A Quick, Painless Tutorial on the Python Language

Norman Matloff
University of California, Davis

October 24, 2006

Contents

1 Overview

1.1 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. But in most applications, execution speed is not important, 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 Perl and Python 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.

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

- Used often for system administration, Web programming and "rapid prototyping."
- Very casual with regard to typing of variables, e.g. little or no distinction between integer, floating-point or string variables. Arrays can mix elements of different "types," such as integers and strings. 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.

1.2 Why Python?

Today the most popular scripting language is probably Perl. However, many people, including me, prefer Python, as it is much cleaner and more elegant. For example, 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.

2 How to Use This Tutorial

2.1 Background Needed

Anyone with even a bit of programming experience should find the material through Section 8 to be quite accessible.

The material beginning with Section 10 will feel quite comfortable to anyone with background in an object-oriented language such as C++ or Java. If you lack this background, you will still be able to read these

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

sections, but will probably need to go through them more slowly than those who do know C++ or Java; just focus on the examples, not the terminology.

There will be a couple of places in which we describe things briefly in a Unix context, so some Unix knowledge would be helpful, but certainly not required. Python is used on Windows and Macintosh platforms too, not just Unix.

2.2 Approach

Our 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. For example, the usual approach would be to state all possible forms that a Python literal can take on.

I avoid this here. Again, the aim is to enable the reader to quickly acquire a Python foundation. He/she should then be able to delve directly into some special topic if and when the need arises.

2.3 What Parts to Read, When

I would suggest that you first read through Section 8, and then give Python a bit of a try yourself. First experiment a bit in Python's interactive mode (Section 3.4). 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, with a basic knowledge of Python's object-oriented programming features and Python modules/packages. So you should next read through Section 15.

That would be a very solid foundation for you from which to make good use of Python. Eventually, you may start to notice that many Python programmers make use of Python's functional programming features, and you may wish to understand what the others are doing or maybe use these features yourself. If so, Section 17 will get you started.

Don't forget the appendices! The key ones are Sections A and B.

I also have a number of tutorials on Python special programming, e.g. network programming, iterators/generators, etc. See `http://heather.cs.ucdavis.edu/~matloff/python.html`.

3 A 5-Minute Introductory Example

3.1 Example Program Code

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

[2] The raw source file for this tutorial is downloadable at `http://heather.cs.ucdavis.edu/~matloff/Python/PythonIntro.tex`, so you don't have to type the programs yourself. You can either edit a copy of this file, saving only the lines of the program example you want, or use your mouse to do a copy-and-paste operation for the relevant lines.

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.

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

for x = 0.1, 0.2, ..., 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
```

3.2 Python Lists

How does the program work? First, Python's **range()** function is an example of the use of **lists**, i.e. Python arrays,[3] 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.

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

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

[3] 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.

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

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
...
8
```

3.3 Python Block Definition

Now focus your attention on that inoccuous-looking colon at the end of the **for** line, 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[4]

```
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.[5] 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++.[6] 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.

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

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

[6] 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**. The boolean operators are **and**, **or** and **not**.

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.

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

3.4 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 this 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.[7]

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.

[7] 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.

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 0x401a1aa0>
```

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. the more typical

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

the object was printed out.

3.5 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'
```

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. Then we must refer to the functions in the context of the math library. For example, the functions **sqrt()** and **sin()** must be prefixed by **math**:[8]

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

[8] A method for avoiding the prefix is shown in Sec. 13.1.2.

4 A 10-Minute Introductory Example

4.1 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:

```
1  # reads in the text file whose name is specified on the command line,
2  # and reports the number of lines and words
3
4  import sys
5
6  def checkline():
7     global l
8     global wordcount
9     w = l.split()
10    wordcount += len(w)
11
12 wordcount = 0
13 f = open(sys.argv[1])
14 flines = f.readlines()
15 linecount = len(flines)
16 for l in flines:
17    checkline()
18 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. 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
- variable scoping

I will discuss these features in the next few sections.

4.2 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 list, analogous to **argv** in C/C++.[9] 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()**.[10] For floating-point, in Python we'd use **float()**.[11]

4.3 Introduction to File Manipulation

The function **open()** is similar to the one in C/C++. This creates an object **f** of **file** class.

The **readlines()** function of the **file** class returns a list 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.)

5 Declaration (or Not), Scope, Functions, Etc.

5.1 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()
```

[9] 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)**.

[10] We would also use it like C/C++'s **floor()**, in applications that need such an operation.

[11] 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 has the value **None** (and this can be assigned to a variable, tested for in an **if** statement, etc.).

5.2 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 13.1.5, 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++.

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. So, in the code for **checkline()**, Python would assume that **l** and **wordcount** are local to **checkline()** if we don't inform it otherwise. We do the latter with the **global** keyword.

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 pretty much *mandatory*.

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

6 A Couple of Built-In Functions

The function **len()** returns the number of elements in a list, in this case, 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.[12] It splits a string into a list of words, for example.[13] 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, [].)

7 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 don't have types, but their values do. In other words, a variable **X** might be bound to an integer at one point in your program and then be rebound to a class instance at another point. In other words, Python uses **dynamic typing**.

Python's types include notions of scalars, **sequences** (lists or **tuples**) and dictionaries (associative arrays, discussed in Sec. 7.3), classes, function, etc.

[12] Member functions of classes are referred to as *methods*.

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

7.1 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 15.4.

7.2 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[14]
- **slicing** operations, i.e. the extraction of subsequences
- use of **+** and ***** operators for concatenation and replication

7.2.1 Lists (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.

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

[14]This function is applicable to dictionaries too.

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

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

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]
>>> y = x.pop(0)  # deletes and returns deleted value
>>> y
1
>>> x
[2, 3, 4, 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.

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
```

Multidimensional arrays 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
```

7.2.2 Tuples

Tuples are like lists, but are **immutable**, i.e. unchangeable. They are enclosed by parentheses or nothing at all, rather than brackets.[15]

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)]
```

7.2.3 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. For example:

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

[15] 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 an empty list, i.e. **(,)**.

(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 11.)

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.

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:[16]

```
>>> '---'.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:[17]

```
>>> x = 'abc'
>>> x.upper()
'ABC'
>>> 'abc'.upper()
'ABC'
>>> 'abc'.center(5) # center the string within a 5-character set
' abc '
```

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.

[16]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.

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

7.2.4 Sorting

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 17.1.)

7.3 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.[18] 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:

```
1  >>> x = {'abc':12,'sailing':'away'}
2  >>> x['abc']
3  12
4  >>> y = x.keys()
5  >>> y
6  ['abc', 'sailing']
```

[18] Now one sees a reason why Python distinguishes between tuples and lists. Allowing mutable keys would be an implementation nightmare, and probably lead to error-prone programming.

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

Note how we added a new element to **x** near the end.

7.4 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. In function names *are* variables; we just temporarily assign a set of code to a name. Consider:

```
>>> def square(x):
...     return x*x
...
>>> square(3)
9
>>> gy = square
>>> gy(3)
9
>>> def cube(x):
...     return x**3
...
>>> cube(3)
27
>>> square = cube
>>> square(3)
27
>>> square = 8.8
>>> square
8.8000000000000007
```

8 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'
```

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__**.

For example, let's add a statement

```
print __name__
```

to our very first code example, from Section 3.1, 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:

```
>>> __name__
'__main__'
>>> 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.

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
...
```

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

10 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 object-oriented programming (OOP).

10.1 Example Program Code

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

```
1  class textfile:
2     ntfiles = 0  # count of number of textfile objects
3     def __init__(self,fname):
4        textfile.ntfiles += 1
5        self.name = fname  # name
6        self.fh = open(fname)  # handle for the file
7        self.lines = self.fh.readlines()
8        self.nlines = len(self.lines)  # number of lines
9        self.nwords = 0  # number of words
10       self.wordcount()
11    def wordcount(self):
12       "finds the number of words in the file"
13       for l in self.lines:
14          w = l.split()
15          self.nwords += len(w)
16    def grep(self,target):
17       "prints out all lines containing target"
18       for l in self.lines:
19          if l.find(target) >= 0:
20             print l
21
22 a = textfile('x')
23 b = textfile('y')
24 print "the number of text files open is", textfile.ntfiles
25 print "here is some information about them (name, lines, words):"
26 for f in [a,b]:
27    print f.name,f.nlines,f.nwords
28 a.grep('example')
```

In addition to the file **x** I used in Section 4 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
```

10.2 The Keyword self

Let's take a look at the class **textfile**. The first thing to note is the prevalence of the keyword **self**, meaning the current instance of the class, analogous to **this** in C++ and Java. So, **self** is a pointer to the current instance of the class.

10.3 Instance Variables

In general OOP terminology, an instance variable **x** of a class is a member variable for which each instance of the class has a separate value of that variable, In the C++ or Java world, you know this as a non-`static` variable. The term *instance variable* is the generic OOP term, non-language specific.

To see how these work in Python, recall first that a variable in Python is created when something is assigned to it. So, an instance variable in an instance of a Python class does not exist until it is assigned to.

```
self.name = fname  # name
```

is executed, the member variable **name** for the current instance of the class is created, and is assigned the indicated value.

10.4 Class Variables

A class variable **v**, having a common value in all instances of the class,[19] 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.[20]

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.

10.5 Constructors and Destructors

The constructor for a class must be named __**init()**__. The argument **self** is mandatory, and you can add others, as I've done in this case with a filename.[21]

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.

10.6 Instance Methods

The method **wordcount()** is an instance method, i.e. applies specifically to the given object of this class.[22] Unlike C++ and Java, where **this** is an implicit argument to instance methods, Python wisely makes the relation explicit; the argument **self** is required.

10.7 Docstrings

There is a double-quoted string, "finds the number of words in the file", at the beginning of **wordcount()**. 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:

[19] Again in the C++ or Java world, you know this as a `static` variable.

[20] 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.

[21] The argument **self** is mandatory, but the name can be different. Unlike C++/Java, **self** is not a reserved word. You can use any name you wish, as long as you are consistent, e.g. in the __**init()**__ method described below. But it would be considered bad form to use some other name.

[22] Again, in C++/Java terminology, these are non-**static** methods.

```
>>> 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.

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

By the way, method arguments in Python can only be pass-by-value, in the sense of C: Functions have side effects with respect to the parameter if the latter is a pointer. (Python does not have formal pointers, but it does have references; see Section 11.)

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.[23]

10.8 Class Methods

Before Version 2.2, Python had no formal provision for class methods, i.e. methods which do not apply to specific objects of the class. Now Python has two (slightly differing) ways to do this, using functions **staticmethod()** and **classmethod()**. I will present use of the former, in the following enhancement to the code in Section 10.1 within the class **textfile**:

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

...

# here we are in "main"
...
textfile.totfiles()
...
```

Note that class methods do not have the **self** argument.

Note also that this method could be called even if there are not yet any instances of the class **textfile**. In the example here, 0 would be printed out, since no files had yet been counted.[24]

10.9 Derived Classes

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

[23] 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.

[24] 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.

```
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
```

The official Python tutorial notes, "[In the C++ sense] all methods in Python are effectively virtual." If you wish to extend, rather than override a method in the base class, you can refer to the latter by prepending the base name, as in our example **a.__init__(self)** above.

10.10 A Word on Class Implementation

A Python class instance is implemented 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. In our "main" program, for example, we could have a statement like

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

11 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]
>>> y = x  # x and y now both point to [1,2,3]
>>> x[2] = 5  # this means y[2] changes to 5 too!
>>> y[2]
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, and though those objects have the same values for the time being, if the object pointed to by **x** changes, **y**'s object won't change.

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
...
>>> x = 5
>>> f(x)
>>> x
5
>>> def g(a):
...     a[0] = 2*a[0]
...
>>> y = [5]
>>> g(y)
>>> y
[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
```

Objects can be deleted using **del**.

For some more practice with these notions, see Section C.

12 Object Comparison

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

```
if x < y:
```

for lists **x** and **y**. The comparison is lexicographic.

For example,

```
>>> [12,'tuv'] < [12,'xyz']
True
>>> [5,'xyz'] > [12,'tuv']
False
```

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

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

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

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

13 Modules and Packages

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! :-)

13.1 Modules

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![25]

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

13.1.1 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, executable code not part of any function, etc.) Our test program file, **tftest.py**, might now look like this:

```
# file tftest.py

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')
```

13.1.2 How `import` Works

The Python interpreter, upon seeing the statement **import tf**, would load the contents of the file **tf.py**.[26] 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. See what happens when we do **import fme2** in an example in Section 9 below.)

[26] 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.

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:

```
1  from tf import textfile
2
3  a = textfile('x')
4  b = textfile('y')
5  print "the number of text files open is", textfile.ntfiles
6  print "here is some information about them (name, lines, words):"
7  for f in [a,b]:
8     print f.name,f.nlines,f.nwords
9  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 *
```

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.

13.1.3 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.

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 B.1).

13.1.4 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.

13.1.5 A Note on Global Variables

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."

Indeed, a global variable in a module is merely an attribute (i.e. a member entity) of that module, similar to a class variable's role within a class. When module B is imported by module A, B's namespace is copied to A's. If module B has a global variable **X**, then module A will create a variable of that name, whose initial value is whatever module B had for its variable of that name at the time of importing. Changes to **X** in one of the modules will NOT be reflected in the other.

Say **X** does change in B, but we want code in A to be able to get the latest value of **X** in B. We can do that by including a function, say named **GetX()** in B. Assuming that A imported everything from B, then A will get a function **GetX()** which is a copy of B's function of that name, and whose sole purpose is to return the value of **X**. Unless B changes that function (which *is* possible, e.g. functions may be assigned), the functions in the two modules will always be the same, and thus A can use its function to get the value of **X** in B.

13.2 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 13.1.1 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.

13.3 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. 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. If you are on a Unix system and using the C shell, for instance, 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
```

The current path is contained in **sys.path**. Again, 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.[27]

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

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

14 Exception Handling

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]
```

This is not just for file operations. Consider this code, for instance:

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

15 Miscellaneous

15.1 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[28]

```
% python x.py
```

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 Unix **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 Unix 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.

[28] This section will be Unix-specific.

15.2 Named Arguments in Functions

Consider this little example:

```
1  def f(u,v=2):
2     return u+v
3
4  def main():
5     x = 2;
6     y = 3;
7     print f(x,y)  # prints 5
8     print f(x)  # prints 4
9
10 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.

15.3 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**.

15.4 Formatted String Manipulation

Python supports C-style "printf()", e.g.

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

prints out

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

Note the importance 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.

This is nice, but it is far more powerful than just for printing, but for general string manipulation. 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.[29]

16 Example of 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).[30]

```
1  # bintree.py, a module for handling sorted binary trees; values to be
2  # stored can be general, as long as an ordering relation exists
3
4  # here, only have routines to insert and print, but could add delete,
5  # etc.
6
7  class treenode:
8     def __init__(self,v):
9        self.value = v;
10       self.left = None;
11       self.right = None;
12    def ins(self,nd):  # inserts the node nd into tree rooted at self
13       m = nd.value
14       if m < self.value:
15          if self.left == None:
16             self.left = nd
17          else:
18             self.left.ins(nd)
19       else:
20          if self.right == None:
21             self.right = nd
22          else:
23             self.right.ins(nd)
24    def prnt(self):  # prints the subtree rooted at self
```

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

[30] But did you get the pun?

```
25        if self.value == None: return
26        if self.left != None: self.left.prnt()
27        print self.value
28        if self.right != None: self.right.prnt()
29
30  class tree:
31     def __init__(self):
32        self.root = None
33     def insrt(self,m):
34        newnode = treenode(m)
35        if self.root == None:
36           self.root = newnode
37           return
38        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 12.) 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']
```

16.1 Making Use of Python Idioms

In the example in Section 10.1, 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.[31] 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 $\mathbf{j} < \mathbf{10}$), 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.[32]

Another great way to avoid loops is to use Python's **functional programming** features, described in Section 17.

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

[31] Nor do they all have to be of the same type at all. One can have diverse items within the same list.

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

17 Functional Programming Features

These features provide quick, 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.

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 16.1, this makes for cleaner, clearer code.

17.1 Lambda Functions

Lambda functions provide a way of defining short functions. They help you avoid cluttering up your code with a lot of "one-liners" which 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 such as **map()** and **filter()**, as seen next.

Here is a more realistic illustration, redoing the sort example from Section 7.2.4:

```
>>> 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]]
```

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.

17.2 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.[33]

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]
```

Clearly, this is not feasible except for simple situations. For more complex cases, we would use a non-lambda function. For example, here is a revised version of the program in Section 4.1:

```
1  import sys
2
3  def checkline(l):
4     global wordcount
5     w = l.split()
6     wordcount += len(w)
7
8  wordcount = 0
9  f = open(sys.argv[1])
10 flines = f.readlines()
11 linecount = len(flines)
12 map(checkline,flines)  # replaces the old 'for' loop
13 print linecount, wordcount
```

Note that **l** is now an argument to **checkline()**.

Of course, this could be reduced even further, with the heart of the "main" program above being changed to

```
map(checkline,open(sys.argv[1]).readlines)
```

But this is getting pretty hard to read and debug.

[33] Side note: Note again that if we had not assigned to **z**, the list [3,6,4] would have been printed out anyway. In interactive mode, any Python non-assignment statement prints the value of the result.

17.3 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.

17.4 List Comprehension

This allows you to compactify a **for** loop which 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()**.

17.5 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.

17.6 Generator Expressions

Recall that a big advantage of iterators is that we can often use them to avoid producing a large list in memory. Generators expressions, new in Python 2.4, produce the same improvement over list comprehensions.

Consider this little example, which will review list comprehension and introduce generator expressions:

```
>>> x = [1,2,4,12,5]
>>> [i for i in x]
[1, 2, 4, 12, 5]
>>> [i for i in x if i > 4]
[12, 5]
>>> sum(i for i in x if i > 4)
17
>>> min(i for i in x if i > 4)
5
```

In other words, a generator expression has the same form, as a list comprehension, with two changes:

- the expression uses parentheses instead of brackets
- a function which is applicable to iterators is applied to the expression[34]

By the way, the **min()** function can be used with ordinary ordinary arguments too:

```
>>> min(5,12,3)
3
>>> y = [5,12,4]
>>> min(y)
4
```

Moreover, it can be used on anything comparable in lexicographic order:

```
>>> min('def','cxyg')
'cxyg'
>>> min([4,'abc'],[8,5],[3,200])
[3, 200]
```

Of course, **max()** is similar. If these are combined in generator expressions, one has a very powerful tool.

A Debugging

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`.

A.1 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.

[34] Note that the parentheses surrounding the expression also serve as the parentheses for the function.

- 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.
- The basic form in conjunction with the DDD GUI for debuggers.

A.1.1 The Basics

You should be able to find PDB in the **lib** subdirectory of your Python package. On a Unix system, for example, that is probably in something like **/usr/lib/python2.2**. **/usr/local/lib/python2.4**, etc. To debug a script **x.py**, type

```
% /usr/lib/python2.2/pdb.py x.py
```

(If **x.py** had had command-line arguments, they would be placed after **x.py** on the command line.)

Of course, since you will use PDB a lot, it would be better to make an alias for it. For example, under Unix in the C-shell:

```
alias pdb /usr/lib/python2.2/pdb.py
```

so that you can write more simply

```
% pdb x.py
```

Once you are in PDB, set your first breakpoint, say at line 12:

```
b 12
```

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
- **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
- **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[35]
- **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**.[36]

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.[37]

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.

[35] The tutorial is run through a pager. Hit the space bar to go to the next page, and the q key to quit.

[36] Note also that if the module is implemented in C, you of course will not be able to break there.

[37] 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.[38]

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.

A.1.2 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 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 Unix is **$HOME/.pdbrc**.[39] 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
```

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
```

[38] 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.

[39] Python will also check for such a file in your current directory.

A.1.3 Using __dict__

In Section A.4.1 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.

A.2 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 Unix/Linux; it is included in most Linux distributions. 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), 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[40] 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 ^	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

[40] There may be several at once, e.g. if your program consists of two or more source files.

A.3 Using PDB with DDD

DDD, available on many Unix systems (and freely downloadable if your system doesn't have it), is a GUI for many debuggers, such GDB (for C/C++), JDB (for Java), Perl's built-in Perl debugger, and so on. It can be used on PDB for Python, and thus make your usage of PDB more enjoyable and productive.

A.3.1 Preparation

The first use of DDD on PDB was designed by Richard Wolff. He modified **pdb.py** slightly for this purpose, calling the new PDB **pydb.py**. It was designed for Python 1.5, but he has kindly provided an update for me. You'll need the files

```
http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydb.py
http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydbcmd.py
http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydbsupt.py
```

Place the files somewhere in your search path, say **/usr/bin**. Make sure you give them execute permission.

A.3.2 DDD Launch and Program Loading

To start, say for debugging **fme2.py** in Section 9, first make sure that **main()** is set up as described in that section.

When you invoke DDD, tell it to use PYDB:

```
ddd --debugger /usr/bin/pydb.py
```

Then in DDD's Console, i.e. the PDB command subwindow (near the bottom), type

```
(pdb) file fme2.py
```

or choose File | Open Source and double-click on the file name.

Later, when you make a change to your source code, issue the command

```
(pdb) file fme2.py
```

Or select either File | Open Source or File | Open Recent and double-click on the file name. Your breakpoints from the last run will be retained.

A.3.3 Breakpoints

To set a breakpoint, right-click somewhere in blank space on the line in your source file and choose Set Breakpoint (or Set Temporary Breakpoint or Continue to Until Here, as the case may be).

A.3.4 Running Your Code

To run, click on Program | Run, fill in your program's command line arguments if any in the Run with Arguments box, and click Run in that pop-up window. You will be asked to hit Continue, which you could do by clicking Program | Run |, but is more conveniently done by clicking Continue in the little command summary window. (But don't use Run there.)

You can then click on Next, Step, Cont etc. The marker for the current execution line is shaped like an 'I', though rather faint when the mouse pointer is not in the source code section of the DDD window..

By the way, do not refer to **sys.argv** in freestanding code within a class. When your program is first loaded, any freestanding code will be executed, and since the command-line arguments won't have been loaded yet, so you will get an "index out of range" error. Avoid this by putting code involving **sys.argv** either inside a function in the class, or outside the class entirely.

A.3.5 Inspecting Variables

You can inspect the value of a variable by moving the mouse pointer to any instance of the variable in the source code window.

As mentioned in Section A.1.3, if **o** is an object of some class, then printing **o.__dict__** will print all the member variables of this object. In DDD, you can do this even more easily, as follows. Simply put that expression in a comment, e.g.

```
# o.__dict__
```

and then whenever you wish to inspect the member variables of **o**, simply move the mouse pointer to that expression in the comment!

Make good use of DDD's feature which allows a variable to be displayed continuously. Simply right-click on any instance of the variable, and then choose Display.

A.3.6 Miscellaneous

DDD, developed originally for C/C++, is not always a perfect match to Python. But since what DDD actually does is relay your clicked commands to PDB, as you can see in DDD's Console, whatever DDD can't do for you, you can type PDB commands directly into the Console.

A.4 Some Python Internal Debugging Aids

There are various built-in functions in Python that you may find helpful during the debugging process.

A.4.1 The `__dict__` Attribute

Recall that class instances are implemented as dictionaries. If you have a class instance **i**, you can view the dictionary which is its implementation via **i.__dict__**. This will show you the values of all the member variables of the class.

A.4.2 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.")

A.5 Other Debugging Tools/IDEs

The new Winpdb debugger (`www.digitalpeers.com/pythondebugger/`),[41] looks very good. It sounds especially good to me because is based on an older debugger, RPDB, which is good at handling Python threads. However, Winpdb's documentation, at least as of January 2006, is weak, and I have not yet had a chance to use it. Check `http://heather.cs.ucdavis.edu/~matloff/rpdb.html` for updates.

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,[42] and in my view they do not add much functionality. However, if you are a fan of IDEs, here are some suggestions:

You may wish to try IDLE, a Python IDE included with the Python package, though I find it sorely lacking. But I've heard very nice things about the following free IDEs: Boa Constructor IDE (get the pun?), `http://boa-constructor.sourceforge.net/`; Eclipse with a Python plug-in, `http://www.python.org/moin/EclipsePythonIntegration`; and JEdit, `http://www.jedit.org/`. I have a tutorial on Eclipse at `http://heather.cs.ucdavis.edu/~matloff/eclipse.html`. Among commercial products, I've always liked Wing Edit, `http://wingware.com/wingide/platforms`, which I used briefly at one time for Java.

B Online Documentation

B.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 10.1 suppose we stop at the line

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

[41] No, it's not just for Microsoft Windows machines, in spite of the name.

[42] 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.

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 7.2.3, 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']
```

B.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 Unix command line, e.g.

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

B.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
```

C Explanation of the Old Class Variable Workaround

Before Python 2.2, there was no provision for class methods. But there was simple workaround, as seen in `http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52304`.

Since Python now allos class methods (two kinds, in fact), we don't need to use the workaround in our programming, but it gives us an excellent example of how object references work, so let's take a look.

In the program **tfe.py** in Section 10.1, we could add

```
1  class ClassMethod:
2     def __init__(self,MethodName):
3        self.__call__ = MethodName
4
5  class textfile:
6     (lots of stuff omitted)
7     def printntfiles():
8        print textfile.ntfiles
9     tmp = ClassMethod(printntfiles)
10    printntfiles = tmp
11
12 textfile.printntfiles()
```

(I've added a line here—the one which is an assignment to **tmp**—for the sake of clarity, as you will see below.)

Recall from Section 11 that functions in Python are (pointers to) objects like anything else, and thus assignable. For example, look at:

```
>>> def f(x):
...    return x*x
...
>>> g = f
>>> g(3)
9
```

In other words, **f** is really just a pointer to a "function object," so if we assign **f** to **g**, then **g** points to that object too, and consequently **g** is the same function.

Second, it is important to know that class instances are callable. This is where the **__call__()** method, implicitly part of every class, comes in. One simply pretends that the class instance is a function, and then calls it. That triggers execution of the **__call__()** method on **self** (i.e. the class instance which we are "calling") with whatever arguments we give it. The **__call__()** method is empty by default, but we can supply one, and use it to our advantage.

For example, say this is the source file **yyy.py**:

```
1  class u:
2     def __init__(self):
3        self.r = 8
4     def __call__(self,z):
5        return self.r*z
```

```
6
7  def main():
8     a = u()
9     print a(5)
10    a.r = 13
11    print a(5)
12
13 if __name__ == '__main__':
14    main()
```

```
python yyy.py
40
65
```

Now, look at the line from the workaround code above,

```
tmp = ClassMethod(printntfiles)
```

This creates an instance of the class **ClassMethod** and assigns it to **tmp**. The parameter was **printntfiles**, so the constructor line

```
self.__call__ = MethodName
```

will mean that **tmp.__call__** is set to **printntfiles**. Forget just for a moment that both **tmp.__call__** and **printntfiles** are functions; just focus on the fact that we have assigned one pointer (**printntfiles**) to another (**tmp.__call__**).

But of course both of those are indeed functions, and so **tmp.__call__** is now a pointer to the code for the original **printntfiles** function, i.e. to the code

```
print textfile.ntfiles
```

Now, the next line

```
printntfiles = tmp
```

means that the variable **printntfiles** no longer points to its own code! Instead, it now points to an instance of the class **ClassMethod**.

Now, remember, **printntfiles** itself is a class variable in our class **textfile**, i.e. its full name is **textfile.printntfiles**. Originally, it had been a function, but remember, anything can be assigned, so now it is simply a variable, pointing to an instance of the class **ClassMethod**.

The point is that if we now call **textfile.printntfiles**—which we can do, because any Python class instance is "callable"!—then we invoke the **__call__**() method of that instance. And that method, as seen above, points to the original code for our **printntfiles** function, i.e.

```
print textfile.ntfiles
```

Therefore that code will be executed!

So, what looks syntactically like a call to a class method,

```
textfile.printntfiles()
```

will in fact act like a class method, even though Python has no such thing.

D Putting All Globals into a Class

As mentioned in Section 5, 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**:

```
1  # reads in the text file whose name is specified on the command line,
2  # and reports the number of lines and words
3
4  import sys
5
6  def checkline():
7     glb.linecount += 1
8     w = glb.l.split()
9     glb.wordcount += len(w)
10
11 class glb:
12    linecount = 0
13    wordcount = 0
14    l = []
15
16 f = open(sys.argv[1])
17 for glb.l in f.readlines():
18    checkline()
19 print glb.linecount, glb.wordcount
```

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