A Quick, Painless Tutorial on the Python Language

Norman Matloff
University of California, Davis

August 12, 2011

Contents

1 Overview **5**

1.1 What Are Scripting Languages? 5

1.2 Why Python? 5

2 How to Use This Tutorial **5**

2.1 Background Needed 5

2.2 Approach 6

2.3 What Parts to Read, When 6

3 A 5-Minute Introductory Example **7**

3.1 Example Program Code 7

3.2 Python Lists 7

3.3 Python Block Definition 8

3.4 Python Also Offers an Interactive Mode 9

3.5 Python As a Calculator 10

4 A 10-Minute Introductory Example **11**

4.1 Example Program Code 11

4.2 Command-Line Arguments 12

4.3 Introduction to File Manipulation 13

4.4 Lack of Declaration 13

4.5 Locals Vs. Globals 13

4.6 A Couple of Built-In Functions 14

5 Types of Variables/Values **14**

6 String Versus Numerical Values **14**

7 Sequences **15**

7.1 Lists (Quasi-Arrays) 15

7.2 Tuples 17

7.3 Strings 17

7.3.1 Strings As Turbocharged Tuples 17

7.3.2 Formatted String Manipulation 19

7.4 Sorting Sequences 19

7.5 Dictionaries (Hashes) 20

7.6 Function Definition 21

8 Use of __name__ **22**

9 Object-Oriented Programming **23**

9.1 Example Program Code 24

9.2 The Objects 24

9.3 Constructors and Destructors 24

9.4 Instance Variables 25

9.5 Class Variables 25

9.6 Instance Methods 25

9.7 Class Methods 26

9.8 Derived Classes 26

9.9 A Word on Class Implementation 27

10 Importance of Understanding Object References **27**

11 Object Deletion **28**

12 Object Comparison **29**

13 Modules and Packages **30**

13.1 Modules . . . 30

13.1.1 Example Program Code . . . 30

13.1.2 How `import` Works . . . 31

13.1.3 Using reload() to Renew an Import . . . 31

13.1.4 Compiled Code . . . 32

13.1.5 Miscellaneous . . . 32

13.1.6 A Note on Global Variables Within Modules . . . 32

13.2 Data Hiding . . . 33

13.3 Packages . . . 33

14 Exception Handling (Not Just for Exceptions!) **34**

15 Docstrings **35**

16 Named Arguments in Functions **36**

17 Terminal I/O Issues **36**

17.1 Keyboard Input . . . 36

17.2 Printing Without a Newline or Blanks . . . 37

18 Example of Data Structures in Python **37**

18.1 Making Use of Python Idioms . . . 39

19 Functional Programming Features **39**

19.1 Lambda Functions . . . 40

19.2 Mapping . . . 40

19.3 Filtering . . . 41

19.4 Reduction . . . 42

19.5 List Comprehension . . . 42

19.6 Example: Textfile Class Revisited . . . 42

20 Decorators **43**

A Debugging **44**

A.1 Python's Built-In Debugger, PDB . . . 44
A.1.1 The Basics . . . 44
A.1.2 Using PDB Macros . . . 47
A.1.3 Using __dict__ . . . 47
A.1.4 The type() Function . . . 47
A.2 Using PDB with Emacs . . . 48
A.3 Debugging with Xpdb . . . 49
A.4 Debugging with Winpdb (GUI) . . . 50
A.5 Debugging with Eclipse (GUI) . . . 50
A.6 Some Python Internal Debugging Aids . . . 50
A.6.1 The `__dict__` Attribute . . . 50
A.6.2 The `id()` Function . . . 50
A.7 Debugging with PUDB . . . 51

B Online Documentation **51**
B.1 The dir() Function . . . 51
B.2 The help() Function . . . 52
B.3 PyDoc . . . 53

C Putting All Globals into a Class **53**

D Looking at the Python Virtual Machine **54**

E Running Python Scripts Without Explicitly Invoking the Interpreter **54**

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 (especially in the case of C). 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 Python and Perl the basic units are lines of text and words within lines. One can work with lines and words in C/C++, but one must go to greater effort to accomplish the same thing.

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

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

1.2 Why Python?

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

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

2 How to Use This Tutorial

2.1 Background Needed

Anyone with even a bit of programming experience should find the material through Section 17.1 to be quite accessible.

The material beginning with Section 9 will feel quite comfortable to anyone with background in an object-oriented programming (OOP) language such as C++ or Java. If you lack this background, you will still be able to read these sections, but will probably need to go through them more slowly than those who do know OOP; just focus on the examples, not the terminology.

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

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

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, with little or no contest. For example, the usual approach would be to first state all possible forms that a Python integer can take on, all possible forms a Python variable name can have, and for that matter how many different command-line options one can launch Python with.

I avoid this here. Again, the aim is to enable the reader to quickly acquire a Python foundation. He/she should then be able to delve directly into some special topic if and when the need arises. So, if you want to know, say, whether Python variable names can include underscores, you've come to the wrong place. If you want to quickly get into Python programming, this is hopefully the right place.

2.3 What Parts to Read, When

I would suggest that you first read through Section 17.1, 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, acquiring a basic knowledge of Python's OOP features and Python modules/packages. So you should next read through Section 19.

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 19 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`.

[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 edit a copy of this file, saving only the lines of the program example you want.

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

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

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

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

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

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

```
python fme.py
```

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

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

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**.

[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. The NumPy add-on package for Python offers true arrays.

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]:
```

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

The code

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

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

Python has a **while** construct too (though not an **until**). There is also a **break** statement like that of C/C++, used to leave loops "prematurely." For example:

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

3.3 Python Block Definition

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

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

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

[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.

```
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.

```
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 that in this tutorial, with interactive mode being an easy way to do a quick illustration of a feature.

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

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

[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**. You'll see examples as we move along.

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

```
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.

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

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

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

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

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

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

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

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

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

[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.

```
% 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.

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

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

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

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

```
This is an
example of a
text file.
```

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

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

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

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

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

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

I will discuss these features in the next few sections.

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 Python 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()**. For floating-point, in Python we'd use **float()**.[10]

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

4.3 Introduction to File Manipulation

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

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

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

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

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

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

4.4 Lack of Declaration

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

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

is executed.

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

4.5 Locals Vs. Globals

Python does not really have global variables in the sense of C/C++, in which the scope of a variable is an entire program. We will discuss this further in Section 13.1.6, but for now assume our source code consists of just a single **.py** file; in that case, Python does have global variables pretty much like in C/C++ (though with important differences).

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

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

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

4.6 A Couple of Built-In Functions

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

The method **split()** is a member of the **string** class.[11] It splits a string into a list of words, for example.[12] 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, [].)

5 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 (i.e. point to) an integer in one place in your program and then be rebound to a class instance at another point.

Python's types include notions of scalars, **sequences** (lists or **tuples**) and dictionaries (associative arrays, discussed in Sec. 7.5), classes, function, etc.

6 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 7.3.2.

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

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

7 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[13]
- **slicing** operations, i.e. the extraction of subsequences
- use of **+** and ***** operators for concatenation and replication

7.1 Lists (Quasi-Arrays)

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

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

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

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

```
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]
```

[13] This function is applicable to dictionaries too.

```
>>> 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 0th element
>>> 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 lists can be implemented as lists of lists. For example:

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

But be careful! Look what can go wrong:

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

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

The Python Wikibook (`http://en.wikibooks.org/wiki/Python_Programming/Lists`) suggests a solution, in the form of **list comprehensions**, which we cover in Section 19.5:

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

7.2 Tuples

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

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)]
```

7.3 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.

7.3.1 Strings As Turbocharged Tuples

Let's see some examples of string operations:

```
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'
```

(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 10.)

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:[14]

```
>>> '---'.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:

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

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

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

```
import string
```

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

[14]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.

7.3.2 Formatted String Manipulation

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

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

prints out

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

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

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

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

the portion

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

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

For example:

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

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

This is another very common idiom, quite powerful.[15]

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

7.4 Sorting Sequences

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

```
>>> x = [[1,4],[5,2]]
>>> x
[[1, 4], [5, 2]]
>>> x.sort()
>>> x
[[1, 4], [5, 2]]
```

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

```
>>> 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 19.1.)

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

7.5 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.[16] 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']
7  >>> z = x.values()
8  >>> z
9  [12, 'away']
10 x['uv'] = 2
11 >>> x
12 {'abc': 12, 'uv': 2, 'sailing': 'away'}
```

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

The keys need not be tuples. For example:

[16] 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.

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

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

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

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

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

7.6 Function Definition

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

```
return 8888
```

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

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

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

8 Use of __name__

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

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

```
% python x.py
```

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

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

For example, let's add a statement

```
print __name__
```

to our very first code example, from Section 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:

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

Our module's statement

```
print __name__
```

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

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

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

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

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

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

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.

9 Object-Oriented Programming

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

9.1 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
```

9.2 The Objects

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

9.3 Constructors and Destructors

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

Actually **self** is not a keyword. Unlike the **this** keyword in C++/Java, you do not HAVE TO call this variable **self**. Whatever you place in that first argument of **__init()__** will be used by Python's interpreter as a pointer to the current instance of the class. If in your definition of **__init()__** you were to name the first argument **me**,

and then write “me” instead of “self” throughout the definition of the class, that would work fine. However , you would invoke the wrath of purist pythonistas all over the world. So don’t do it.

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

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

9.4 Instance Variables

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

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

9.5 Class Variables

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

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

above.[17]

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.

9.6 Instance Methods

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

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

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, as we saw on page 28. (Python does not have formal pointers, but it does have references; see Section 10.)

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

[17] 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.

within the object string, or returning -1 if none is found.[18]

9.7 Class Methods

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

Python has two (slightly differing) ways to designate a function as a class method, via the functions **staticmethod()** and **classmethod()**. We will use only the former. As our first example, consider following enhancement to the code in within the class **textfile** above:

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

...

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

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

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

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

```
ntfiles = 0
```

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

```
class textfile:
```

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

9.8 Derived Classes

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

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

[18] 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.

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
```

9.9 A Word on Class Implementation

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

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

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

10 Importance of Understanding Object References

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> def f(a):
...    a = 2*a  # numbers are immutable
...
>>> x = 5
>>> f(x)
>>> x
5
>>> def g(a):
...    a[0] = 2*a[0]  # lists are mutable
...
>>> 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
```

11 Object Deletion

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

```
>>> del x
```

NOTE CAREFULLY THAT THIS IS DIFFERENT FROM DELETION FROM A LIST OR DICTIONARY. If you use **remove()** or **pop()**, for instance, you are simply removing the pointer to the object

from the given data structure, but as long as there is at least one **reference**, i.e. a pointer, to an object, that object still takes up space in memory.

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

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**. This "dictionary" ordering first compares the first element of one sequence to the first element of the other. If they aren't equal, we're done. If not, we compare the second elements, etc.

For example,

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

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

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

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

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

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

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

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

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

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![19]

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

```
1  # file tf.py
2
3  class textfile:
4     ntfiles = 0  # count of number of textfile objects
5     def __init__(self,fname):
6        textfile.ntfiles += 1
7        self.name = fname  # name
8        self.fh = open(fname)  # handle for the file
9        self.lines = self.fh.readlines()
10       self.nlines = len(self.lines)  # number of lines
11       self.nwords = 0  # number of words
12       self.wordcount()
13    def wordcount(self):
14       "finds the number of words in the file"
15       for l in self.lines:
16          w = l.split()
17          self.nwords += len(w)
18    def grep(self,target):
19       "prints out all lines containing target"
20       for l in self.lines:
21          if l.find(target) >= 0:
22             print l
```

Note that even though our module here consists of just a single class, we could have several classes, plus global variables,[20] executable code not part of any function, etc.)

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

```
1  # file tftest.py
2
3  import tf
4
5  a = tf.textfile('x')
6  b = tf.textfile('y')
```

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

[20] Though they would be global only to the module, not to a program which imports the module. See Section 13.1.6.

```
7  print "the number of text files open is", tf.textfile.ntfiles
8  print "here is some information about them (name, lines, words):"
9  for f in [a,b]:
10    print f.name,f.nlines,f.nwords
11 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**.[21] 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 8 below.)

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 Using reload() to Renew an Import

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

```
reload(tf)
```

[21] 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.

13.1.4 Compiled Code

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

Since modules are objects, the names of the variables, functions, classes etc. of a module are attributes of that module. Thus they are retained in the **.pyc** file, and will be visible, for instance, when you run the **dir()** function on that module (Section B.1).

13.1.5 Miscellaneous

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

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

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

13.1.6 A Note on Global Variables Within Modules

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

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

```
# x.py

import y

def f():
   global x
   x = 6

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

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

and **y.py**:

```
# y.py

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

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

```
x += 1
```

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

The line above the one generating the error message,

```
global x
```

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

```
x += 1
```

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

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

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, which is a collection of directories in which the interpreter will look for modules.

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

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

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

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

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

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

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

You can access the current path from within a Python program in the variable **sys.path**. It consists of a list of strings, one string for each directory, separated by colons. It can be printed out or changed by your code, just like any other variable.[22]

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

14 Exception Handling (Not Just for Exceptions!)

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

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

Here's another example:

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

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

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

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

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

Let's try it:

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

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

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

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

15 Docstrings

There is a double-quoted string, "finds the number of words in the file", at the beginning of **wordcount()** in the code in Section 9.1. This is called a **docstring**. It serves as a kind of comment, but at runtime, so that it can be used by debuggers and the like. Also, it enables users who have only the compiled form of the method, say as a commercial product, access to a "comment." Here is an example of how to access it, using **tf.py** from above:

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

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

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

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

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

16 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.

17 Terminal I/O Issues

17.1 Keyboard Input

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

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

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

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

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

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

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

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

17.2 Printing Without a Newline or Blanks

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

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

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

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

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

18 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).[23]

```
1  # bintree.py, a module for handling sorted binary trees; values to be
2  # stored can be general, not just numeric, as long as an ordering
3  # relation exists
4
5  # here, only have routines to insert and print, but could add delete,
6  # etc.
7
8  class treenode:
9     def __init__(self,v):  # create a 1-node tree, storing value v
10       self.value = v;
11       self.left = None;
12       self.right = None;
13    def ins(self,nd):  # inserts the node nd into tree rooted at self
14       m = nd.value
15       if m < self.value:
16          if self.left == None:
17             self.left = nd
18          else:
19             self.left.ins(nd)
20       else:
21          if self.right == None:
22             self.right = nd
23          else:
24             self.right.ins(nd)
25    def prnt(self):  # prints the subtree rooted at self
26       if self.value == None: return
27       if self.left != None: self.left.prnt()
28       print self.value
```

[23] But did you get the pun?

```
29        if self.right != None: self.right.prnt()
30
31  class tree:
32     def __init__(self):
33        self.root = None
34     def insrt(self,m):
35        newnode = treenode(m)
36        if self.root == None:
37           self.root = newnode
38           return
39        self.root.ins(newnode)
```

And here is a test:

```
1   # trybt1.py:  test of bintree.py
2   # usage:  python trybt.py numbers_to_insert
3
4   import sys
5   import bintree
6
7   def main():
8       tr = bintree.tree()
9       for n in sys.argv[1:]:
10         tr.insrt(int(n))
11      tr.root.prnt()
12
13  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']
```

18.1 Making Use of Python Idioms

In the example in Section 9.1, it is worth calling special attention to the line

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

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

You should almost never use C/C++ style **for** loops—i.e. where an index (say **j**), is tested against an upper bound (say $\mathbf{j} < \mathbf{10}$), 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.[24]

Another great way to avoid loops is to use Python's **functional programming** features, described in Section 19.

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

19 Functional Programming Features

These features provide concise ways of doing things which, though certainly doable via more basic constructs, compactify your code and thus make it easier to write and read. They may also make your code run

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

much faster. Moreover, it may help us avoid bugs, since a lot of the infracture we'd need to write ourselves, which would be bug-prone, is automatically taken care of us by the functional programming constructs.

Except for the first feature here (lambda functions), these features eliminate the need for explicit loops and explicit references to list elements. As mentioned in Section 18.1, this makes for cleaner, clearer code.

19.1 Lambda Functions

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

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

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

Here is a more realistic illustration, redoing the sort example from Section 7.4:

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

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

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

You see that the first argument is a named argument (recall Section 16), **cmp**. That is our compare function, which we defined above by writing `lambda u,v: u[1]-v[1]`.

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.

19.2 Mapping

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

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

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

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

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

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

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

we could work as follows:

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

Or, a little fancier:

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

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

19.3 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.

19.4 Reduction

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

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

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

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

19.5 List Comprehension

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

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

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

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

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

Here is pseudocode for what is going on:

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

Is that compactness worth the loss of readability? Only you can decide.

19.6 Example: Textfile Class Revisited

Here is the text file example from Section 9.1 again, now redone with functional programming features:

```
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
12    def wordcount(self):
13       "finds the number of words in the file"
14       self.nwords = \
15          reduce(lambda x,y: x+y,map(lambda line: len(line.split()),self.lines))
16    def grep(self,target):
17       "prints out all lines containing target"
18       lines = filter(lambda line: line.find(target) >= 0,self.lines)
19       print lines
20
21 a = textfile('x')
22 b = textfile('y')
23 print "the number of text files open is", textfile.ntfiles
24 print "here is some information about them (name, lines, words):"
25 for f in [a,b]:
26    print f.name,f.nlines,f.nwords
27 a.grep('example')
```

20 Decorators

Recall our example in Section 9.7 of how to designate a method as a class method:

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

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

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

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

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

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

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.
- 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.

A.1.1 The Basics

You should be able to find PDB in the **lib** subdirectory of your Python package. On a Linux 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 Linux in the C-shell:

```
alias pdb /usr/lib/python2.6/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
```

You can make it conditional, e.g.

```
b 12, z > 5
```

Hit **c** ("continue"), which you will get you into **x.py** and then stop at the breakpoint. Then continue as usual, with the main operations being like those of GDB:

- **b** ("break") to set a breakpoint
- **tbreak** to set a one-time breakpoint
- **ignore** to specify that a certain breakpoint will be ignored the next k times, where k is specified in the command
- **l** ("list") to list some lines of source code
- **n** ("next") to step to the next line, not stopping in function code if the current line is a function call
- **s** ("subroutine") same as **n**, except that the function *is* entered in the case of a call
- **c** ("continue") to continue until the next break point
- **w** ("where") to get a stack report
- **u** ("up") to move up a level in the stack, e.g. to query a local variable there
- **d** ("down") to move down a level in the stack
- **r** ("return") continue execution until the current function returns
- **j** ("jump") to jump to another line *without* the intervening code being executed
- **h** ("help") to get (minimal) online help (e.g. **h b** to get help on the **b** command, and simply **h** to get a list of all commands); type **h pdb** to get a tutorial on PDB[25]
- **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
```

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

Note, though, that you can't do this until **y** has actually been imported by **x**.[26]

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.[27]

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.

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.[28]

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.

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

[27] 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**.

[28] 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.

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 hit **c** to continue, when you next stop at a breakpoint you automatically get a listing of the neighboring code. This will really do a lot to make up for PDB's lack of a GUI.

In fact, this is so important that you should put it in your PDB startup file, which in Linux is **$HOME/.pdbrc**.[29] 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
```

A.1.3 Using __dict__

In Section A.6.1 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.1.4 The type() Function

In reading someone else's code, or even one's own, one might not be clear what type of object a variable currently references. For this, the **type()** function is sometimes handy. Here are some examples of its use:

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

```
>>> x = [5,12,13]
>>> type(x)
<type 'list'>
>>> type(3)
<type 'int'>
>>> def f(y): return y*y
...
>>> f(5)
25
>>> type(f)
<type 'function'>
```

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 Linux. But even if you are not an Emacs aficionado, you may find it to be an excellent way to use PDB. You can split Emacs into two windows, one for editing your program and the other for PDB. As you step through your code in the second window, you can see yourself progress through the code in the first.

To get started, say on your file **x.py**, go to a command window (whatever you have under your operating system), and type either

```
emacs x.py
```

or

```
emacs -nw x.py
```

The former will create a new Emacs window, where you will have mouse operations available, while the latter will run Emacs in text-only operations in the current window. I'll call the former "GUI mode."

Then type **M-x pdb**, where for most systems "M," which stands for "meta," means the Escape (or Alt) key rather than the letter M. You'll be asked how to run PDB; answer in the manner you would run PDB externally to Emacs (but with a full path name), 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[30] 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

A.3 Debugging with Xpdb

Well, this one is my own creation. I developed it under the premise that PDB would be fine if only it had a window in which to watch my movement through my source code (as with Emacs above). Try it! Very easy to set up and use. Go to `http://heather.cs.ucdavis.edu/~matloff/xpdf.html`.

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

A.4 Debugging with Winpdb (GUI)

The Winpdb debugger (`http://winpdb.org/`),[31] is very good. Its functionality is excellent, and its GUI is very attractive visually.

Among other things, it can be used to debug threaded code, curses-based code and so on, which many debuggers can't.

Winpdb is a GUI front end to the text-based RPDB2, which is in the same package. I have a tutorial on both at `http://heather.cs.ucdavis.edu/~matloff/winpdb.html`.

A.5 Debugging with Eclipse (GUI)

I personally do not like integrated development environments (IDEs). They tend to be very slow to load, often do not allow me to use my favorite text editor,[32] and in my view they do not add much functionality. However, if you are a fan of IDEs, here are some suggestions:

However, if you like IDEs, I do suggest Eclipse, which I have a tutorial for at `http://heather.cs.ucdavis.edu/~matloff/eclipse.html`. My tutorial is more complete than most, enabling you to avoid the "gotchas" and have smooth sailing.

A.6 Some Python Internal Debugging Aids

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

A.6.1 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.6.2 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
```

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

[32] 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.

(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.7 Debugging with PUDB

What a nice little tool! Uses Curses, so its screen footprint is tiny (same as your terminal, as it runs there). Use keys like n for Next, as usual. Variables, stack etc. displayed in right-hand half of the screen.

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 9.1 suppose we stop at the line

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

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

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

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

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

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

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

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

Suppose we want to find out what methods and attributes are associated with strings. As mentioned in Section 7.3, 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 Linux 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 Putting All Globals into a Class

As mentioned in Section 4.4, instead of using the keyword **global**, we may find it clearer or more organized to group all our global variables into a class. Here, in the file **tmeg.py**, is how we would do this to modify the example in that section, **tme.py**:

```
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.

D Looking at the Python Virtual Machine

One can inspect the Python virtual machine code for a program. For the program **srvr.py** in `http://heather.cs.ucdavis.edu/~matloff/Python/PyThreads.pdf`, I once did the following:

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

```
import srvr
```

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

I then ran

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

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

E 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[33]

```
% python x.py
```

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

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

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

```
% x.py
```

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

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

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

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

[33] This section will be Linux-specific.