object
      MachineClass.NextID += 1
      MachineClass.NUp += 1  # machines start in the up mode
   def Run(self):
      while 1:
         self.StartUpTime = now()
         # simulate up time for this machine
         yield hold,self,G.Rnd.expovariate(MachineClass.UpRate)
         # a breakdown has occurred on this machine
         MachineClass.TotalUpTime += now() - self.StartUpTime
         # update number of breakdowns
         MachineClass.NRep += 1
         # check whether we get repair service immediately; how many
         # repairpersons (1 or 0 here) are available?
         if G.RepairPerson.n == 1:
            MachineClass.NImmedRep += 1
         # need to request, and possibly queue for, the repairperson;
         # we’ll basically come back right after if the repairperson is
         # free
         yield request,self,G.RepairPerson
         # OK, we’ve obtained access to the repairperson; now
         # hold for repair time
         yield hold,self,G.Rnd.expovariate(MachineClass.RepairRate)
         # release the repairperson
         yield release,self,G.RepairPerson

def main():
   initialize()
   # set up the two machine processes
   for I in range(2):
      M = MachineClass()
      activate(M,M.Run())
   MaxSimtime = 10000.0
   simulate(until=MaxSimtime)
   print ’proportion of up time:’, MachineClass.TotalUpTime/(2*MaxSimtime)
   print ’proportion of times repair was immediate:’, \
      float(MachineClass.NImmedRep)/MachineClass.NRep

if __name__ == ’__main__’:  main()
```

Again, make sure to read the comments in the first few lines of the code to see what kind of system this program is modeling before going further.

Now, let’s see the details.

SimPy’s thread class is **Process**. The application programmer writes one or more subclasses of this one to serve as SimPy thread classes. Similar to the case for the **threading** class, the subclasses of **Process** must include a method **Run()**, which describes the actions of the SimPy thread.

The SimPy method **activate()** is used to add a SimPy thread to the run list. Remember, though, that the **Run()** methods are generators. So, activation consists of calling **Run()**, producing an iterator. So for instance, the line from the above code,

```
activate(M,M.Run())
```

has a *call* to **M.Run** as an argument, rather than the function itself being the argument. In other words, the argument is the iterator. More on this below.

The main new ingredient here is the notion of simulated time. The current simulated time is stored in the variable **Simulation._t**. Each time an event is created, via execution of a statement like

```
yield hold, self, holdtime
```

Recall that **yield** can have one argument, which in this case is the tuple **(hold, self, holdtime)**. (Remember, parentheses are optional in tuples.)

The value **hold** is a constant within SimPy, and it signifies, "hold for the given amount of simulated time before continuing. SimPy schedules the event to occur **holdtime** time units from now, i.e. at time **Simulation_t+holdtime**. What I mean by "schedule" here is that SimPy maintains an internal data structure which stores all future events, ordered by their occurrence times. Let's call this the scheduled events structure, SES. Note that the elements in SES are SimPy threads, i.e. instances of the class **Process**. A new event will be inserted into the SES at the proper place in terms of time ordering.

The main loop in SimPy repeatedly cycles through the following:

- Remove the earliest event, say **v**, from SES.
- If the occurrence time of **v** is past the simulation limit, exit the loop. Otherwise, advance the simulated time clock **Simulation._t** to the occurrence time of **v**.
- Call the **.next()** method of iterator for **v**, i.e. the iterator for the **Run()** generator of that thread, thus causing the code in **Run()** to resume at the point following its last executed **yield**.
- Act on the result of the next **yield** that **Run()** hits (could be the same one, in a loop), e.g. handle a **hold** command. (Note that **hold** is simply a constant in the SimPy internals, as are the other SimPy actions such as **request** below.)

Note that this is similar to what an ordinary threads manager does, but differs due to the time element. In ordinary threads programming, there is no predicting as to which thread will run next. Here, we know which one it will be.[6]

[6] This statement is true as long as there are no tied event times, which in most applications do not occur. In most applications, the main hold times are exponentially distributed, or have some other continuous distribution. Such distributions imply that the probability of a tie is 0, and though that is not strictly true given the finite word size of a machine, in practice one usually doesn't worry about ties.

In simulation programming, we often need to have one entity wait for some event to occur. In our example here, if one machine goes down while the other is being repaired, the newly-broken machine will need to wait for the repairperson to become available. Clearly this is like the condition variables construct in most threads packages.

Specifically, SimPy includes a **Resource** class. In our case here, the resource is the repairperson. When a line like

```
yield request,self,G.RepairPerson
```

is executed, SimPy will look at the internal data structure in which SimPy stores the queue for the repairperson. If it is empty, the thread that made the request will acquire access to the repairperson, and control will return to the statement following **yield request**. If there are threads in the queue (here, there would be at most one), then the thread which made the request will be added to the queue. Later, when a statement like

```
yield release,self,G.RepairPerson
```

is executed by the thread currently accessing the repairperson, SimPy will check its queue, and if the queue is nonempty, SimPy will remove the first thread from the queue, and have it resume execution where it left off.[7]

Since the simulated time variable **Simulation._t** is in a separate module, we cannot access it directly. Thus SimPy includes a "getter" function, **now()**, which returns the value of **Simulation._t**.

Most discrete event simulation applications are stochastic in nature, such as we see here with the random up and repair times for the machines. Thus most SimPy programs import the Python **random** module, as in this example.

For more information on SimPy, see the files **DESimIntro.pdf**, **AdvancedSimPy.pdf** and **SimPyInternals.pdf** in `http://heather.cs.ucdavis.edu/~matloff/156/PLN`.

[7] This will not happen immediately. The thread that triggered the release of the resource will be allowed to resume execution right after the **yield release** statement. But SimPy will place an artificial event in the SES, with event time equal to the current time, i.e. the time at which the release occurred. So, as soon as the current thread finishes, the awakened thread will get a chance to run again, most importantly, at the same simulated time as before.

Chapter 7

Python Curses Programming

As one of my students once put it, **curses** is a "text-based GUI." No, this is not an oxymoron. What he meant was that this kind of programming enables one to have a better visual view of a situation, while still working in a purely text-based context.

7.1 Function

Many widely-used programs need to make use of a terminal's cursor-movement capabilities. A familiar example is **vi**; most of its commands make use of such capabilities. For example, hitting the **j** key while in **vi** will make the cursor move down line. Typing **dd** will result in the current line being erased, the lines below it moving up one line each, and the lines above it remaining unchanged. There are similar issues in the programming of **emacs**, etc.

The `curses` library gives the programmer functions (*APIs, Application Program Interfaces*) to call to take such actions.

Since the operations available under `curses` are rather primitive—cursor movement, text insertion, etc.—libraries have been developed on top of `curses` to do more advanced operations such as pull-down menus, radio buttons and so on. More on this in the Python context later.

7.2 History

Historically, a problem with all this was that different terminals had different ways in which to specify a given type of cursor motion. For example, if a program needed to make the cursor move up one line on a VT100 terminal, the program would need to send the characters Escape, [, and A:

```
printf("%c%c%c",27,'[','A');
```

(the character code for the Escape key is 27). But for a Televideo 920C terminal, the program would have to send the ctrl-K character, which has code 11:

```
printf("%c",11);
```

Clearly, the authors of programs like **vi** would go crazy trying to write different versions for every terminal, and worse yet, anyone else writing a program which needed cursor movement would have to "re-invent the wheel," i.e. do the same work that the **vi**-writers did, a big waste of time.

That is why the `curses` library was developed. The goal was to alleviate authors of cursor-oriented programs like **vi** of the need to write different code for different terminals. The programs would make calls to the API library, and the library would sort out what to do for the given terminal type.

The library would know which type of terminal you were using, via the environment variable **TERM**. The library would look up your terminal type in its terminal database (the file **/etc/termcap**). When you, the programmer, would call the `curses` API to, say, move the cursor up one line, the API would determine which character sequence was needed to make this happen.

For example, if your program wanted to clear the screen, it would not directly use any character sequences like those above. Instead, it would simply make the call

```
clear();
```

and `curses` would do the work on the program's behalf.

7.3 Relevance Today

Many dazzling GUI programs are popular today. But although the GUI programs may provide more "eye candy," they can take a long time to load into memory, and they occupy large amounts of territory on your screen. So, `curses` programs such as **vi** and **emacs** are still in wide usage.

Interestingly, even some of those classical `curses` programs have also become somewhat GUI-ish. For instance **vim**, the most popular version of **vi** (it's the version which comes with most Linux distributions, for example), can be run in **gvim** mode. There, in addition to having the standard keyboard-based operations, one can also use the mouse. One can move the cursor to another location by clicking the mouse at that point; one can use the mouse to select blocks of text for deletion or movement; etc. There are icons at the top of the editing window, for operations like Find, Make, etc.

7.4 Examples of Python Curses Programs

7.4.1 Useless Example

The program below, **crs.py**, does not do anything useful. Its sole purpose is to introduce some of the `curses` APIs.

There are lots of comments in the code. Read them carefully, first by reading the introduction at the top of the file, and then going to the bottom of the file to read **main()**. After reading the latter, read the other functions.

```
# crs.py; simple illustration of curses library, consisting of a very
# unexciting "game"; keeps drawing the user’s input characters into a
# box, filling one column at a time, from top to bottom, left to right,
# returning to top left square when reach bottom right square

# the bottom row of the box is displayed in another color

# usage:  python crs.py boxsize

# how to play the "game":  keep typing characters, until wish to stop,
# which you do by hitting the q key

import curses, sys, traceback

# global variables
class gb:
   boxrows = int(sys.argv[1])  # number of rows in the box
   boxcols = boxrows  # number of columns in the box
   scrn = None  # will point to window object
   row = None  # current row position
   col = None  # current column position

def draw(chr):
   # paint chr at current position, overwriting what was there; if it’s
   # the last row, also change colors; if instead of color we had
   # wanted, say, reverse video, we would specify curses.A_REVERSE instead of
   # curses.color_pair(1)
   if gb.row == gb.boxrows-1:
      gb.scrn.addch(gb.row,gb.col,chr,curses.color_pair(1))
   else:
      gb.scrn.addch(gb.row,gb.col,chr)
   # implement the change
   gb.scrn.refresh()
   # move down one row
   gb.row += 1
   # if at bottom, go to top of next column
   if gb.row == gb.boxrows:
      gb.row = 0
      gb.col += 1
      # if in last column, go back to first column
      if gb.col == gb.boxcols: gb.col = 0

```

```
# this code is vital; without this code, your terminal would be unusable
# after the program exits
def restorescreen():
   # restore "normal"--i.e. wait until hit Enter--keyboard mode
   curses.nocbreak()
   # restore keystroke echoing
   curses.echo()
   # required cleanup call
   curses.endwin()

def main():
   # first we must create a window object; it will fill the whole screen
   gb.scrn = curses.initscr()
   # turn off keystroke echo
   curses.noecho()
   # keystrokes are honored immediately, rather than waiting for the
   # user to hit Enter
   curses.cbreak()
   # start color display (if it exists; could check with has_colors())
   curses.start_color()
   # set up a foreground/background color pair (can do many)
   curses.init_pair(1,curses.COLOR_RED,curses.COLOR_WHITE)
   # clear screen
   gb.scrn.clear()
   # set current position to upper-left corner; note that these are our
   # own records of position, not Curses’
   gb.row = 0
   gb.col = 0
   # implement the actions done so far (just the clear())
   gb.scrn.refresh()
   # now play the "game"
   while True:
      # read character from keyboard
      c = gb.scrn.getch()
      # was returned as an integer (ASCII); make it a character
      c = chr(c)
      # quit?
      if c == ’q’: break
      # draw the character
      draw(c)
   # restore original settings
   restorescreen()

if __name__ ==’__main__’:
   # in case of execution error, have a smooth recovery and clear
   # display of error message (nice example of Python exception
   # handling); it is recommended that you use this format for all of
   # your Python curses programs; you can automate all this (and more)
   # by using the built-in function curses.wrapper(), but we’ve shown
   # it done "by hand" here to illustrate the issues involved
   try:
      main()
   except:
      restorescreen()
      # print error message re exception
      traceback.print_exc()
```

7.4.2 Useful Example

The following program allows the user to continuously monitor processes on a Unix system. Although some more features could be added to make it more useful, it is a real working utility.

```
# psax.py; illustration of curses library

# runs the shell command ’ps ax’ and saves the last lines of its output,
# as many as the window will fit; allows the user to move up and down
# within the window, killing those processes

# run line:  python psax.py

# user commands:

#     ’u’:  move highlight up a line
#     ’d’:  move highlight down a line
#     ’k’:  kill process in currently highlighted line
#     ’r’:  re-run ’ps ax’ for update
#     ’q’:  quit

# possible extensions:  allowing scrolling, so that the user could go
# through all the ’ps ax’ output; allow wraparound for long lines; ask
# user to confirm before killing a process

import curses, os, sys, traceback

# global variables
class gb:
   scrn = None  # will point to Curses window object
   cmdoutlines = []  # output of ’ps ax’ (including the lines we don’t
                     # use, for possible future extension)
   winrow = None  # current row position in screen
   startrow = None  # index of first row in cmdoutlines to be displayed

def runpsax():
   p = os.popen(’ps ax’,’r’)
   gb.cmdoutlines = []
   row = 0
   for ln in p:
      # don’t allow line wraparound, so truncate long lines
      ln = ln[:curses.COLS]
      # remove EOLN if it is still there
      if ln[-1] == ’\n’: ln = ln[:-1]
      gb.cmdoutlines.append(ln)
   p.close()

# display last part of command output (as much as fits in screen)
def showlastpart():
   # clear screen
   gb.scrn.clear()
   # prepare to paint the (last part of the) ’ps ax’ output on the screen
   gb.winrow = 0
   ncmdlines = len(gb.cmdoutlines)
   # two cases, depending on whether there is more output than screen rows
   if ncmdlines <= curses.LINES:
      gb.startrow = 0
      nwinlines = ncmdlines
   else:
      gb.startrow = ncmdlines - curses.LINES - 1
      nwinlines = curses.LINES
   lastrow = gb.startrow + nwinlines - 1
   # now paint the rows
   for ln in gb.cmdoutlines[gb.startrow:lastrow]:
      gb.scrn.addstr(gb.winrow,0,ln)
      gb.winrow += 1
   # last line highlighted
   gb.scrn.addstr(gb.winrow,0,gb.cmdoutlines[lastrow],curses.A_BOLD)
   gb.scrn.refresh()

# move highlight up/down one line
def updown(inc):
   tmp = gb.winrow + inc
   # ignore attempts to go off the edge of the screen
   if tmp >= 0 and tmp < curses.LINES:
      # unhighlight the current line by rewriting it in default attributes
      gb.scrn.addstr(gb.winrow,0,gb.cmdoutlines[gb.startrow+gb.winrow])
      # highlight the previous/next line
      gb.winrow = tmp
      ln = gb.cmdoutlines[gb.startrow+gb.winrow]
      gb.scrn.addstr(gb.winrow,0,ln,curses.A_BOLD)
      gb.scrn.refresh()

# kill the highlighted process
def kill():
   ln = gb.cmdoutlines[gb.startrow+gb.winrow]
   pid = int(ln.split()[0])
   os.kill(pid,9)

# run/re-run ’ps ax’
def rerun():
   runpsax()
   showlastpart()

def main():
   # window setup
   gb.scrn = curses.initscr()
   curses.noecho()
   curses.cbreak()
   # run ’ps ax’ and process the output
   gb.psax = runpsax()
   # display in the window
   showlastpart()
   # user command loop
   while True:
      # get user command
      c = gb.scrn.getch()
      c = chr(c)
      if c == ’u’: updown(-1)
      elif c == ’d’: updown(1)
      elif c == ’r’: rerun()
      elif c == ’k’: kill()
      else: break
   restorescreen()
```

```

def restorescreen():
   curses.nocbreak()
   curses.echo()
   curses.endwin()

if __name__ =='__main__':
   try:
      main()
   except:
      restorescreen()
      # print error message re exception
      traceback.print_exc()
```

Try running it yourself!

7.4.3 A Few Other Short Examples

See the directory **Demo/curses** in the Python source code distribution

7.5 What Else Can Curses Do?

7.5.1 Curses by Itself

The examples above just barely scratch the surface. We won't show further examples here, but to illustrate other operations, think about what **vi**, a `curses`-based program, must do in response to various user commands, such as the following (suppose our window object is **scrn**):

- **k** command, to move the cursor up one line: might call **scrn.move(r,c)**, which moves the `curses` cursor to the specified row and column[1]

- **dd** command, to delete a line: might call **scrn.deleteln()**, which causes the current row to be deleted and makes the rows below move up[2]

- ∼ command, to change case of the character currently under the cursor: might call **scrn.inch()**, which returns the character currently under the cursor, and then call **scrn.addch()** to put in the character of opposite case

- **:sp** command (**vim**), to split the current **vi** window into two subwindows: might call **curses.newwin()**

[1] But if the movement causes a scrolling operation, other `curses` functions will need to be called too.

[2] But again, things would be more complicated if that caused scrolling.

- mouse operations in **gvim**: call functions such as **curses.mousemask()**, **curses.getmouse()**, etc.

You can imagine similar calls in the source code for **emacs**, etc.

7.6 Libraries Built on Top of Curses

The operations provided by `curses` are rather primitive. Say for example you wish to have a menu subwindow in your application. You could do this directly with `curses`, using its primitive operations, but it would be nice to have high-level libraries for this.

A number of such libraries have been developed. One you may wish to consider is **urwid**, `http://excess.org/urwid/`.

7.7 If Your Terminal Window Gets Messed Up

Curses programs by nature disable the "normal" behavior you expect of a terminal window. If your program has a bug that makes it exit prematurely, that behavior will not automatically be re-enabled.

In our first example above, you saw how we could include to do the re-enabling even if the program crashes. This of course is what is recommended. Butif you don't do it, you can re-enable your window capabilities by hitting ctrl-j then typing "reset", then hitting ctrl-j again.

7.8 Debugging

The open source debugging tools I usually use for Python—PDB, DDD—but neither can be used for debugging Python `curses` application. For the PDB, the problem is that one's PDB commands and their outputs are on the same screen as the application program's display, a hopeless mess. This ought not be a problem in using DDD as an interface to PDB, since DDD does allow one to have a separate execution window. That works fine for `curses` programming in C/C++, but for some reason this can't be invoked for Python. Even the Eclipse IDE seems to have a problem in this regard.

However, the Winpdb debugger (`www.digitalpeers.com/pythondebugger/`),[3] solves this problem. Among other things, it can be used to debug threaded code, curses-based code and so on, which many debuggers can't. Winpdb is a GUI front end to the text-based RPDB2, which is in the same package. I have a tutorial on both at `http://heather.cs.ucdavis.edu/~matloff/winpdb.html`.

[3] No, it's not just for Microsoft Windows machines, in spite of the name.

Chapter 8

Python Debugging

One of the most undervalued aspects taught in programming courses is debugging. It's almost as if it's believed that wasting untold hours late at night is good for you!

Debugging is an art, but with good principles and good tools, you really can save yourself lots of those late night hours of frustration.

8.1 The Essence of Debugging: The Principle of Confirmation

Though debugging is an art rather than a science, there are some fundamental principles involved, which will be discussed first.

As Pete Salzman and I said in our book on debugging, *The Art of Debugging, with GDB, DDD and Eclipse* (No Starch Press, 2008), the following rule is the essence of debugging:

> **The Principle of Confirmation**
>
> Fixing a buggy program is a process of confirming, one by one, that the many things you *believe* to be true about the code actually *are* true. When you find that one of your assumptions is *not* true, you have found a clue to the location (if not the exact nature) of a bug.

Another way of saying this is,

> Surprises are good!

For example, say you have the following code:

```
x = y**2 + 3*g(z,2)
w = 28
if w+q > 0: u = 1
else: v = 10
```

Do you think the value of your variable **x** should be 3 after **x** is assigned? Confirm it! Do you think the "else" will be executed, not the "if," on that third line? Confirm it!

Eventually one of these assertions that you are so sure of will turn out to *not* confirm. Then you will have pinpointed the likely *location* of the error, thus enabling you to focus on the *nature* of the error.

8.2 Plan for This Chapter

Do NOT debug by simply adding and subtracting **print** statements. Use a debugging tool! If you are not a regular user of a debugging tool, then you are causing yourself unnecessary grief and wasted time; see my debugging slide show, at `http://heather.cs.ucdavis.edu/~matloff/debug.html`.

The remainder of this chapter will be devoted to various debugging tools. We'll start with Python's basic built-in debugger, PDB. It's pretty primitive, but it's the basis of some other tools, and I'll show you how to exploit its features to make it a pretty decent debugger.

Then there will be brief overviews of some other tools.

8.3 Python's Built-In Debugger, PDB

The built-in debugger for Python, PDB, is rather primitive, but it's very important to understand how it works, for two reasons:

- PDB is used indirectly by more sophisticated debugging tools. A good knowledge of PDB will enhance your ability to use those other tools.
- I will show you here how to increase PDB's usefulness even as a standalone debugger.

I will present PDB below in a sequence of increasingly-useful forms:

- The basic form.
- The basic form enhanced by strategic use of macros.
- The basic form in conjunction with the Emacs text editor.

8.3.1 The Basics

Nowadays PDB is an integral part of Python, rather than being invoked as a separate module. To debug a script **x.py**, type

```
% python -m pdb x.py
```

(If **x.py** had had command-line arguments, they would be placed after **x.py** on the command line.)

Once you are in PDB, set your first breakpoint, say at line 12:

```
b 12
```

You can make it conditional, e.g.

```
b 12, z > 5
```

Hit **c** ("continue"), which you will get you into **x.py** and then stop at the breakpoint. Then continue as usual, with the main operations being like those of GDB:

- **b** ("break") to set a breakpoint
- **tbreak** to set a one-time breakpoint
- **ignore** to specify that a certain breakpoint will be ignored the next k times, where k is specified in the command
- **l** ("list") to list some lines of source code
- **n** ("next") to step to the next line, not stopping in function code if the current line is a function call
- **s** ("subroutine") same as **n**, except that the function *is* entered in the case of a call
- **c** ("continue") to continue until the next break point
- **w** ("where") to get a stack report
- **u** ("up") to move up a level in the stack, e.g. to query a local variable there
- **d** ("down") to move down a level in the stack
- **r** ("return") continue execution until the current function returns

- **j** ("jump") to jump to another line *without* the intervening code being executed
- **h** ("help") to get (minimal) online help (e.g. **h b** to get help on the **b** command, and simply **h** to get a list of all commands); type **h pdb** to get a tutorial on PDB[1]
- **q** ("quit") to exit PDB

Upon entering PDB, you will get its `(Pdb)` prompt.

If you have a multi-file program, breakpoints can be specified in the form `module_name:line_number`. For instance, suppose your main module is **x.py**, and it imports **y.py**. You set a breakpoint at line 8 of the latter as follows:

```
(Pdb) b y:8
```

Note, though, that you can't do this until **y** has actually been imported by **x**.[2]

When you are running PDB, you are running Python in its interactive mode. Therefore, you can issue any Python command at the PDB prompt. You can set variables, call functions, etc. This can be highly useful.

For example, although PDB includes the **p** command for printing out the values of variables and expressions, it usually isn't necessary. To see why, recall that whenever you run Python in interactive mode, simply typing the name of a variable or expression will result in printing it out—exactly what **p** would have done, without typing the 'p'.

So, if **x.py** contains a variable **ww** and you run PDB, instead of typing

```
(Pdb) p ww
```

you can simply type

```
ww
```

and the value of **ww** will be printed to the screen.[3]

If your program has a complicated data structure, you could write a function to print to the screen all or part of that structure. Then, since PDB allows you to issue any Python command at the PDB prompt, you could simply call this function at that prompt, thus getting more sophisticated, application-specific printing.

[1] The tutorial is run through a pager. Hit the space bar to go to the next page, and the q key to quit.

[2] Note also that if the module is implemented in C, you of course will not be able to break there.

[3] However, if the name of the variable is the same as that of a PDB command (or its abbreviation), the latter will take precedence. If for instance you have a variable **n**, then typing **n** will result in PDB's **n[ext]** command being executed, rather than there being a printout of the value of the variable **n**. To get the latter, you would have to type **p n**.

After your program either finishes under PDB or runs into an execution error, you can re-run it without exiting PDB—important, since you don't want to lose your breakpoints—by simply hitting **c**. And yes, if you've changed your source code since then, the change will be reflected in PDB.[4]

If you give PDB a single-step command like **n** when you are on a Python line which does multiple operations, you will need to issue the **n** command multiple times (or set a temporary breakpoint to skip over this).

For example,

```
for i in range(10):
```

does two operations. It first calls **range()**, and then sets **i**, so you would have to issue **n** twice.

And how about this one?

```
y = [(y,x) for (x,y) in x]
```

If **x** has, say, 10 elements, then you would have to issue the `n` command 10 times! Here you would definitely want to set a temporary breakpoint to get around it.

8.3.2 Using PDB Macros

PDB's undeniably bare-bones nature can be remedied quite a bit by making good use of the **alias** command, which I strongly suggest. For example, type

```
alias c c;;l
```

This means that each time you hit **c** to continue, when you next stop at a breakpoint you automatically get a listing of the neighboring code. This will really do a lot to make up for PDB's lack of a GUI.

In fact, this is so important that you should put it in your PDB startup file, which in Linux is **$HOME/.pdbrc**.[5] That way the alias is always available. You could do the same for the **n** and **s** commands:

```
alias c c;;l
alias n n;;l
alias s s;;l
```

[4] PDB is, as seen above, just a Python program itself. When you restart, it will re-import your source code.

By the way, the reason your breakpoints are retained is that of course they are variables in PDB. Specifically, they are stored in member variable named **breaks** in the the **Pdb** class in **pdb.py**. That variable is set up as a dictionary, with the keys being names of your **.py** source files, and the items being the lists of breakpoints.

[5] Python will also check for such a file in your current directory.

There is an **unalias** command too, to cancel an alias.

You can write other macros which are specific to the particular program you are debugging. For example, let's again suppose you have a variable named **ww** in **x.py**, and you wish to check its value each time the debugger pauses, say at breakpoints. Then change the above alias to

```
alias c c;;l;;ww
```

8.3.3 Using __dict__

In Section 8.8.3 below, we'll show that if **o** is an object of some class, then printing **o.__dict__** will print all the member variables of this object. Again, you could combine this with PDB's alias capability, e.g.

```
alias c c;;l;;o.__dict__
```

Actually, it would be simpler and more general to use

```
alias c c;;l;;self
```

This way you get information on the member variables no matter what class you are in. On the other hand, this apparently does not produce information on member variables in the parent class.

8.3.4 The type() Function

In reading someone else's code, or even one's own, one might not be clear what type of object a variable currently references. For this, the **type()** function is sometimes handy. Here are some examples of its use:

```
>>> x = [5,12,13]
>>> type(x)
<type 'list'>
>>> type(3)
<type 'int'>
>>> def f(y): return y*y
...
>>> f(5)
25
>>> type(f)
<type 'function'>
```

8.3.5 Using PDB with Emacs

Emacs is a combination text editor and tools collection. Many software engineers swear by it. It is available for Windows, Macs and Linux. But even if you are not an Emacs aficionado, you may find it to be an excellent way to use PDB. You can split Emacs into two windows, one for editing your program and the other for PDB. As you step through your code in the second window, you can see yourself progress through the code in the first.

To get started, say on your file **x.py**, go to a command window (whatever you have under your operating system), and type either

```
emacs x.py
```

or

```
emacs -nw x.py
```

The former will create a new Emacs window, where you will have mouse operations available, while the latter will run Emacs in text-only operations in the current window. I'll call the former "GUI mode."

Then type **M-x pdb**, where for most systems "M," which stands for "meta," means the Escape (or Alt) key rather than the letter M. You'll be asked how to run PDB; answer in the manner you would run PDB externally to Emacs (but with a full path name), including arguments, e.g.

```
/usr/local/lib/python2.4/pdb.py x.py 3 8
```

where the 3 and 8 in this example are your program's command-line arguments.

At that point Emacs will split into two windows, as described earlier. You can set breakpoints directly in the PDB window as usual, or by hitting **C-x space** at the desired line in your program's window; here and below, "C-" means hitting the control key and holding it while you type the next key.

At that point, run PDB as usual.

If you change your program and are using the GUI version of Emacs, hit IM-Python | Rescan to make the new version of your program known to PDB.

In addition to coordinating PDB with your error, note that another advantage of Emacs in this context is that Emacs will be in Python mode, which gives you some extra editing commands specific to Python. I'll describe them below.

In terms of general editing commands, plug "Emacs tutorial" or "Emacs commands" into your favorite Web search engine, and you'll see tons of resources. Here I'll give you just enough to get started.

First, there is the notion of a **buffer**. Each file you are editing[6] has its own buffer. Each other action you take produces a buffer too. For instance, if you invoke one of Emacs' online help commands, a buffer is created for it (which you can edit, save, etc. if you wish). An example relevant here is PDB. When you do **M-x pdb**, that produces a buffer for it. So, at any given time, you may have several buffers. You also may have several windows, though for simplicity we'll assume just two windows here.

In the following table, we show commands for both the text-only and the GUI versions of Emacs. Of course, you can use the text-based commands in the GUI too.

action	text	GUI
cursor movement	arrow keys, PageUp/Down	mouse, left scrollbar
undo	C-x u	Edit | Undo
cut	C-space (cursor move) C-w	select region | Edit | Cut
paste	C-y	Edit | Paste
search for string	C-s	Edit | Search
mark region	C-@	select region
go to other window	C-x o	click window
enlarge window	(1 line at a time) C-x ˆ	drag bar
repeat folowing command n times	M-x n	M-x n
list buffers	C-x C-b	Buffers
go to a buffer	C-x b	Buffers
exit Emacs	C-x C-c	File | Exit Emacs

In using PDB, keep in mind that the name of your PDB buffer will begin with "gud," e.g. **gud-x.py**.

You can get a list of special Python operations in Emacs by typing **C-h d** and then requesting info in **python-mode**. One nice thing right off the bat is that Emacs' **python-mode** adds a special touch to auto-indenting: It will automatically indent further right after a **def** or **class** line. Here are some operations:

action	text	GUI
comment-out region	C-space (cursor move) C-c #	select region | Python | Comment
go to start of def or class	ESC C-a	ESC C-a
go to end of def or class	ESC C-e	ESC C-e
go one block outward	C-c C-u	C-c C-u
shift region right	mark region, C-c C-r	mark region, Python | Shift right
shift region left	mark region, C-c C-l	mark region, Python | Shift left

[6] There may be several at once, e.g. if your program consists of two or more source files.

8.4 Debugging with Xpdb

Well, this one is my own creation. I developed it under the premise that PDB would be fine if only it had a window in which to watch my movement through my source code (as with Emacs above). Try it! Very easy to set up and use. Go to `http://heather.cs.ucdavis.edu/~matloff/Python/Xpdb`.

8.5 Debugging with Winpdb (GUI)

The Winpdb debugger (`http://winpdb.org/`),[7] is very good. Its functionality is excellent, and its GUI is very attractive visually.

Among other things, it can be used to debug threaded code, curses-based code and so on, which many debuggers can't.

Winpdb is a GUI front end to the text-based RPDB2, which is in the same package. I have a tutorial on both at `http://heather.cs.ucdavis.edu/~matloff/winpdb.html`.

8.6 Debugging with Eclipse (GUI)

I personally do not like integrated development environments (IDEs). They tend to be very slow to load, often do not allow me to use my favorite text editor,[8] and in my view they do not add much functionality. However, if you are a fan of IDEs, here are some suggestions:

However, if you like IDEs, I do suggest Eclipse, which I have a tutorial for at `http://heather.cs.ucdavis.edu/~matloff/eclipse.html`. My tutorial is more complete than most, enabling you to avoid the "gotchas" and have smooth sailing.

8.7 Debugging with PUDB

What a nice little tool! Uses Curses, so its screen footprint is tiny (same as your terminal, as it runs there). Use keys like n for Next, as usual. Variables, stack etc. displayed in right-hand half of the screen.

See `http://heather.cs.ucdavis.edu/~matloff/pudb.htm` for a slideshow-like tutorial.

[7] No, it's not just for Microsoft Windows machines, in spite of the name.

[8] I use vim, but the main point is that I want to use the same editor for all my work—programming, writing, e-mail, Web site development, etc.

8.8 Some Python Internal Debugging Aids

There are various built-in functions in Python that you may find helpful during the debugging process.

8.8.1 The `str()` Method

The built-in method **str()** converts objects to strings. For scalars or lists, this has the obvious effect, e.g.

```
>>> x = [1,2,3]
>>> str(x)
'[1, 2, 3]'
```

But what if **str()** is applied to an instance of a class? If for example we run our example program from Section 1.11.1 in PDB, we would get a result like this

```
(Pdb) str(b)
'<tfe.textfile instance at 0x81bb78c>'
```

This might not be too helpful. However, we can override **str()** as follows. Within the definition of the class **textfile**, we can override the built-in method **__str__()**, which is the method that defines the effect of applying **str()** to objects of this class. We add the following code to the definition of the class **textfile**:

```
   def __str__(self):
      return self.name+' '+str(self.nlines)+' '+str(self.nwords)
```

(Note that the method is required to return a string.)

Running in PDB now, we get

```
(Pdb) str(b)
'y 3 3'
```

Again, you could arrange so that were printed automatically, at every pause, e.g.

```
alias c c;;l;;str(b)
```

8.8.2 The `locals()` Function

The **locals()** function, for instance, will print out all the local variables in the current item (class instance, method, etc.). For example, in the code **tfe.py** from Section 1.11.1, let's put breakpoints at the lines

```
self.nwords += len(w)
```

and

```
print "the number of text files open is", textfile.ntfiles
```

and then call **locals()** from the PDB command line the second and first times we hit those breakpoints, respectively. Here is what we get:

```
(Pdb) locals()
{’self’: <__main__.textfile instance at 0x8187c0c>, ’l’: ’Here is an
example\n’, ’w’: [’Here’, ’is’, ’an’, ’example’]}
...
(Pdb) c
> /www/matloff/public_html/Python/tfe.py(27)main()
-> print "the number of text files open is", textfile.ntfiles
(Pdb) locals()
{’a’: <__main__.textfile instance at 0x8187c0c>, ’b’: <__main__.textfile
instance at 0x81c0654>}
```

There is also a built-in **globals()** function which plays a similar role,

8.8.3 The __dict__ Attribute

This was covered in Section 8.3.3.

8.8.4 The id() Function

Sometimes it is helpful to know the actual memory address of an object. For example, you may have two variables which you think point to the same object, but are not sure. The **id()** method will give you the address of the object. For example:

```
>>> x = [1,2,3]
>>> id(x)
-1084935956
>>> id(x[1])
137809316
```

(Don’t worry about the “negative” address, which just reflects the fact that the address was so high that, viewed as a 2s-complement integer, it is “negative.”)