Minimal Python for Scientific Computing

This document is intended to help non-programmers get started using Python for scientific computing. I don't assume any programming experience – but I do assume that Python and either SciPy or NumPy are already installed on your computer.

This tutorial is designed for participative learning: type the examples into a Python session, think about the output, and try examples as they occur to you. This is not even close to a complete explanation of the language. References to more comprehensive introductions and documentation are shown at the end.

Starting Python

Start an interactive Python session using one of:

Idle	- Windows or Unix
Pythonwin	- Windows
Python IDE	- MacPython under MacOS X

Each opens an interactive Python window which shows the prompt >>>.

Using Python like a calculator

To get a feel for how Python works, try the following. The hash symbol # marks the beginning of a comment in Python. I've used that here to explain some of the commands. Don't type in the comments.

```
4./3.                   # The default floating point type is double precision.
4/3                     # Integer division truncates back to integers
4 / 3.                  # Converts type.
5*3
5.**3
5*3-2
5*(3-2)
10.2 % 3                # Remainder.
x=3.                    # Use of variables is very natural.
y=x**3                  # After assignment (=), Python doesn't show the value.
x,y
print x,y
print X                 # Case sensitive.
Vinitial = 4.           # Use meaningful names (begin with a letter).
x == 1                  # Logic tests return 1 (true) or 0 (false).
x == 3.
```

```
x != y              # Not equal.
x <= y
z=3+2J              # Complex type.
print z
w=2.-2j             # Either J or j works.
z*w
_*2                 # The last computed value is in variable "_".
z.real              # The first hint that Python is object-oriented: real and imag
z.imag              #      are "attributes" of complex numbers.
z.conjugate()       # Objects can have associated functions, called "methods:"
                    #      conjugate() is a method of complex numbers.
```

Idle and Pythonwin have limited command-line editing abilities. Try moving to a higher line using the up arrow, hitting return, editing, and hitting return again. With IDE you have to cut and paste.

Data types

You've seen the simplest numerical types:

```
a=3                 # Integer.
b=4.                # Floating point (double precision).
c=4+2j              # Complex (double precision).
type(a)             # A function that shows the type. Notice that function
                    #    arguments are enclosed within parentheses.
```

A **string** consists of characters within single or double quotation marks:

```
a = "An"
b = 'ex'            # Single or double quotation marks.
c = "parrot"
print a,b,c         # Adds a space between each pair.
print a+b+c         # Concatenates strings.
a[0]                # You can index each character (beginning with zero).
a[1]                # Notice the square brackets.
len(a)              # Length.
```

Python has three other built-in **container** data types besides strings: lists, tuples, and dictionaries. **Lists** are written in square brackets, with the elements separated by commas:

```
x = [ 6,7,8,9,10 ]
type(x)
x[0]                # Lists can be indexed. Notice the square brackets.
x[1]                # The first index is 0, not 1.
len(x)              # Length.
x[-1]               # A negative index begins from the right.
range(5)            # A built-in function which produces a simple list.
range(3,9)
```

Objects which can be indexed (such as lists and strings) can also be accessed by **slices**, which give a range of indices:

```
x[1:3]      # The first index is included but the last is not!
x[:3]       # Dropping one of the indices means "everything in that direction."
x[3:]
x[:]        # Dropping both indices means "everything."
```

The way Python works with ranges takes some getting used to: x[first:last] means the elements x[first], x[first+1],…,x[last-1]. That is, the last element is not included. This means something like "first <= index < last." This same behavior showed up above when you used range.

Lists are **mutable**. That means that elements of the list can be changed while leaving other elements alone.

```
x[0] = 0
print x
```

Lists can be made up of anything, not just numbers.

```
y = [ "Nobody" , "expects" ]
z = [ "the" , "Spanish" ,  "Inquisition" ]
y + z                   # What does addition mean for lists?
```

The second built-in container data type is the **tuple**. These are similar to lists, but without the square brackets:

```
y = 1,2,3
type(y)
print y
y[0]                    # Can be indexed …
y[0:2]                  # … and sliced.
z = (1,2,3)             # The parentheses are optional.
print z
```

Unlike lists, tuples are **immutable**. That means that individual elements cannot be modified.

```
y[0] = 3                # Gives an error.
```

Tuples can be quickly packed and unpacked into other variables:

```
a,b,c = y               # Unpacks y (which can be a tuple or a list) into a,b,c.
print a,b,c
a = 2*a
print a
print y
y = a,b,c               # Packs a,b,c into y (a tuple).
print y
```

Dictionaries are like lists without ordering. They are indexed by "key" rather than position. They are denoted with braces:

```
review = { "Inquisition" : "Funny" , "Parrot" : "Hysterical"}
print review[0]                  # Not indexed by number …
print review['Inquisition']      # … but by key.
print review["Parrot"]
```

Dictionaries are a very convenient way to organize large collections of data, process them, and keep track of everything along the way. You won't usually use them directly in numerical work, but sometimes they are very efficient. (They are used in SciPy to implement sparse matrix operations). Since the entries can be any object (including functions), dictionaries can be remarkably powerful.

Scientific and Numerical Python

The Python core doesn't have the basic data types and functions most needed for scientific programming. Many are available in an add-on module called Numerical Python (or NumPy). This in turn is part of a more comprehensive module called SciPy which includes NumPy plus much more. As of this writing SciPy is only available on Windows and Unix machines. I assume you have already installed one of these.

To get access to these features they need to be imported. Once per session type:

```
from scipy import *              # If SciPy is installed; or
from Numeric import *            # If NumPy (but not SciPy) is installed.
```

You will also need to do this in any module (see below) that needs these features.

NumPy defines a new container data type which is important for scientific programming: the multiarray, or, for short, the **array**. Arrays are similar to lists but have properties closer to matrices than to strings. Define an array using the function array() with (usually) one argument:

```
a = array( (1,2,3) )
type(a)
a
print a
b = array( [1,2,3] )          # Brackets or parentheses are both OK.
print b
c = array( 1,2,3 )            # But you need one or the other.
a / 2                         # Oops … integer arithmetic
a = array( (1.,2,3) )         # Chooses the "bigger" type automatically.
a / 2
a = array((1,2,3),Float)      # Or set the type with an optional second argument.
print a
b = array( [4,5,6] )
a + b                         # Add like matrices.
a=array( [[1,2],[3,4]] )      # 2 by 2 array.
print a
```

Many of the imported numerical functions are "universal functions," or **ufuncs**, which means that they act element-by-element on arrays.

```
a = arange( 10 )          # "array range": more efficient than range().
print a
a**2                      # Squares each element.
a * (pi/9)                # pi is part of NumPy.
sin(a)                    # Takes the sine of each element of a.
sin(2+3j)                 # Works with any data type.
```

Elemental operations are very useful, but make sure that you don't erroneously expect ordinary matrix multiplication.

```
a = array( (1,2) )
print a*2                 # Does what you'd expect.
b = array( (4,5) )
print a*b                 # Elemental multiplication
print dot(a,b)            # What you'd expect.
a = array( ((1,2),(3,4)) )
print a
print b
print matrixmultiply(a,b) # What you'd expect
print a*b
```

This last operation needs explanation. When two arrays don't have compatible shapes, Python tries to broadcast the smaller to the larger. Sometimes that's easy to understand:

```
a = array( ((1,2),(3,4)) )
print a+2
```

Here the scalar 2 is added to each element of the matrix. Now try

```
b = array( (4,5) )
print a+b
```

You might be able to see what happened. The first element of b was added to the first column of a, and the second to the second. The same thing happens with a*b.

Programming

You won't use the interactive window very much. Instead you will write programs in files and then run the programs. A collection of Python code saved in a file is called a module. The Python interfaces have special editors to make this easier. To begin a new module, click File and then New (or New Window). (Using Pythonwin, you will then need to choose "Python Script.") In each case this starts an editor. Type in some very simple Python code, such as:

```
t=2
y=t**2
print y
```

You will of course need to save your modules. Be careful where you put them. Don't use the default location (somewhere down inside the Python directory structure). Use a directory of your own. And use reasonable names (not "Script1"). The details of how to save and run depend on which interface you're using:

- Idle: Save from the File menu. You must explicitly type a suffix .py when you name the module (*e.g.*, "first.py"). Then run by using "Edit->Run Script."

- Pythonwin: You can save either from the File menu or by clicking on the disk icon in the tool bar. Do not explicitly add the suffix .py (*e.g.*, just give it the name "first"). Once it's saved you can run it using "File->Run," or by clicking the little running person icon in the tool bar. These bring up a dialogue box. Click OK and the code runs. (Holding down the shift button while clicking skips the dialogue box.)

- IDE: Save from the File menu. Certain things work better if you add the suffix .py to the name. Run your code by clicking "Run all" on the editor window, or "Python->Run window" from the menu bar.

Change your code and run it again. (For instance, make t=4.) With Idle you need to explicitly save before running again. With Pythonwin the act of running automatically saves the modified version. With IDLE saving is unnecessary (until you're finished!).

Python will complain if you try to run a module containing a syntax error. For example, change the middle line above to y=t***2 and try to run.

The editors can help you find syntax errors. Using Idle, select "File->Check Module." This will highlight the first error it finds. Using Pythonwin, select "File->Check" or click the check mark in the tool bar. The cursor moves to the point where a syntax error is found. With IDE, click the little triangle just below the upper right corner in the editor's window, and then select "Modularize." With any of these, once you fix the error, checking again will find the next error (if any). [Below you will see that editors also help catch syntax errors by adding automatic indentation and (maybe) colors. For example, if you notice that the indentation is wrong in some block, you probably omitted a colon.]

To get all the syntax bugs out of your program: make a change and then check or run. Do it again; and again; and again … until it's perfect.

Control statements

Python has for and while loops, and if statements. The syntax is peculiar: Python uses indentation to determine how far a block extends instead of "end if" or "end for" statements. Make and run a module with the following:

```
x=5
if x<4 or x>27:                 Notice the colon!
        print “Bring”
        print “us”              This line and the above have identical indentation.
print “finished”                Aligned with “if” so outside the block.
```
Change the first line to x=3 and run again.

The comparison operators most often used in if statements are ==, >, >=, <, <=, !=.
Multiple tests use elif (meaning “else if”) and else constructs:

```
x=5
if x<4 or x>27:
        print “Bring”
elif x<7:                       # Notice the colon
        print “us”
elif x<12:
        print “a”
else:
        print “shrubbery”
                                # Can end a block with a blank line.
```
Try changing x to 5, 3, 8, 20, and 30. Make sure you understand the results.

All Python blocks have the same basic structure, a line ending with a colon followed by a block of indented code. The amount of indentation is arbitrary, but be careful to remain consistent or you’ll get an error.

The basic loop is the for loop:

```
for k in [ 1,2,3 ]:             # Notice the keyword “in” and the colon.
        print “k=”,k
```

In the above example the iteration is over the members of a list. In fact, a for loop can iterate over any sequence (a list, tuple, string, or array).

```
for k in [“I” , “feel” , “fine” ]:
        print k

for k in 1,2,3:                 # A tuple
        print k

for k in “swallow”:             # A string
        print k

for k in arange(5):             # An array. To use this in a module you need
        print k                 # to put this as the first line of the module:
                                from Numeric import *

for k in range(5):              # A list
        print k
```

The last two examples are the most important for numerical programming. They show quick ways to iterate over a sequence of numbers. Both arange() and range() permit arbitrary starting points, ending points, and stride. Moreover, in arange() these don't have to be integers:

```
print range(-10,10,2)
for x in arange(-pi/2,pi/2,pi/4):
        print x/pi
```

Notice again that the upper limit in the range is not included.

Another type of loop is provided by while:

```
a,b = 0,1
while b<10:                 # Notice the colon.
        print a,b
        a,b = b,a+b         # Cute multiple assignment.
```

Sometimes within loops you have if tests and, depending on the outcome, want to break out of the loop or instead continue the next iteration. The break statement jumps out of the innermost enclosing loop, and the continue statement continues with the next iteration of the loop.

Functions

We want to write collections of code which can be called with different parameters or arguments, and produce results. In Python these objects are called **functions**, and they are created with the def keyword. Put the following in a module:

```
def square(x):              # Notice the colon
        y = x**2            # and the indentation
        return y
def f(x):
        y = square(x)
        y = y-4
        return y
print square(4)
print f(4)
print square(1)
print f(1)
```

Now run the module. I'm sure you see the point. The return keyword does just what it says: it determines what is returned as the function's value, and causes the return.

Now change the last four lines to the following, and run again.

```
y=4.
print square(y)
print y
```

Notice that the printed y is still 4; it was not changed by calling square. Why? Because the variable y in function square is local to that function, and can't be seen from outside. This is a very important benefit of using functions. You can use whatever variable name you like inside and they won't conflict with names on the outside. (This is close to the truth; a more careful explanation is in the section "Objects" below.)

Functions have a number of oher features (e.g., keywords and default arguments) that are explained in the Python documentation.

Modules

A file containing a collection of function definitions (and also possibly some other code not within a function definition) is known as a **module**. Your programs above were simple modules. Most of Python's features are kept in various modules. You get access to them using the import command. There are three variants to this command. If you type:

```
from scipy import *
```

then you get access to all (well, most) of the functions within SciPy (such as sin, cos, *etc*.). If instead you type:

```
import scipy
```

then you have imported the module called SciPy, but have not imported its function definitions. You could then access the latter with the dotted form:

```
scipy.sin(x)
```

The third form of this command is to import only certain functions. For example,

```
from scipy import sin,cos
```

brings in the two trig functions and nothing else.

To see what objects you have defined (by importing or any other way), use the function dir(). If you have imported a module (using the form import scipy) you can see its associated methods using dir(scipy). (IDE has a nice alternative: click the little triangle just below the upper right corner and select "Browse namespace.") To eliminate some object or binding use del(x). If you've edited a module, re-import it using reload().

When you write simple code it is easiest to keep everything in one file, and simply go through the cycle of edit and run as above. But with larger projects it is useful to separate components into different modules, and import them as necessary.

One thing to bear in mind is that Python has a certain path within which it searches for modules. This is probably Python's worst-implemented feature. The path includes some installation directories, any you've added by hand, and also the "current directory." MacPython adds a simple feature which works around this. Select a folder using the menu item Python->Preferences->Set Script Folder. MacPython will automatically search here for modules when you issue the import command. Pythonwin takes advantage of the notion of the current directory: if you open one of your saved files from some directory or save an edited file to a directory, the directory you use becomes the current directory. Unfortunately, Idle and IDE don't change the current directory for you.

When you set up Python on a machine of your own, you may want to change the default path. On any platform you can create files with suffix .pth in Python's root directory (see the Python documentation).

If you need to, you can see and modify the search path while running Python, as follows.

```
import sys                          # Import the module "sys."
sys.path                            # Shows the search path.
sys.path.append("C:/my directory")
sys.path
```

Here append() is a method of sys.path which appends its argument to the search path. You can also change the current directory by hand:

```
import os
os.chdir("C:/my directory")
```

A complete program

The simplest structure for a scientific or numerical code puts any needed imports at the top, followed by function definitions and then the main executable code. For example:

```
# Example of a complete program.
from scipy import *               # If SciPy is installed.
#from Numeric import *            # Otherwise
def square(x):
    "Squares the argument x"      # A doc string (see below).
    return x**2

# Main routine
days=arange(7)+1
print square(days)
```

This example introduces another important idea: fill your code with comments! Who will read them? You will – next week, or next year, when you come back to this program. If your code has inadequate comments you will have a long and unpleasant time trying to remember how it works. A few minutes adding comments now saves hours later.

Getting help

Python has a few forms of interactive help. The above example introduces the **doc string**. This is the string in the line following the def. If you have been using Pythonwin or Idle, you have seen doc strings frequently. For example, in the interactive window type:

```
sin(
```

The explanation you see is a doc-string. Also try (this works in IDE too):

```
sin.__doc__             # Double underscores on each side.
```

The use of doc strings lets you get some help while running Python. For instance, suppose you're hoping for some kind of modulus function. If you've imported everything within SciPy or NumPy, try dir() or (with IDE) browse the namespace. You'll see a function fmod listed. Then you can examine its doc string as above.

A more elaborate help facility is provided by the module pydoc:

```
import pydoc
pydoc.help()
```

This opens a simple interactive help system. Yet another help facility is provided by SciPy, which has a function help().

Objects

Python is an object-oriented language like C++ or Java. Object orientation, and particularly the ability to define classes, gives the language tremendous flexibility and power. It's not particularly helpful for most numerical programming. But some aspects of Python are confusing until you recognize its object orientation.

To begin with, everything in Python is an **object**. Python objects can have **attributes** and **methods**, which are subsidiary variables and functions. These are accessed using a dot notation. We looked at an example earlier:

```
z = 2+8J
z.real              # An attribute of complex numbers.
z.imag              # Another attribute.
z.conjugate()       # A method of complex numbers.
```

The same dot notation is used for functions that are contained within modules. If you import an entire module (such as import scipy) then you can access its functions using the syntax scipy.sin(3). Thus you can view sin as a method of scipy.

You can make interesting objects with arbitrary attributes and methods using **classes**. That's a big topic which I won't discuss further here.

Everything in Python is an object, and every object can have multiple names **bound** to it. You can see what this means using the id() function. In the interactive window type:

```
x=[1,2,3]       # Creates an object (a list) and binds the name x to the object.
id(x)           # The location of the corresponding object in memory.
y=x
id(y)
```

Notice that x and y have the same id. That means that they are two different names bound to the same object. You can think of y as an alias of x, or as a pointer to the same location. Change an entry in one and you change the other as well.

```
y[0]=4
print x
print y
```

Here we see that an assignment with an index or slice on the left-hand side modifies a mutable object. It does not create a new object, nor does it bind a name to a different object. However if you do an assignment without indices or sections, you create a new object (or change a binding):

```
y = [5,6,7]
id(x),id(y)
print x
print y
```

Here we created a new object (another list) and bound y to it.

In summary, an assignment of the form

```
y=…
```

binds the name y to the object on the right-hand side. (If this object didn't previously exist, this assignment also creates the object.) But an assignment of the form

```
y[1:2]=…,
```

which involves indices or slices on the left, modifies the contents of the mutable object without changing the binding.

If you really want y to be a separate object whose initial value is equal to x, then you can use the copy() method for arrays:

```
y = x.copy()
print id(x),id(y)
y[0]=27
print x,y
```

To copy a list use y=x[:]. Or for any type of object use y=copy.copy(x).

None of this matters for immutable objects. Consider:

```
x=7
y=x
id(x),id(y)
print x,y
y=3
id(x),id(y)
print x,y
```

Again the assignment y=x binds y to the same object as x. But because y is immutable, the only way to change it is a new assignment (such as y=3). This rebinds y rather than modifying it. You can't modify y without rebinding it, so no change to y can change x.

Earlier I said that variables within a function are local to that function. In fact, that's only true if the variable is assigned a value within the function (*e.g.*, z=5). If you do not assign a value within the function, then Python looks in the surrounding module for a variable with the same name, and uses that. If you define z=4 in the module's main part, then print z in an included function will show 4. Thus an unassigned variable z in a function is bound to the same object as z in its enclosing module. But if within the function you assign z=5 then Python rebinds z to be a new local object. (You can override this with

the declaration global z within the function.) Similarly, within a function f(x) the argument x is initially bound to the external argument. An assignment of the form x=2 breaks the binding and produces a new local variable x. And assignments of the form x[1]=2 or z[1]=2 modify the external objects instead of making new local variables. These rather technical ideas fall under the heading of **namespace** and **scope**. More complete explanations are in the documentation (especially the Python Tutorial).

Input and output

You've already learned the simplest way to get output: use the print command. You can get more control over your output using the % operator. For instance:

```
print 'Pi is %5.3f (to three decimal places)' % pi
print 'Pi is %5.3f and pi squared is %5.3f' % (pi,pi**2)
```

The formats are taken from C. Here f is the format for floating-point numbers. Use %s for strings and a form like %3d for integers. (Using %f and %d gives a default spacing.)

Sometimes it's convenient to get interactive input to a routine. One way:

```
x = raw_input("Enter a number:")        # Prompts for value, assigns it to x.
```

Finally I'll briefly describe how read from and write to files. Here I'll assume that the file is in the current directory. (Otherwise you can use an explicit path "C:/mydir/myfile".) The steps are: open the file, read/write using methods, and then close using a method.

```
f=open('myfile','w')                    # Write access to a file myfile
f.write('Heavenly Choir\n')             # Includes newline.
f.write('pi is %5.3f and  pisquare is %5.3f\n' % (pi, pi**2))
f.close()                               # When you're finished, close it.
```

To read from an existing file:

```
f = open('myfile','r')
f.read()                                # Gets the whole thing
f.readline()                            # Instead, read one line.
```

If you want to save a Python object to a file to be read back into Python later, you should learn about the pickle command.

Plotting

There are many ways to generate plots from your Python results. Here I'll mention several. One way is to write data to a file and then use an external plotting program. If you have a package that you like (Origin, Excel, etc.), use it. Here I'll briefly describe how to plot Python results using **gnuplot**, a freeware program available for practically any platform. First, some code to write data to a file (here, a Unix-style path):

```
from Numeric import *
f=open('/Users/Jojo/Python/booplot','w')
for x in arange(0,pi,pi/100):                    # Non-integer limits and stride
        y = sin(x)
        f.write('%8.5f  %8.5f\n' % (x,y))
f.close()
```

Now start up gnuplot. This gives an interactive facility. Type at the prompt as shown.

```
gnuplot>    cd '/Users/Jojo/Python'
gnuplot>    plot 'booplot' with l   (lower-case L, not numeral one)
gnuplot>    set title 'My lovely plot'
gnuplot>    Unset key
gnuplot>    replot
```

The last three lines are an example of gnuplot's many plotting options.

Here is another example of writing to a file which adds a few wrinkles:

```
from scipy import *
x = arange(6)
y = x**2
name = raw_input('Enter filename:')              # Pick name interactively
name = "C:\\mdj\\"+name                          # File is in directory C:\mdj
f=open(name,'w')
for x in arange(len(x)):                         # Loop over (x,y) pairs
        f.write('%5.3  %5.3f\n' % (x[I],y[I]))
f.close()
```

You can also do your plotting entirely within python. I will describe three approaches. The first again uses Gnuplot, but calls it from within Python. I have used this under MacPython, but haven't tried it under Windows. The second two use plotting packages built into SciPy. These work fine under Windows.

If you have installed Gnuplot you can download and install package gnuplot.py into your python directories. (Using this with MacPython requires that you also download an application called AquaTerm.) Here is an example of its use:

```
from Numeric import *
import Gnuplot,Gnuplot.funcutils
x=arange(0,pi,pi/10.)
y1=sin(x)
y2=cos(x)
g=Gnuplot.Gnuplot(debug=1)
g('set terminal aqua')                  # Needed with MacPython
g.title('My beautiful plot')            # optional
g('set data style linespoints')         # can give gnuplot an arbitrary command
d1=Gnuplot.Data(x,y1,with='line 4',title='sin(x)')
d2=Gnuplot.Data(x,y2,title='cos(x)')
g.plot(g1,g2)
```

The gnuplot.py installation includes a file demo.py which contains useful examples. Please examine it.

The other two plot packages that I will describe are part of SciPy: plt and gplt. (I haven't used a third package in SciPy, xplt,) First look at plt. A note of caution: this works well with Pythonwin but crashes Idle. Once per session (or once per module) you need to import two packages (gui_thread and plt). Then use method plt.plot:

```
from scipy import *
import gui_thread
from scipy import plt        # Didn't come in with the *
x=arange(0,pi,pi/100)
y=sin(x)                     # Elemental operation
plt.plot(x,y)
plt.title("My lovely plot")
```

Here are examples using package gplt.

```
# Put two curves on one figure
from scipy import *
import gui_thread
from scipy import gplt              # Didn't come in with the *
x=arange(0,pi,pi/100)
y1=sin(x)                           # Elemental operation
y2=cos(x)
gplt.plot(x,y1)
gplt.hold('on')
gplt.plot(x,y2)

# Alternatively you could replace the last three lines by:
gplt.plot(x,y1,x,y2)

# Put two curves on separate figures:
gplt.figure()
gplt.plot(x,y1)
gplt.figure()
gplt.plot(x,y2)
```

Error messages

You have been getting two different kinds of error messages: **syntax errors**, and run-time **exceptions**. A syntax error means you have typed some illegal Python code. An exception means that execution caused an error (for instance, division by zero). It can take a while to learn what error messages mean.

What next?

Now you should look through more complete Python documentation. Many general introductions and reference materials are available at www.python.org. I like the tutorial written by Python's creator, Guido van Rossum. See www.python.org/doc/current.

A rapid introduction to many of Python's more advanced features is *Dive into Python*, found at diveintopython.org.

The numerical package NumPy is documented at www.numpy.org. Likewise the scientific package SciPy is described at www.scipy.org.

There are a number of books available. I enjoyed *Python: The Complete Reference*, by Martin C. Brown (Osborne/McGraw-Hill).