Python\_Scripting\_Computational\_Science\_C03

Chapter 3 Basic Python

The present chapter provides an overview of functionality frequently needed in Python scripts, including file reading and writing, list and dictionary oper- ations, simple text processing, writing and calling Python functions, checking a file’s type, size, and age, listing and removing files, creating and removing directories, and traversing directory trees. In a sense, the overview is a kind of quick reference with embedded examples containing useful code segments in Python scripts. A corresponding overview of more advanced Python func- tionality is provided in Chapter 8. For real, complete quick references, see links in doc.html.

The many Python modules developed as part of this book project, and referred to in this and other chapters, are collected in a package scitools. This package must be downloaded and installed (by running a setup.py script) as described in Chapter 1.2. The various modules in scitools are accessible through the dot notation, e.g., scitools.misc denotes the misc module within the scitools package. Many of the functions referred to in the forthcoming sections are found in the misc module.

Lots of examples are from now on presented in interactive mode (see Chapter 2.2.6) such that it is easy to see the result of Python expressions or the contents of variables. According to the tradition in the Python literature, we prefix interactive Python commands with the prompt >>>, while output lines have no prefix. Continuation of an input line is indicated by the ... prompt:

>>> x = 0.1

>>> def f(x):

... return math.sin(x) ...

>>> f(x) 0.099833416646828155

Note that these interactive sessions look different in IPython, because the prompt is different, but the input and output are the same.

Introductory Topics

Some recommended Python documentation to be used in conjunction with the presented book is mentioned in Chapter 3.1.1. Chapter 3.1.2 lists the syntax of basic contol statements in Python: if tests, for loops, while loops, and the break and continue statements. Running stand-alone programs (or operating system commands in general) is the focus of Chapter 3.1.3. A summary of basic file reading and writing is listed in Chapter 3.1.4, while controlling the output format, especially in text containing numbers, is the subject of Chapter 3.1.5.

3.1.1 Recommended Python Documentation

The exposition in this book is quite brief and focuses on “getting started” examples and overview rather than in-depth treatment of language-specific topics. In addition to the book you will therefore need complete references to Python programming.

The primary Python reference is the official Python documentation to which you can find relevant links in the file doc.html (the file comes with the software associated with this book, see Chapter 1.2). The documents are available as web pages and as printable PDF/PostScript files. Of particular importance in the official documentation is the Python Library Reference [37]. The doc.html file contains a useful link to the index of this reference. The reader is strongly encouraged to become familiar with the Python Li- brary Reference. The official Python documentation also contains a Python Tutorial [38] with an overview of language constructs. The doc.html has a link to a handy facility for searching the documents in the electronic Python documentation.

Another important documentation is pydoc, which comes with the stan- dard Python distribution. Writing pydoc X on the command line brings up the documentation of any module or function X that Python can find, includ- ing your own modules. The pydoc documentation is slightly different from the Python Library Reference. Contrary to the latter, pydoc always lists all classes and functions found in a module.

Beazley’s Python reference book [2] extends the material in the Python Library Reference and is highly recommended. An excellent and more com- prehensive reference book is Martelli’s “Python in a Nutshell” [22]. An even more voluminous reference is [3] by Brown. A slimmer alternative, focusing on Python’s standard library modules, is Lundh [18]. Windows users may find “Python Programming on Win 32” [11] helpful. Many programmers find quick references very handy: the pocket book [19] or the electronic quick references to which there is a link in in doc.html.

A recommended textbook on the Python language, which also covers some advanced material, is the “Quick Python Book” [12]. The “Learning Python” book [21] represents an alternative tutorial. The treatment of GUI building with Python in these books is quite limited, but there is fortunately a com- prehensive textbook [10] devoted to creating professional GUIs with Python. More advanced aspects of Python are very well treated in the second edition of “Programming Python” [20]. A fairly complete collection of Python books is available from the Python home page www.python.org.

3.1.2 Control Statements

If Tests and True/False Expressions. The if-else statement can be illus- trated as follows:

if answer == ’copy’: copyfile = ’tmp.copy’

elif answer == ’run’ or answer == ’execute’: run = True

elif answer == ’quit’ and not eps < eps\_crit: quit = True

else:

print ’Invalid answer’, answer

The test if var returns false if var is None, a numeric type with value 0, a boolean with value False, an empty string (’’), an empty list ([]), an empty tuple (()), or an empty dictionary ({}). Otherwise, the if test is true.

For Loops. Looping over a list is done with the for statement: for arg in sys.argv[1:]:

# work with string arg

An explicit integer index can also be used:

for i in range(1, len(sys.argv), 1): # work with string sys.argv[i]

More advanced for loops are covered in Chapter 3.2.4.

While Loops. The syntax of a while loop is illustrated next:

r=0; dr=0.1 while r <= 10:

print ’sin(%.1f)=%g’ % (r, math.sin(r)) r += dr

The range function only generates integers so for loops with a real number counter are better implemented as while loops (which was illustrated above for a counter r running as 0, 0.1, 0.2, . . . , 9.9, 10).

The while var condition evaluates to true or false in the same way as the if var test.

Break and Continue for Modified Loop Behavior.

The break statement breaks

out of a loop:

f = open(filename, ’r’) while 1:

line = f.readline()

if line == ’’:

break

# process line

...

# empty string means end of file # jump out of while loop

With continue the program continues with the next iteration in the loop:

files = os.listdir(os.curdir) # all files/dirs in current dir. for file in files:

if not os.path.isfile(file):

continue # not a regular file, continue with next

<process file>

3.1.3 Running Applications

A simple way of executing a stand-alone application, say

cmd = ’myprog -c file.1 -p’ # run application myprog

or any operating system command cmd, is to employ the technique used in

the simviz1.py script from Chapter 2.3.5:

failure = os.system(cmd) if failure:

print ’Execution of "%s" failed!\n’ % cmd, output sys.exit(1)

Frequently, one needs to capture the output of the program that is exe- cuted. A technique is to open a pipe and get a file-like object for the output that we can read from:

output = os.popen(cmd) lines = output.readlines() failure = output.close() if failure:

print ’Execution of "%s" failed!\n’ % cmd, output sys.exit(1)

The lines list contains the lines written to standard output by the executed program. We can process the output in the usual way:

for line in lines:

# process the string line, representing a line of output

A quicker approach, limited to Unix systems, is to use the following construc- tion:

import commands

failure, output = commands.getstatusoutput(cmd) if failure:

print ’Execution of "%s" failed!\n’ % cmd, output sys.exit(1)

The returned output variable is a string, not a list of lines, and it contains all containing the text written by the command to both standard output and standard error. Processing this output can be done by

lines = output.splitlines() for line in lines:

# process line

The scitools.misc module has a function system that encapsulates an operating system call, captures its output, and performs various actions (sys.exit, raise exception, print warning, or continue silently) in case of fail- ure. This function can save quite some typing in scripts with many operating system calls.

Python versions older than 2.4 had several tools for executing operat- ing system commands (the commands and popen2 modules and functions like os.system, os.popen\*, os.spawn\*, etc.). These tools are now replaced by the subprocess module. The standard way of executing an application without capturing its output is to use the call function:

from subprocess import call try:

returncode = call(cmd, shell=True) if returncode:

print ’Failure with returncode’, returncode; sys.exit(1) except OSError, message:

print ’Execution failed!\n’, message; sys.exit(1)

More advanced use of subprocess employs its Popen object. For example,

capturing the output of a command is done by:

from subprocess import Popen, PIPE

p = Popen(cmd, shell=True, stdout=PIPE) output, errors = p.communicate()

Here, output and errors are strings containing standard output and standard error, respectively.

To feed data to an application, we can use a redirection of standard input to a file:

cmd = ’myprog -c file.1 -p < input\_file’

Alternatively, we can use Popen to feed data from the Python script to the application. Here is an an example on how to instruct the interactive Gnuplot program to draw a sine function in a plot window1:

1 ThisexampledoesnotworkonWindowsbecausetheWindowsversionofGnuplot uses a GUI instead of standard input to fetch commands.

pipe = Popen(’gnuplot -persist’, shell=True, stdin=PIPE).stdin pipe.write(’set xrange [0:10]; set yrange [-2:2]\n’) pipe.write(’plot sin(x)\n’)

pipe.write(’quit’) # quit Gnuplot

Sometimes it is desirable to establish a two-way communication with an external application, i.e., we want to pipe data to the application and record the application’s response. For this purpose the pexpect module is recom- mended (rather than subprocess.Popen which may easily hang in two-way communications). With pexpect (see doc.html for a link) it becomes possible to automate execution of interactive programs.

The statement after an operating system command is not executed before the operating system command has terminated. If the script is supposed to continue with other task while the application is executing, one must run the application in the background. This is enabled by adding an ampersand & on Unix or begin the command with start on Windows. Coding of such platform-specific actions is exemplified on page 323. An alternative solution is to use threads (see Chapter 8.5.4) for running a system command in parallel with the script. The simplest approach may look like this:

import threading

t = threading.Thread(target=os.system, args=(cmd,)) t.start()

To capture the output, one has to use os.popen or other techniques from Chapter 3.1.3 and derive a subclass of Thread, where the output is grabbed in the run method, see Chapter 8.5.4 for details.

3.1.4 File Reading and Writing

Here are some basic Python statements regarding file reading: infilename = ’.myprog.cpp’

infile = open(infilename, ’r’) # open file for reading

# read the file into a list of lines: lines = infile.readlines()

for line in lines:

# process line

# read the file line by line: for line in infile:

# process line

# alternative reading, line by line: while 1:

line = infile.readline()

if not line: break

# process line

load the file into a string instead: filestr = infile.read()

# read n characters (bytes) into a string: chunck = infile.read(n)

infile.close()

The for line in infile construction is fine when we want to pass through the whole file in one loop. The classical Python construction with an “infinite” while loop and a termination criterion inside the loop is better suited when different chunks of the file require different processing.

In case you open a non-existing file, Python will give a clear error message, see the opening of Chapter 8.8.

Reading from standard input is like reading from a file object, and the name of this object is sys.stdin. There is, of course, no need to open and close sys.stdin. Reading data from the keyboard is normally done by the obvi- ous command sys.stdin.readline(), or by the special function raw\_input(). With sys.stdin.read() one can read several lines, terminated by Ctrl-D.

Basic file writing is illustrated by the following code segment:

outfilename = ’.myprog2.cpp’

outfile = open(outfilename, ’w’) # open file for writing line\_no = 0 # count the line number in the output file for line in list\_of\_lines:

line\_no += 1

outfile.write(’%4d: %s’ % (line\_no, line)) outfile.close()

Writing of a string is performed with write, whereas writing a list of lines is performed with writelines:

outfile.write(some\_string) outfile.writelines(list\_of\_lines)

One can of course append text to a new or existing file, accomplished by the string ’a’ as the second argument to the open function. Below is an example of appending a block of text using Python’s multi-line (triple quoted) string:

outfile = open(outfilename, ’a’) # open file for appending text outfile.write("""

/\*

This file, "%(outfilename)s", is a version

of "%(infilename)s" where each line is numbered \*/

""" % vars())

For printing to standard output, one can use print or sys.stdout.write. The sys.stdout object behaves like an ordinary file object. The print function can also be used for writing to a file:

f = open(’somefile’, ’w’) print >> f, ’text...’

Python 2.6 offers an alternative construction for reading and writing files, using the new with statement. Until version 2.6 becomes available, one can make the with keyword available by writing

from \_\_future\_\_ import with\_statement

File reading can be done like this:

with open(somefile, ’r’) as f: for line in f:

<process line>

When the execution leaves the with block the f file object is automatically

closed.

3.1.5 Output Formatting

The following interactive Python shell session exemplifies alternative ways of controlling the output format:

>>> r = 1.2

>>> s = math.sin(r)

>>> # print adds a space between comma-separated arguments: >>> print "sin(", r, ")=", s

sin( 1.2 )= 0.932039085967

>>> # use + between the strings to avoid any extra space: >>> print ’sin(’ + str(r) + ’)=’ + str(s) sin(1.2)=0.932039085967

>>> # format control via the printf-like syntax: >>> print "sin(%g)=%12.5e" % (r,s)

sin(1.2)= 9.32039e-01

>>> # format control via variable interpolation: >>> print ’sin(%(r)g)=%(s)12.5e’ % vars() sin(1.2)= 9.32039e-01

Instead of print you can write to sys.stdout in the same way as you write to file objects:

sys.stdout.write(’sin(%g)=%12.5e\n’ % (r,s))

Note that write does not add a newline, whereas print adds a newline unless you end the print statement with a comma.

There are numerous specifications of a format string. Some examples are listed below.

an integer written in a field of width 5 chars an integer written in a field of width 5 chars, but adjusted to the left

an integer written in a field of width 5 chars, padded with zeroes from the left (e.g. 00041)

a float variable written in %f or %e notation

a float variable written in scientific notation

as %e, but upper case E is used for the exponent as %g, but upper case E is used for the exponent

a float variable written in scientific notation with 3 decimals in a field of width 11 chars

a float variable written in scientific notation with 3 decimals in a field of minimum width

a float variable written in fixed decimal notation with 1 decimal in a field of width 5 chars

a float variable written in fixed decimal form with 3 decimals in a field of minimum width

%d : an integer

%s : a string

%-20s : a string adjusted to the left in a field of

width 20 chars

The %s format can in fact be used for any variable x: an automatic string conversion by str(x) is performed if x is not a string.

For a complete specification of the possible printf-style format strings, follow the link from the item “printf-style formatting” in the index of the Python Library Reference. Other relevant index items in this context are “vars” and “string formatting”. See also Chapter 8.7.

Variable interpolation does not work with list or dictionary entries, e.g.,

’a[%(i)d]=%(a[i])g’ % vars() # illegal!

In this case you need to apply the printf-style formatting

’a[%d]=%g’ % (i, a[i])

We mention here that there is a Python module Itpl15 (available on the Internet), which offers the same type of interpolation as in Perl. That is, one can work with expressions like ’a[$i]=$a[i]’ in the previous example.

3.2 Variables of Different Types

The next sections describe basic operations with variables of Python’s most common built-in types. Chapter 3.2.1 deals with boolean variables, Chap- ter 3.2.2 with the handy None variable, and Chapter 3.2.3 discusses use of numbers, i.e, integers, floating-point variables, and complex variables. Fre- quent operations on lists and tuples are listed in Chapter 3.2.4, while Chap- ter 3.2.5 addresses frequent operations on dictionaries. Chapters 3.2.6–3.2.8 deal with strings, including split and join operations, text searching, text

substitution, and an overview of common regular expression2 functionality. User-created variable types, defined through classes, are outlined in Chap- ter 3.2.9, while more details of class programming are left for Chapter 8.6. Examination of what b = a really means and how to copy objects in vari- ous ways constitute the contents of Chapter 3.2.10. Finally, Chapter 3.2.11 explains how one can determine the type of a given variable.

3.2.1 Boolean Types

Originally, Python used integers (as in C) to represent boolean values: 0 cor- responds to false, while all other integer values are considered true. However, it is good programming practice to limit an integer’s values in a boolean context to 0 and 1.

Recent Python versions offer a special boolean type, bool, whose values are True or False. These values can be interchanged with 1 and 0, respectively. The script src/py/intro/booldemo.py demonstrates how True and False can be interchanged with integers.

3.2.2 The None Variable

Python defines a special variable None denoting a “null object”, which is convenient to use when a variable is available but its value is considered “undefined”:

answer = None

<may update answer from other data...> if answer is None:

quit = True

elif answer == ’quit’:

quit = True

else:

quit = False

To check if a variable answer is None or not, always use if answer is None or if answer is not None. Testing just if not answer is dangerous, because the test is true if answer is an empty string (or empty list, dictionary, etc., see pages 75 and 392), although it is also true if answer is None.

At this point we might mention the difference between the is and == operators: is tests for object identity, while == tests if two objects have the same value (i.e., the same content). There is only one instance of the null object None so if answer is None tests whether answer is the same object as the null object. With if answer == None we test if the value of answer is the same as the value of the null object (and that works well too). Chapter 3.2.10 has several examples on the difference between the is and == operators.

2 Regular expressions are introduced and explained in detail in Chapter 8.2.

Instead of using None to mark a variable as “undefined”, we may set the variable to an empty object of the appropriate kind and test if the variable is true, see page 75.

3.2.3 Numbers and Numerical Expressions

There are four built-in numeric types in Python:

* – Integers of type int: 0, 1, -3.
* – Long integers of type long: 0L, 1L, -3L. These integers can have arbitrary length.

* – Double precision real numbers of type float: 0., .1, -0.0165, 1.89E+14.
* – Double precision complex numbers of type complex: 0j, 1+.5j, -3.14-2j

(j denotes the imaginary unit √−1).

Python’s int and float correspond to long int and double in C.

The real and imaginary parts of a complex variable r are obtained by r.real and r.imag, respectively (these are float variables). The cmath module implements the mathematical functions in math for complex types. The next

function works with cmath and complex numbers:

def roots(a, b, c): """

Return two roots of the quadratic algebraic equation ax 2 + bx + c = 0, where a, b, and c may be complex. """

import cmath # complex functions

q = b\*b - 4\*a\*c

r1 = -(b - cmath.sqrt(q))/(2\*a)

r2 = -(b + cmath.sqrt(q))/(2\*a)

# r1 and r2 are complex because cmath.sqrt returns complex,

# convert to real if possible:

if r1.imag == 0.0: r1 = r1.real

if r2.imag == 0.0: r2 = r2.real

if r1 == r2: r2 = None # use r2=None to indicate double root return r1, r2

This code can be made more compact if we utilize the smarter sqrt func- tion from SciPy (Chapter 4.4.2). That implementation of sqrt transparently returns a float or a complex number, dependent on the argument3:

def roots(a, b, c):

from scipy import sqrt q = b\*b - 4\*a\*c

3 Note the 2.0 factor when q==0 to ensure floating-point division. With just 2, the fraction implies integer division if a and b are given as integers, cf. page 84. The general root expressions have a sqrt call that returns float, which ensures correct float division.

if q == 0:

return -b/(2.0\*a), None

else:

return -(b - sqrt(q))/(2\*a), -(b + sqrt(q))/(2\*a)

Python supports the same numerical expressions as C. Programmers being used to Perl or Tcl should notice that strings are not automatically trans- formed to numbers when required. Here is a sample code:

b= 1.2

b = ’1.2’

a = 0.5 \* b

a = 0.5 \* float(b) # this works

#bisanumber

# b is a string

# illegal: b is not converted to a real number

Number comparisons can easily confuse you if you happen to mix strings and numbers. Suppose you load sys.argv[1] into a variable b and that 1.2 was supplied as the first command-line argument. The test b < 100.0 is then false: b is a string, and we compare a string and a floating-point number. No error messages are issued in this case, showing how important it is to explicitly convert input strings to the right type, here b=float(sys.argv[1]).

In Python, any type of objects (numbers, strings, user-defined classes, etc.) are compared using the standard operators ==, !=, <, <=, and so on. In many other dynamically typed languages, such as Perl, Tcl, and Bash, different operators are used for comparing numbers and strings.

Conversion between strings and numbers can be performed as exemplified below.

>>> s = ’13.8’

>>> float(s)

13.800000000000001

>>> int(s)

ValueError: invalid literal for int(): 13.8 >>> f = float(s)

>>> int(f) # truncate decimals 13

>>> complex(s)

(13.800000000000001+0j)

>>> # convert float to string (three different alternatives): >>> ’%(f)g’ % vars(), ’%g’ % f, str(f)

(’13.8’, ’13.8’, ’13.8’)

Python programmers must be very careful with mathematical expressions involving integers and the division operator. As in many other languages, division of two integers implies integer division, i.e., for integers p and q, p/q is the largest integer that when multiplied by q becomes less than or equal to p.

* >>> p=3; q=6
* >>> p/q 0

>>> float(p)/q

0.5



# define two integers

# Python applies integer division

# one float operand yields float division

>>> from \_\_future\_\_ import division # turn off integer division >>> p/q # now this is float division

0.5

Integer division is a common source of error in numerical codes.

3.2.4 Lists and Tuples

Python lists can contain numbers, strings, and any other data structures in an arbitrarily nested, heterogeneous fashion. A list is surrounded by square brackets, and items are separated by commas, e.g.,

arglist = [myarg1, ’displacement’, "tmp.ps"]

Note that myarg1 can be of any type, not necessarily a string as the two other items.

Python has in some sense two types of lists: ordinary lists enclosed in brackets,

[item1, item2, ...]

and tuples enclosed in standard parenthesis: (item1, item2, ...)

The parenthesis can sometimes be left out. This will be illustrated in forth- coming examples.

Empty lists and tuples are created by

mylist = []

mytuple = ()

Ordinary lists are mutable, meaning that the contents can be changed in-place. This makes lists behave like ordinary arrays known from Fortran or C-like languages:

words = [’tuple’, ’rhymes with’, ’couple’]

words[1] = ’and’ # can change the second list item

Tuples are immutable objects whose contents cannot be altered: words = (’tuple’, ’rhymes with’, ’couple’)

words[1] = ’and’ # illegal - Python issues an error message

Numbers, strings, and tuples are immutable objects while lists, dictionaries, and instances of user-defined classes are mutable.

Tuples with One Item. A trailing comma is needed after the element in tuples that has one element only, e.g., mytuple=(str1,). Without the comma, (str1) is just a variable enclosed in parenthesis, and mytuple just becomes

a reference to str1. If you want mytuple to be a tuple, you need the trailing comma. On the other hand, declaring a list with a single item needs no comma, e.g., mylist=[str1], but a comma does not harm: mylist=[str1,].

Adding, Indexing, Finding, and Removing List Items. Adding an object myvar2 to the end of a list arglist is done with the append function:

arglist.append(myvar2)

Extracting list or tuple items in separate variables can be done through these constructions:

[filename, plottitle, psfile] = arglist # or with tuples:

(filename, plottitle, psfile) = arglist

filename, plottitle, psfile = arglist

The arglist variable is a list or tuple and must in this case have exactly three items, otherwise Python issues an error. Alternatively, one can use explicit indexing:

filename = arglist[0] plottitle = arglist[1] psfile = arglist[2]

Searching for an item ’tmp.ps’ and deleting this item, if arglist is a list, can be done with

i = arglist.index(’tmp.ps’) # find index of the ’tmp.ps’ item del arglist[i] # delete item with index i

or simpler

arglist.remove(’tmp.ps’) # remove item with value ’tmp.ps’

The in operator can be used to check if a list or tuple contains a specific

element

if file in filelist:

# filelist contains file as an item

More complete documentation of list functions is found by following the index link “list type, operations on” in the Python Library Reference. The index “tuple object” leads to an overview of legal operations on tuples.

Iterating over Lists. A loop over all items in a list or tuple is expressed by the syntax

for item in arglist:

print ’item is ’, item

This is referred to as iterating over a list or tuple in Python terminology. One can also iterate over a list or tuple using a C-style for loop over the array indices:

start = 0; stop = len(arglist); step = 1 for index in range(start, stop, step):

print ’arglist[%d]=%s’ % (index, arglist[index])

Here we must emphasize that stop-step is the maximum index encountered in this loop. As another example, the sequence 1,3,5,7,9 must be generated by a call range(1,10,2). A single argument in range is also possible, implying start at 0 and unit increment:

for index in range(len(arglist)):

print ’arglist[%d]=%s’ % (index, arglist[index])

We remark that Python data structures are normally not printed by explic- itly looping over the entries. Instead you should just write print arglist, and the output format is then valid Python code for initializing a list or a tuple, cf. Chapter 8.3.1. The loop above is convenient, however, for explicitly displaying the index of each list item.

The range function returns a list of integers, so for very long loops range may imply significant storage demands. The xrange function is then an alter- native. It works like range, but it consumes less memory and CPU time (see footnote on page 138).

Iterating over several lists or tuples simultaneously can be done using a loop over a common index,

for i in range(len(xlist)):

x = xlist[i]; y = ylist[i]; z = zlist[i]

# or more compactly: x, y, z = xlist[i], ylist[i], zlist[i] # work with x, y, and z

A shorter and more Pythonic alternative is to apply the zip function: for x, y, z in zip(xlist, ylist, zlist):

# work with x, y, and z

The size of this loop equals the length of the shortest list among xlist, ylist, and zlist.

List items can be changed in-place:

for i in range(len(A)):

if A[i] < 0.0: A[i] = 0.0

Now there are no negative elements in A. The following construction does not work as intended4:

for r in A:

if r < 0.0: r = 0.0

4 The similar construction in Perl changes the list entries, a fact that might be confusing for Python programmers with a background in Perl.

Here r refers an item in the list A, but then we assign a new float object to r. The corresponding list item is not affected (see Chapter 3.2.10 for more material on this issue).

Compact Item-by-Item Manipulation of Lists. Occasionally, one wants to manipulate each element in a list by a function. This can be compactly per- formed by list comprehensions. A common example may be5

y = [float(yi) for yi in line.split()]

Here, a string line is split into a list of words, and for each element yi in this list of strings, we apply the function float to transform the string to a floating-point number. All the resulting numbers are then formed as a list, which we assign to y.

The same task can also be carried out using the map function: y = map(float, line.split())

Again, float is applied to each element in the line.split() list to form a new list.

In general, we may write

new\_list = [somefunc(x) for x in somelist] # or

new\_list = map(somefunc, somelist)

The somefunc function may be user defined, and its return value yields the corresponding list element. With list comprehensions we can also have an expression with the loop iterator instead of a call like somefunc(x). Here is an example where we create n+1 coordinates xi = a+ih, h = 1/(n−1), i = 0,...,n:

>>> a = 3.0; n = 11; h = 1/float(n-1) >>> x = [ a+i\*h for i in range(n+1) ]

List comprehensions may contain any number of nested lists, combined with conditional expressions if desired:

>>> p = [(x,y) for x in range(-3,4,1) if x > 0 \ for y in range(-5,2,1) if y >= 0]

>>> p

[(1, 0), (1, 1), (2, 0), (2, 1), (3, 0), (3, 1)]

We refer to the chapter “Data Structures”, subsection “List Comprehen- sions”, in the electronic Python Tutorial for more documentation on list comprehensions.

The map function can do more than exemplified here, see the Python Library Reference (index “map”). Expressions, such as a+i\*h in the previous

5 This construction is used to read numbers from file in the convert2.py script from Chapter 2.5.

example, must be implemented via lambda constructions (see page 117) in conjunction with the map operation.

Nested Lists. Nested lists are constructed and indexed as exemplified in the following code segment:

# curves1 is a list of filenames and lists of (x,y) tuples: curves1 = [’u1.dat’, [(0,0), (0.1,1.2), (0.3,0), (0.5,-1.9)],

x\_coor

file

points

’H1.dat’, xy1] # xy1 is a list of (x,y) tuples

= curves1[1][2][0] # yields 0.3

= curves1[2] # yields ’H1.dat’

= curves1[1] # yields a list of points (x,y)

We see that curves1 is a list of different data types. Determining an item’s type in heterogeneous lists or tuples is frequently needed, and this is covered in Chapter 3.2.11. Now we know that curves1[1] is a list of 2-tuples, and iterating over this list can be done conveniently by

for x,y in curves1[1]:

# yields x=0, y=0, then x=0.1, y=1.2, and so on

Let us reorganize the curves1 list to be a list of filename–points pairs: curves2 = [[’u1.dat’, [(0,0), (0.1,1.2), (0.3,0), (0.5,-1.9)]],

[’H1.dat’, xy1]] # xy1 is a list of (x,y) tuples Suppose we want to dump the list of points in curves2 to the files u1.dat and

H1.dat. With the new organization of the data this is elegantly performed by

for filename, points in curves2:

f = open(filename, ’w’)

for x,y in points: f.write(’%g\t%g\n’ % (x,y)) f.close()

This type of attractive iteration over nested data structures requires that each single list has elements of the same type. The curves2 list fulfills this requirement, and it can therefore be argued that the design of curves2 is better than that of curves1.

Slicing. Python has some convenient mechanisms for slicing list and tuple structures. Here is a demo session from the interactive Python shell:

>>> a = ’demonstrate slicing in Python’.split() >>> print a

[’demonstrate’, ’slicing’, ’in’, ’Python’]

>>> a[-1] # the last entry

’Python’

>>> a[:-1] # everything up to but, not including, the last entry

[’demonstrate’, ’slicing’, >>> a[:] # everything [’demonstrate’, ’slicing’, >>> a[2:] # everything [’in’, ’Python’]

’in’]

’in’, ’Python’]

from index 2 and upwards

>>> a[-1:] # the last entry

[’Python’]

>>> a[-2:] # the last two entries

[’in’, ’Python’]

>>> a[1:3] # from index 1 to 3-1=2

[’slicing’, ’in’]

>>> a[:0] = (’here we’).split() # add list in the beginning >>> print a

[’here’, ’we’, ’demonstrate’, ’slicing’, ’in’, ’Python’]

The next session illustrates assignment and slicing:

>>> a = [2.0]\*6 # create list of 6 entries, each equal to 2.0 >>> a

[2.0, 2.0, 2.0, 2.0, 2.0, 2.0]

>>> a[1] = 10

>>> b = a[:3]

>>> b

[2.0, 10, 2.0]

>>> b[1] = 20

>>> a

# a[1] becomes the integer 10

# is a[1] affected?

[2.0, 10, 2.0, 2.0, 2.0, 2.0] # no b is a copy of a[:3]

>>> a[:3] = [-1] # first three entries are replaced by one entry >>> a

[-1, 2.0, 2.0, 2.0]

These examples show that assignment to a slice is an in-place modification of the original list, whereas assignment of a slice to a variable creates a copy of the slice.

Reversing and Sorting Lists. Reversing the order of the entries in a list mylist is performed by

mylist.reverse()

Sorting a list mylist is similarly done with mylist.sort()

We remark that reverse and sort are in-place operations, changing the se- quence of the list items. In Python2.4 a new function sorted appeared, which returns a copy of a sorted sequence:

newlist = sorted(mylist)

By default, the sort and sorted functions sort the list using Python’s com- parison operators (<, <=, >, >=). This means that lists of strings are sorted in ascending ASCII order, while list of numbers are sorted in ascending numeric order. You can easily provide your own sort criterion as a function. Here is an example:

def ignorecase\_sort(s1, s2):

# ignore case when sorting

s1 = s1.lower(); s2 = s2.lower()

if s1 < s2:

elif s1 == s2:

else

return -1

return 0

return 1

# or an equivalent, shorter function, using the built-in # comparison function cmp:

def ignorecase\_sort(s1, s2):

return cmp(s1.lower(), s2.lower())

# apply the ignorecase\_sort function: mylist.sort(ignorecase\_sort)

newlist = sorted(mylist, ignorecase\_sort)

A function consisting of a single expression, like cmp(...), can be defined as an anonymous inline function using the lambda construct (see page 117):

mylist.sort(lambda s1, s2: cmp(s1.lower(), s2.lower()))

Remark. List copying and list assignment are non-trivial topics dealt with in Chapter 3.2.10.

3.2.5 Dictionaries

A dictionary, also called hash or associative array in other computer lan- guages, is a kind of list where the index, referred to as key, can be an arbitrary text6. The most widely used operations on a dictionary d are

d[’dt’]

d.keys()

d.has\_key(’dt’)

’dt’ in d

’dt’ not in

d.get(’dt’,

d.items()

d.update(q)

del d[’dt’]

len(d)

# extract item corresponding to key ’dt’ # return copy of list of keys

# does d have a key ’dt’?

# same test as d.has\_key(’dt’)

# same test as not d.has\_key(’dt’)

# as d[’dt’] but a default value 1.0 is

# returned if d does not have ’dt’ as key # return list of (key,value) tuples

# update d with (key,value) from dict q

# delete an item

# the number of items

d 1.0)

**Example**. Now we present an example showing the convenience of dictionar- ies. All parameters that can be specified on the command line could be placed in a dictionary in the script, with the name of the option (without the hyphen prefix) as key. Hence, if we have two options -m and -tstop, the corresponding parameters in the program will be cmlargs[’m’] and cmlargs[’tstop’].

Initializing items in a dictionary is done by

cmlargs = {} # initialize as empty dictionary cmlargs[’m’] = 1.2 # add ’m’ key and its value cmlargs[’tstop’] = 6.0

6 In fact, a key in a Python dictionary can be any immutable object! Strings, numbers, and tuples can be used as keys, but lists can not.

Alternatively, multiple (key,value) pairs can be initialized at once:

cmlargs = {’tstop’: 6.0, ’m’: 1.2} # or

cmlargs = dict(tstop=6.0, m=1.2)

With such a dictionary we can easily process an arbitrary number of command- line arguments and associated script variables:

# loop through the command-line options

# (assumed to be in pairs: -option arg\_counter = 1

while arg\_counter < len(sys.argv):

option = sys.argv[arg\_counter] if option[0] == ’-’: option = else:

value or --option value)

option[1:] # remove 1st hyphen

# not an option, proceed with next sys.argv entry

arg\_counter += 1; continue

if option[0] == ’-’: option = option[1:] # remove 2nd hyphen

if option in cmlargs:

# next command-line argument is the value: arg\_counter += 1

value = sys.argv[arg\_counter] cmlargs[option] = value

else:

print ’The option %s is not registered’ % option

arg\_counter += 1

The advantage with this technique is that each time you need to add a new pa- rameter and a corresponding command-line option to the script, you can sim- ply add a new item to the dictionary cmlargs. Exercise 8.1 on page 324 demon- strates an interesting combination of cmlargs and the getopt or optparse module. The downside with the code segment above is that all the variables cmlargs[option] are of string type, i.e., we must explicit convert them to floating-point numbers in order to perform arithmetic computations with them. A more flexible, but also more advanced solution using the same ideas, is presented in Chapter 11.4.

Dictionaries behave like lists when it comes to copying and assignment, see Chapter 3.2.10 for the various options that are available.

Iterating over the keys in a dictionary is done with the standard Python construction for element in data\_structure, e.g.,

for key in cmlargs: # visit items, key by key print "cmlargs[’%s’]=%s" % (key, cmlargs[key])

There is no predefined sequence of the keys in a dictionary. Sometimes you need to have control of the order in which the keys are processed. You can then work with the keys in sorted order:

for option in sorted(cmlargs): # visit keys in sorted order print "cmlargs[’%s’]=%s" % (option, cmlargs[option])

This construction was new in Python 2.4. In older Python versions one had to get the keys and sort this list in-place:

keys = cmlargs.keys() keys.sort()

for option in keys:

print "cmlargs[’%s’]=%s" % (option, cmlargs[option])

Environment Variables. All environment variables a user has defined are available in Python scripts throught the dictionary-like variable os.environ. The syntax for accessing an environment variable X is os.environ[’X’]. One can read and modify environment variables within the script. Child processes (as started by the subprocess module, or commands.getstatusoutput, or sim- ilar) inherit modified environment variables.

The get method in dictionary-like objects is particularly convenient for testing the content of a specific environment variable, e.g.,

root = os.environ.get(’HOME’, ’/tmp’)

Here we set root as the home directory if HOME is defined as an environment

variable, otherwise we use /tmp. The alternative if test is more verbose:

if ’HOME’ in os.environ:

root = os.environ[’HOME’]

else:

root = ’/tmp’

Here is an example, where we add the directory $scripting/src/py/intro to the PATH environment variable. This enables us to run scripts from the in- troductory part of this book regardless of what the current working directory is.

if ’PATH’ in os.environ and ’scripting’ in os.environ: os.environ[’PATH’] += os.pathsep + os.path.join(

os.environ[’scripting’], ’src’, ’py’, ’intro’)

The os.pathsep variable holds the separator in the PATH string, typically colon on Unix and semi-colon on Windows. Recall that the os.path.join function concatenates the individual directory names (and optionally a filename) to a full path with the correct platform-specific separator. Our use of os.path.join and os.pathsep makes the code valid on all operating systems supported by Python. Running

failure, output = commands.getstatusoutput(’echo $PATH’) print output

shows that the child process has inherited a PATH variable with our recently added directory $scripting/src/py/intro at the end.

The example of modifying the PATH environment variable is particularly useful when you want to run certain programs as an operating system com- mand but do not know if the user of the script has the correct PATH variable

to “see” the programs. The technique is important in CGI scripts (see Chap- ter 7.2). An alternative to extending the PATH variable is to construct the complete path of the program, e.g.,

simviz1 = os.path.join(os.environ[’scripting’], ’src’, ’py’, ’intro’, ’simviz1.py’)

However, this solution may easily fail on Windows machines if directories contain blanks. Say your scripting variable is set to some name of a direc- tory under C:\My Documents. A command running something like ’simviz1 ’ + ... will then actually try to run a program C:\My since the first space is interpreted as a delimiter between the program and its command-line ar- guments. Adding directories with spaces to the PATH variable works well, so extending the PATH variable is the recommended cross-platform way of exe- cuting programs in other directories.

The Unix-specific which command can easily be given a cross-platform implementation in Python. The basic ingredients of a relevant code segment consist of splitting the PATH variable into a list of its directories and checking if the program is found in one of these directories. This is a typical example of a task that is very convenient to perform in Python:

import os

program = ’vtk’ # a sample program to search for pathdirs = os.environ[’PATH’].split(os.pathsep) for d in pathdirs:

if os.path.isdir(d): # skip non-existing directories if os.path.isfile(os.path.join(d, program)):

program\_path = d; break

try: # program was found if program\_path is defined print ’%s found in %s’ % (program, program\_path)

except:

print ’%s not found’ % program

Exercises 3.6–3.10 develop some useful tools related to this code segment. A professional which.py script is linked from the doc.html page.

3.2.6 Splitting and Joining Text

Splitting a string into words is done with the built-in split function in strings:

>>> files = ’case1.ps case2.ps case3.ps’ >>> files.split()

[’case1.ps’, ’case2.ps’, ’case3.ps’]

One can also specify a split with respect to a delimiter string, e.g.,

>>> files = ’case1.ps, case2.ps, case3.ps’

>>> files.split(’, ’)

[’case1.ps’, ’case2.ps’, ’case3.ps’]

>>> files.split(’, ’) # extra erroneous space after comma... [’case1.ps, case2.ps, case3.ps’] # no split

Strings can also be split with respect to a general regular expression (as explained in Chapter 8.2.7):

>>> files = ’case1.ps, case2.ps, case3.ps’ >>> import re

>>> re.split(r’,\s\*’, files)

[’case1.ps’, ’case2.ps’, ’case3.ps’]

As another example, consider reading a series of real numbers from a file of the form

1.432 5E-09

1.0

3.2 5 69 -111 478

That is, the file contains real numbers, but the number of reals on each line differs, and some lines are empty. If we load the file content into a string, extracting the numbers is trivial using a split with respect to whitespace and converting each resulting word to a floating-point number:

f = open(somefile, ’r’)

numbers = [float(x) for x in f.read().split()]

Such an example demonstrates the potential increase in human efficiency when programming in a language like Python with strong support for high- level text processing (consider doing this in C!).

The inverse operation of splitting, i.e., combining a list (or tuple) of strings into a single string, is accomplished by the join function in string objects. For example,

>>> filenames = [’case1.ps’, ’case2.ps’, ’case3.ps’] >>> cmd = ’print ’ + ’ ’.join(filenames)

>>> cmd

’print case1.ps case2.ps case3.ps’

3.2.7 String Operations

Strings can be written in many ways in Python. Different types of quotes can be used interchangeably: ’, ", """, and ’’’, even when using printf-style formatting or variable interpolation.

s1 = ’with single quotes’

s2 = "with double quotes"

s3 = ’with single quotes and a variable: %g’ % r1 s4 = """as a triple double quoted string"""

s5 = """triple double (or single) quoted strings allow multi-line text (i.e., newline is preserved)

and there is no need for backslashes before embedded

quotes like " or ’

"""

s6 = r’raw strings start with r and \ is always a backslash’ s7 = r’’’Windows paths such as C:\projects\sim\src

qualify for raw strings’’’

The raw strings, starting with r, are particularly suited in cases where back- slashes appear frequently, e.g., in regular expressions, in LATEX source code, or in Windows/DOS paths. In the statement s8="\\t" the first backslash is used to quote the next, i.e., preserve the meaning of the second backslash as the character \. The result is that s8 contains \t. With raw strings, s8=r"\\t" sets s8 to \\t. Hence, if you just want the text \t, the code becomes more readable by using a raw string: s8=r"\t".

Strings are concatenated using the + operator: myfile = filename + ’\_tmp’ + ".dat"

As an example, the myfile variable becomes ’case1\_tmp.dat’ if filename is ’case1’.

Substrings of filename are extracted by slicing:

>>> teststr = ’0123456789’ >>> teststr[0:5]; teststr[:5] ’01234’

’01234’

>>> teststr[3:8]

’34567’

>>> teststr[3:]

’3456789’

The need for checking if a string starts or ends with a specific text arises frequently:

if filename.startswith(’tmp’): ...

if filename.endswith(’.py’): ...

Other widely used string operations are

s1.upper() # change s1 to upper case s1.lower() # change s2 to lower case

We refer to the Python Library Reference for a complete documentation of built-in methods in strings (follow the “string object” link in the index and proceed with the section on “String Methods”).

The String Module. In older Python code you may see use of the string module instead of built-in methods in string objects. For example,

import string

lines = string.split(filestr, ’\n’) filestr = string.join(lines, ’\n’)

is equivalent to

lines = filestr.splitlines() # or filestr.split(’\n’) filestr = ’\n’.join(lines)

Most built-in string methods are found in the string module under the same names (see the Python Library Reference for a complete documentation of the string module).

3.2.8 Text Processing

Text Searching. There are several alternatives for testing whether a string contains a specified text:

– Exact string match:  
 if line == ’double’:  
   
 # line equals ’double’

* if ’double’ in line:  
   # line contains ’double’  
     
  # equivalent, but less intuitive test: if line.find(’double’) != -1:  
     
  # line contains ’double’
* – Matching with Unix shell-style wildcard notation:  
     
  import fnmatch  
   if fnmatch.fnmatch(line, ’double’):  
     
  # line contains ’double’  
   Here, double can be any valid wildcard expression, such as [Dd]ouble and  
     
  double\*.
* – Matching with full regular expressions (Chapter 8.2):  
     
  import re  
   if re.search(r’double’, line):  
     
  # line contains ’double’  
     
  In this example, double can actually be replaced by any valid regular expression. Note that the raw string representation (see Chapter 3.2.7) of ’double’ has no effect in this particular example, but it is a good habit to use raw strings in regular expression specifications.  
     
  Text Substitution. Substitution of a string s by another string t in some string r is done with the replace method in string objects:  
     
  r = r.replace(s, t)  
     
  Substitution of a regular expression pattern by some text replacement in a string r goes as follows:

r = re.sub(pattern, replacement, r)

# or:

cre = re.compile(pattern)

r = cre.sub(replacement, r)

Here is a complete example where double is substituted by float everywhere in a file:

f = open(filename, ’r’)

filestr = f.read().replace(’float’, ’double’) f.close()

f = open(filename, ’w’)

f.write(filestr)

f.close()

For safety, we should take a copy of the file before the overwrite.

Regular Expression Functionality. Text processing frequently makes heavy use of regular expressions, a topic covered in Chapter 8.2. A list of common Python functionality in the re module when working with regular expressions is presented here as a quick reference.

* – Compile a regular expression:  
   c = re.compile(pattern, flags)

– Match a pattern:  
 m = re.search(pattern, string, flags)  
   
 m = c.search(string)

* – Substitute a pattern:  
   string = re.sub(pattern, replacement, string)  
     
  string = c.sub(replacement, string)  
     
  # backreferences (in substitutions):  
   # \1, \2, etc., or  
   # \g<1>, \g<2>, etc., or  
   # named groups: \g<name1>, \g<name2>, etc.

– Find multiple matches in a string: list = re.findall(pattern, string)  
   
 list = c.findall(string)

* – Split strings:  
   list = re.split(pattern, string)  
     
  list = c.split(string)  
   The re.search function returns a MatchObject instance, here stored in m, with  
     
  several useful methods:  
     
  – m.groups() returns a list of all groups in the match, m.group(3) returns the 3rd matched group, and m.group(0) returns the entire match.

– string[m.start(2):m.end(2)] returns the part of string that is matched by the 2nd group.

We mention that the re module has a function match for matching a pattern at the beginning of the string, but in most cases the search function, which searches for a match everywhere in the string, is what you want.

3.2.9 The Basics of a Python Class

Readers familiar with class programming7 from, e.g., C++ or Java may get started with Python classes through a simple example:

class MyBase:

def \_\_init\_\_(self, i, j): # constructor

self.i = i; self.j = j

def write(self): # member function

print ’MyBase: i=’, self.i, ’j=’, self.j

This class has two data members, i and j, recognized by the prefix self. These members are called data attributes or just attributes in Python terminology. Attributes can be “declared” anywhere in the class: just assign values to them and they come into existence, as usual in dynamically typed languages.

The \_\_init\_\_ function is a constructor, used to initialize the instance at creation time. For example,

inst1 = MyBase(6,9)

leads to a call to the constructor, resulting in i and j as the integers 6 and 9, respectively. An instance of class MyBase is created, and the variable inst1 is a reference to this instance. We can access the attributes as inst1.i and inst1.j.

A function in a class is referred to as a method in Python terminology, and every method must have self as the first argument. However, this argument is not explicitly used when calling the method. The self variable is Python’s counterpart to the this pointer in C++, with the exception that Python requires its use when accessing attributes or methods.

The write method is an example of an ordinary method, taking only the required argument self. When called, self is omitted:

inst1.write()

Inside the write method, the self argument becomes a reference to the inst1

instance.

Subclasses. A subclass MySub of MyBase can be created as follows:

7 If you are new to class programming, it might be better to jump to Chapter 8.6.1.

class MySub(MyBase):

def \_\_init\_\_(self, i, j, k): # constructor

MyBase.\_\_init\_\_(self, i, j) # call base class constructor self.k = k

def write(self):

print ’MySub: i=’, self.i, ’j=’, self.j, ’k=’, self.k

The syntax should be self-explanatory: the subclass adds an attribute k and defines its own version of the constructor and the write method. Since a subclass inherits data attributes and methods from the base class, class MySub contains three data attributes: i, j, and k.

Here is an interactive session demonstrating what we can do with our two trivial classes:

>>> def write(v):

v.write()

>>> i1 = MyBase(’some’, ’text’)

>>> write(i1)

MyBase: i= some j= text

>>> i2 = MySub(’text’, 1.1E+09, [1,9,7]) >>> write(i2)

MySub: i= text j= 1100000000.0 k= [1, 9, 7]

Classes with Function Behavior. A class implementing a method \_\_call\_\_

may act as an ordinary function. Let us look at an example:

class F:

def \_\_init\_\_(self, a=1, b=1, c=1):

self.a = a; self.b = b; self.c = c

def \_\_call\_\_(self, x, y):

return self.a + self.b\*x + self.c\*y\*y

f = F(a=2, b=4)

v = f(2, 1) + f(1.2, 0)

We make an instance f of class F and call f as if it were an ordinary func- tion! The call f(1.2, 0) actually translates to f.\_\_call\_\_(1.2, 0) (see Chap- ter 8.6.6). This feature is particularly useful for representing functions with parameters, where we need to distinguish between the parameters and the independent variables. Say we have a function

f (x, y; a, b, c) = a + bx + cy2 .

Here x and y are independent variables, while a, b, and c are parameters (we have in the notation f(x,y;a,b,c) explicitly indicated this). If we want to pass such a function to, e.g., an integration routine, that routine will assume that the function takes two independent variables as arguments, but how can the function then get the values of a, b, and c? The classical solution from Fortran and C is to use global variables for the parameters and let the function arguments coincide with the independent variables:

global a, b, c

def f(x, y):

return a + b\*x + c\*y\*y

A class like F above, where the parameters are attributes, is a better solution since we avoid global variables. The parameters become a part of the instance (“function object”) but not of the call syntax. In our example above, f(1.2,0) evaluates f(1.2,0;2,4,1) = 2+4·1.2+1·0·0. The parameters were set when we constructed f, but we can alter these later by assigning values to the attributes in F (e.g., f.c=6).

Instances of classes with \_\_call\_\_ methods are in this book referred to as callable instances8 and used in many places.

Chapter 8.6 contains much more information about classes in Python. Extended material on callable instances appears in Chapter 12.2.2.

3.2.10 Copy and Assignment

Newcomers to Python can be confused about copying references and copying objects in assignments. That is, in a statement like b = a, will b be a sort of reference to a such that the contents of b are changed if those of a are changed? Or will b be a true copy of a and hence immune to changes in a?

Variables in Python are references to Python objects. The assignment b = a therefore makes b refer to the same object as a does. Changing a might or might not affect b – this depends on whether we perform in-place modifi- cations in a or let a refer to a new object. Some examples will hopefully make this clear. Consider

a=3 b=a a=4

Here, a first refers to an int object with the value 3, b refers to the same object as a, and then a refers to a new int object with the value 4, while b remains referring to the int object with value 3. If the a=4 statement should affect b, we must perform in-place modification of the int object that a refers to, but this is not possible (number objects are immutable).

Deleting a variable may not imply destruction of the object referred to by the variable unless there are no other references to the variable:

a=3

b=a

# remove a, but not the int object since b still refers to it: del a

print b # prints 3

# remove b and the int object (no more references to the object): del b

8 In C++ this is known as function objects or functors [1].

Python has an id function that returns an integer identification of an object. We can either use id or the special is operator to test whether two variables refer to the same object:

>>> a = 3

>>> b = a

>>> id(a), id(b) (135531064, 135531064) >>> id(a) == id(b) True

>>> a is b

True

>>> a = 4

>>> id(a), id(b) (135532056, 135531064) >>> a is b

False

Let us make a corresponding example with a list:

>>> a = [2, 6]

>>> b = a

>>> a is b

True

>>> a = [1, 6, 3]

>>> a is b

False

Now a and b refer to two different lists. Instead of assigning the latter (new) list to a, we could perform in-place modifications of the original list referred to by a:

>>> a = [2, 6]

>>> b = a

>>> a[0] = 1

>>> a.append(3)

>>> a

[1, 6, 3]

>>> b

[1, 6, 3]

>>> a is b

True

Dictionaries are mutable objects, like lists, and allows in-place changes in the same way:

>>> a = dict(q=6, error=None) >>> b = a

>>> a[’r’] = 2.5

>>> a

>>> b {’q’:

{’q’: 6, ’r’: 2.5, ’error’: None} >>> a is b

True

>>> a

= ’a string’ # make a refer to a new (string) object # new contents in a do not affect b

6, ’r’: 2.5, ’error’: None}

What if we want to have b as a copy of a? For list we can use a[:] to extract a copy9 of the elements in a:

>>> a = [2, 6, 1]

>>> b = a[:]

>>> b is a

False

>>> a[0] = ’some string’

>>> b[0] # not affected by assignment to a[0] 2

For dictionaries, we use the copy method:

>>> a = {’refine’: False} >>> b = a.copy()

>>> b is a

False

With instances of user-defined classes the situation gets a bit more com- plicated. The shallow and deep copy concepts are closely related to the assignment issue. Shallow copy means copying references and deep copy im- plies copying the complete contents of an object (roughly speaking). Python’s copy module lets us control whether an assignment should be a shallow or deep copy. We refer to the documentation of the copy module in the Python Library Reference for capabilities of the module and more precise handling and definition of copy issues. Here, we shall as usual limit the presentation to an illustrative example, showing what assignment and deep vs. shallow copy means for user-defined objects, lists, and dictionaries.

Turning the attention to user-defined data types, we can create a very simple class A with a single data item (self.x):

class A:

def \_\_init\_\_(self, value):

self.x = value

def \_\_repr\_\_(self):

return ’x=%s’ % self.x

The \_\_repr\_\_ method allows printing any instance of class A, also when the instance is part of a nested list. This feature is exploited in the tests below.

Assignment, shallow copy, and deep copy of an instance of A are performed by

>>> a = A(-99)

>>> b\_assign = a

>>> b\_shallow = copy.copy(a)

>>> b\_deep = copy.deepcopy(a)

# make instance a # assignment

# shallow copy

# deep copy

We then change the a.x attribute from -99 to 9. Let us see how this affects the contents of the other variables:

9 Note that for Numerical Python arrays, a[:] will not make a copy of the elements, but a reference to all elements in a, see page 137.

>>> a.x = 9

>>> print ’a.x=%s, b\_assign.x=%s, b\_shallow.x=%s, b\_deep.x=%s’ %\

(a.x, b\_assign.x, b\_shallow.x, b\_deep.x) a.x=9, b\_assign.x=9, b\_shallow.x=-99, b\_deep.x=-99

The assignment of user-defined data types, as in b\_assign = a, stores a ref- erence to a in b\_assign. Changing an attribute in a will then be reflected in b\_assign. The shallow copy copy.copy(a) creates an object of type A and inserts references to the objects in a, i.e., b\_shallow.x is a reference to the integer a.x. The deep copy statement copy.deepcopy(a) results in b\_deep.x being a true copy of the value in a.x, not just a reference to it. When chang- ing the integer a.x to 9, the shallow copy holds a reference to the previous integer object pointed to by a.x, not the new integer object with value 9, and that is why the change in a is not reflected in b\_shallow. However, if we let a.x be a list, a = A([-2,3]), and perform an in-place change of the list,

>>> a = A([-2,3])

>>> b\_assign = a

>>> b\_shallow = copy.copy(a)

>>> b\_deep = copy.deepcopy(a)

>>> a.x[0] = 8 # in-place modification

the reference in the shallow copy points to the same list and will reflect the change:

>>> print ’a.x=%s, b\_assign.x=%s, b\_shallow.x=%s, b\_deep.x=%s’ %\ (a.x, b\_assign.x, b\_shallow.x, b\_deep.x)

a.x=[8, 3], b\_assign.x=[8, 3], b\_shallow.x=[8, 3], b\_deep.x=[-2, 3]

These examples should demonstrate the fine differences between assignment, shallow copy, and deep copy.

Let us look at a case with a heterogeneous list, where we change two list items, one of them being an A instance:

>>> a = [4,3,5,[’some string’,2], A(-9)]

>>> b\_assign

>>> b\_shallow

>>> b\_deep

>>> b\_slice

>>> a[3] = 999; a[4].x = -6

>>> print ’b\_assign=%s\nb\_shallow=%s\nb\_deep=%s\nb\_slice=%s’ % \

(b\_assign, b\_shallow, b\_deep, b\_slice) b\_assign=[4, 3, 5, 999, x=-6]

b\_shallow=[4, 3, 5, [’some string’, 2], x=-6] b\_deep=[4, 3, 5, [’some string’, 2], x=-9] b\_slice=[4, 3, 5, [’some string’, 2], x=-6]

The deep copy makes a complete copy of the object, and there is thus no track of the changes in a. The variable b\_assign is a reference, which reflects all changes in a. Each item in the b\_shallow list is a reference to the corre- sponding item in a. Hence, when the list in a[3] is replaced by an integer 999, b\_shallow[3] still holds a reference to the old list. On the other hand,

the reference b\_shallow[4] to an A instance remains unaltered, only the x attribute of that instance changes, and that is why the new value is “visible” from b\_shallow. Dictionaries behave in a completely similar way. A script src/ex/copytypes.py contains the shown constructions and is available for further investigation.

3.2.11 Determining a Variable’s Type

The are basically three ways of testing a variable’s type. Let us define

>>> files = [’myfile1.dat’, ’myfile2’]

and then show how to test if files is a list. The isinstance function checks

if an object is of a certain type (list, str, dict, float, int, etc.): >>> isinstance(files, list)

True

The second argument to isinstance can also be a tuple of types. For example, testing if files is either a list, tuple, or an instance of class MySeq, we could issue

>>> isinstance(files, (list, tuple, MySeq)) True

The type(x) function returns the class object associated with x. Here are two typical tests:

>>> type(files) == type([]) True

>>> type(files) == list True

The module types contains type objects used in older Python codes: ListType, StringType, DictType, FloatType, IntType, etc.

>>> import types

>>> type(files) == types.ListType True

We stick to the isinstance function in this book.

The next example concerns determining the type of the entries in a het-

erogeneous list:

somelist = [’text’, 1.28736, [’sub’, ’list’],

{’sub’ : ’dictionary’, ’heterogeneous’ : True},

(’some’, ’sub’, ’tuple’), 888, MyClass(’some input’)]

class\_types = ((int, long), list, tuple, dict, str, basestring, float, MyClass)

3. Basic Python

def typecheck(i):

for c in class\_types:

if isinstance(i, c): print c,

for i in somelist:

print i, ’is’,

func(i)

print

The output of the tests becomes

text is <type ’str’> <type ’basestring’>

1.28736 is <type ’float’>

[’sub’, ’list’] is <type ’list’>

{’heterogeneous’: 1, ’sub’: ’dictionary’} is <type ’dict’> (’some’, ’sub’, ’tuple’) is <type ’tuple’>

888 is (<type ’int’>, <type ’long’>)

<\_\_main\_\_.MyClass instance at 0x4021e50c> is \_\_main\_\_.MyClass

Note that the string ’text’ is both a str and basestring. It is recommended to test for strings with isinstance(s, basestring) rather than isinstance(s, str), because the former is true whether the string is a plain string (str) or a Unicode string (unicode).

The current code example is available in src/py/examples/type.py. This file also contains alternative versions of the typecheck function using type.

Occasionally it is better to test if a variable belongs to a category of types rather than to test if it is of a particular type. Python distinguishes between

– sequence types (list, tuple, Numerical Python array), – number types (float, int, complex),

– mapping types (dictionary), and

– callable types (function, class with \_\_call\_\_ operator).

For variables in each of these classes there are certain legal operations. For instance, sequences can be iterated, indexed, and sliced, and callables can be called like functions. The operator module has some functions for checking if a variable belongs to one of the mentioned type classes:

operator.isSequenceType(a) operator.isNumberType(a) operator.isMappingType(a) operator.isCallable(a) callable(a)

3.2.12 Exercises

# True if a is a sequence #Trueifaisanumber #Trueifaisamapping

# True if a is a callable

# recommended for callables

Exercise 3.1. Write format specifications in printf-style.

Consider the following initialization of a string, two integers, and a floating- point variable:

name = ’myfile.tmp’; i = 47; s1 = 1.2; s2 = -1.987;

Write the string in a field of width 15 characters, and adjusted to the left; write the i variable in a field of width 5 characters, and adjusted to the right; write s1 as compactly as possible in scientific notation; and write s2 in decimal notation in a field of minimum width. ⋄

Exercise 3.2. Write your own function for joining strings.

Write a function myjoin that concatenates a list of strings to a single string, with a specified delimiter between the list elements. That is, myjoin is supposed to be an implementation of string object’s join function (or string.join) in terms of basic string operations. ⋄

Exercise 3.3. Write an improved function for joining strings.

Perl’s join function can join an arbitrary composition of strings and lists of strings. The purpose of this exercise is to write a similar function in Python. Recall that the built-in join method in string objects, or the string.join function, can only join strings in a list object. The function must handle an arbitrary number of arguments, where each argument can be a string, a list of strings, or a tuple of strings. The first argument should represent the delimiter. As an illustration, the function, here called join, should be able to handle the following examples:

>>> list1 = [’s1’, ’s2’, ’s3’]

>>> tuple1 = (’s4’, ’s5’)

>>> ex1 = join(’ ’, ’t1’, ’t2’, list1, tuple1, ’t3’, ’t4’) >>> ex1

’t1 t2 s1 s2 s3 s4 s5 t3 t4’

>>> ex2 = join(’ # ’, list1, ’t0’)

>>> ex2

’s1 # s2 # s3 # t0’

Hint: Variable number of arguments in functions is treated in Chapter 3.3.3, whereas Chapter 3.2.11 explains how to check the type of the arguments. ⋄

Exercise 3.4. Never modify a list you are iterating on.

Try this code segment:

print ’plain remove in a for loop:’ list = [3,4,2,1]

for item in list:

print ’visiting item %s in list %s’ % (item, list) if item > 2:

list.remove(item)

After the loop, the list is [4,2,1] even though the item 4 is bigger than 2 and should have been removed. The problem is that the for loop visits index 1 in the second iteration of the loop, but the list is then [4,2,1] (since the first item is removed), and index 1 is then the element 1, i.e., we fail to visit the item 4.

The remedy is to never modify a list that you are iterating over. Instead you should take a copy of the list. An element by element copy is provided by list[:] so we can write

for item in list[:]:

if item > 2:

list.remove(item)

This results in the expected list [2,1].

Write a code segment that removes all elements larger than 2 in the list

[3,4,2,1], but use a while loop and an index that is correctly updated in each pass in the loop.

The same problem appears also with other list modification functions, such as del, e.g.,

list = [3,4,2,1]

for item in list:

del list[0]

Explain why the list is not empty (print list and item inside the loop if you are uncertain). Construct a new loop where del list[0] successfully deletes all list items, one by one. ⋄

Exercise 3.5. Make a specialized sort function.

Suppose we have a script that performs numerous efficiency tests. The output from the script contains lots of information, but our purpose now is to extract information about the CPU time of each test and sort these CPU times. The output from the tests takes the following form:

...

f95 -c -O0 versions/main\_wIO.f F77WAVE.f f95 -o app -static main\_wIO.o F77WAVE.o app < input > tmp.out

CPU-time: 255.97 f95 -O0 formatted I/O f95 -c -O1 versions/main\_wIO.f F77WAVE.f f95 -o app -static main\_wIO.o F77WAVE.o app < input > tmp.out

CPU-time: 252.47 f95 -O1 formatted I/O f95 -c -O2 versions/main\_wIO.f F77WAVE.f f95 -o app -static main\_wIO.o F77WAVE.o app < input > tmp.out

CPU-time: 252.40 f95 -O2 formatted I/O ...

-lf2c

-lf2c

-lf2c

First we need to extract the lines starting with CPU-time. Then we need to sort the extracted lines with respect to the CPU time, which is the number appearing in the second column. Write a script to accomplish this task. A suitable testfile with output from an efficiency test can be found in src/misc/efficiency.test.

Hint: Find all lines with CPU time results by using a string comparison of the first 7 characters to detect the keyword CPU-time. Then write a tailored

sort function for sorting two lines (extract the CPU time from the second column in both lines and compare the CPU times as floating-point numbers). ⋄

Exercise 3.6. Check if your system has a specific program.

Write a function taking a program name as argument and returning true if the program is found in one of the directories in the PATH environment variable and false otherwise. This function is useful for determining whether a specific program is available or not. Hint: Read Chapter 3.2.5. ⋄

Exercise 3.7. Find the paths to a collection of programs.

A script often makes use of other programs, and if these programs are not available on the computer system, the script will not work. This exercise shows how you can write a general function that tests whether the required tools are available or not. You can then terminate the script and notify to the user about the software packages that need to be installed.

The idea is to write a function findprograms taking a list of program names as input and returning a dictionary with the program names as keys and the programs’ complete paths on the current computer system as values. Search the directories in the PATH environment variable as indicated in Exericise 3.6. Allow a list of additional directories to search in as an optional argument to the function. Programs that are not found should have the value None in the returned dictionary.

Here is an illustrative example of using findprograms to test for the exis- tence of some utilities used in this book:

programs = {

’gnuplot’

’gs’

’f2py’

’swig’

’convert’

}

: ’plotting program’,

: ’ghostscript, ps/pdf converter and previewer’,

: ’generator for Python interfaces to Fortran’,

: ’generator for Python interfaces to C/C++’,

: ’image conversion,part of the ImageMagick package’,

installed = findprograms(programs.keys()) for program in installed:

if installed[program]:

print ’You have %s (%s)’ % (program, programs[program])

else:

print ’\*\*\* Program’, program, ’was not found’ print ’ .....(%s)’ % programs[program]

On Windows you need to test for the existence of the program names with .exe or .bat extensions added (Chapter 8.1.2 explains how you can make separate code for Unix and Windows in this case). ⋄

Exercise 3.8. Use Exercise 3.7 to improve the simviz1.py script.

Use the findprograms function from Exercise 3.7 to check that the script simviz1.py from Chapter 2.3 has access to the two programs oscillator and gnuplot. ⋄

Exercise 3.9. Use Exercise 3.7 to improve the loop4simviz2.py script.

The loop4simviz2.py script from Chapter 2.4.4 needs access to a range of different tools (oscillator, gnuplot, convert, etc.). Use the findprograms function from Exercise 3.7 to check that all the required tools are available to the user of the script. In case a tool is missing, drop the corresponding action (if not essential) and dump a warning message. ⋄

Exercise 3.10. Find the version number of a utility.

The findprograms function developed in Exercise 3.7 is fine for checking that certain utilities are available on the current computer system. However, in many occasions it is not sufficient that a particular program exists, a special version of the program might be needed. The purpose of the present exercise is to produce code segments for checking the version of a program.

Suppose you need to know the version number of the Ghostscript (gs) utility. Ghostview offers, like many other programs, a command-line option for printing the version number. You can type gs -v and get a typical output

GNU Ghostscript 6.53 (2002-02-13)

Copyright (C) 2002 artofcode LLC, Benicia, CA. All rights reserved.

This Python code segment extracts 6.53 as the version number from the output of gs -v:

installed = findprograms(programs.keys()) if installed[’gs’]:

failure, output = commands.getstatusoutput(’gs -v’) version = float(output.read().split()[2]) output.close()

Write functions that return the version of gs, perl, convert, and swig. The former three programs write their version information to standard output, while swig writes to standard error, but both standard output and standard error are captured by the system command above.

By the way, the version of Python is obtained from the built-in string sys.version or the sys.version\_info tuple:

>>> print sys.version

2.5 (r25:409, Feb 27 2007, 19:35:40)

[GCC 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)] >>> sys.version[:3]

’2.5’

>>> sys.version\_info

(2, 5, 0, ’final’, 0)

3.3 Functions

A typical Python function can be sketched as

⋄

def function\_name(arg1, arg2, arg3): # statements

return something

Any data structure can be returned, and None is returned in the absence of a

return statement. A simple example of a Python function may read

def debug(comment, var):

if os.environ.get(’PYDEBUG’, ’0’) == ’1’:

print comment, var

The function prints the contents of an arbitrary variable var, with a leading text comment, if the environment variable PYDEBUG is defined and has a value ’1’. (Environment variables are strings, so true and false are taken as the strings ’1’ and ’0’.) One can use the function to dump the contents of data structures for debugging purposes:

v1 = file.readlines()[3:]

debug(’file %s (exclusive header):’ % file.name, v1) # dump list

v2 = somefunc()

debug(’result of calling somefunc:’, v2)

The debugging is turned on and off by setting PYDEBUG in the executing envi- ronment10 :

export PYDEBUG=1

export PYDEBUG=0

Note the power of a dynamically typed language as Python: debug can be used to dump the contents of any printable data structure!

Function Variables are Local. All variables declared in a function are local to that function, and destroyed upon return, unless one explicitly specifies a variable to be global:

def somefunc():

global cc # allow assignment to global variable cc

Global variables that are only accessed, not assigned, can be used without a global statement. We refer to Chapter 8.7 for more detailed information on the scope of variables in Python.

3.3.1 Keyword Arguments

Python allows the use of keyword arguments, also called named arguments. This makes the code easy to read and use. Each argument is specified by a keyword and a default value. Here is an example of a flexible function for making directories (cf. the method we explain on page 53):

10 Python has a built-in variable \_\_debug\_\_ that we could use instead of our own PYDEBUG environment variable. \_\_debug\_\_ is set to false if the Python interpreter is run with the -O (optimize) option, i.e., we run python -O scriptname.

3. Basic Python

def mkdir(dirname, mode=0777, remove=True, chdir=True): if os.path.isdir(dirname):

if remove:

shutil.rmtree(dirname)

else:

return False # did not make a new directory

os.mkdir(dirname, mode)

if chdir: os.chdir(dirname)

return True # made a new directory

In this function, dirname is a positional (also called required) argument, whereas mode, remove, and chdir are keyword arguments with the specified default values. If we call

mkdir(’tmp1’)

the default values for mode, remove, and chdir are used, meaning that tmp1 is removed if it exists, then created, and thereafter we change the current working directory to tmp1. Some or all of the keyword arguments can be supplied in the call, e.g.,

mkdir(’tmp1’, remove=False, mode=0755)

The sequence of the keyword arguments can be arbitrary as long as the keyword is included in the call. In this latter example, chdir becomes True (the default value). Note that keyword arguments must appear after the positional arguments.

Sensible use of names in keyword arguments helps to document the code. I think both function definitions and calls to functions are easier to read with keyword arguments. Novice users can rely on default values, whereas more experienced users can fine-tune the call (cf. the discussion on page 11). We shall see that the Tkinter GUI module presented in Chapter 6 relies heavily on keyword arguments.

3.3.2 Doc Strings

It is a Python programming standard to include a triple-quoted string, right after the function heading, for documenting the function:

def mkdir(dirname, mode=0777, remove=True, chdir=True): """

Create a directory dirname (os.mkdir(dirname,mode)). If dirname exists, it is removed by shutil.rmtree if remove is true. If chdir is true, the current working directory is set to dirname (os.chdir(dirname)).

""" ...

Such a string is called a doc string and will be used frequently hereafter in this book. Appendix B.2 explains more about doc strings and how different tools

can automatically extract doc strings and generate documentation. The doc string often contains an interactive session from a Python shell demonstrating usage of the function. This session can be used for automatic testing of a function, see Appendix B.4.5.

3.3.3 Variable Number of Arguments

Variable-length argument lists are allowed in Python functions. An asterix as prefix to the argument name signifies a variable-length argument list. Here is a sketch of a sample code:

def somefunc(a, b, \*args):

# args is a tuple of all supplied positional arguments ...

for arg in args:

<work with arg>

A double asterix as prefix denotes a variable-length set of of keyword argu- ments:

def somefunc(a, b, \*args, \*\*kwargs):

# args is a tuple of all supplied positional arguments

# kwargs is a dictionary of all supplied keyword arguments ...

for arg in args:

<work with arg>

for key in kwargs:

<work with argument key and its value kwargs[key]>

A function statistics with a variable number of arguments appears below. The function returns a tuple containing the average and the minimum and maximum value of all the arguments:

def statistics(\*args):

"""

Compute the average, minimum and maximum of all arguments. Input: a variable no of arguments (must be numbers). Output: tuple (average, min, max).

"""

avg=0; n=0; #avgandnarelocalvariables for number in args: # sum up all numbers (arguments)

n += 1; avg += number avg /= float(n)

min = args[0]; max = args[0] for term in args:

if term < min: min = term if term > max: max = term

return avg, min, max

# example on using the statistics function:

average, vmin, vmax = statistics(v1, v2, v3, b)

print ’average =’, average, ’min =’, vmin, ’max=’, vmax

Observe that three numbers are computed in the function and returned as a single data structure (a tuple). This is the way to return multiple values from a Python function. (C/C++ programmers may get worried about returning local variables, but in Python only references are transferred, and the garbage collecting system does not delete objects as long as there are references to them.)

We remark that the statistics function was made for illustrating basic Python programming. An experienced Python programmer would probably write

def statistics(\*args):

return reduce(operator.add, args)/float(len(args)), \

min(args), max(args)

The reader is encouraged to look up the documentation of the four functions reduce, operator.add, min, and max to understand this compact version of the statistics function. With Python’s sum function the statistics function can be even shorter and more understandable:

def statistics(\*args):

return sum(args)/float(len(args)), min(args), max(args)

3.3.4 Call by Reference

Fortran, C, and C++ programmers are used to pass variables to a function and get the variables modified inside the function. This is commonly referred to as call by reference, achieved by using pointers or references11. Some also speak about in situ or in-place modification of arguments. In Python the same effect is not straightforward to obtain, because Python’s way of trans- ferring arguments applies an assignment operator between the argument and the value in the call (“call by assignment” could be an appropriate way of describing Python’s call mechanism). That is, given a function def f(x,y) and a call f(2,a), the x and y arguments get their values by assignments x=2 and y=a. If we want to change the a argument inside the f function and notice the change in the calling code, a must therefore be a mutable object (list, dictionary, class instance, Numerical Python array) that allows in-place mod- ifications. An immutable a object, like numbers, strings, and tuples, cannot be changed in-place, and a new assignment to y, as in y=3, has no effect on a. Note also that the x and y arguments are local variables which are destroyed when returning from the function.

Let us illustrate how elements of a list or a dictionary can be changed inside a function:

11 WeremarkthatbydefaultandcontrarytoFortran,CandC++passesarguments by value (i.e., the functions work on copies of the arguments). The point is that the mentioned languages have constructs for call by reference.

>>> def somefunc(mutable, item, item\_value): mutable[item] = item\_value

>>> a = [’a’,’b’,’c’] # a list >>> somefunc(a, 1, ’surprise’) >>> print a

[’a’, ’surprise’, ’c’]

>>> a = {’build’ : ’yes’, ’install’ : ’no’}

>>> somefunc(a, ’copy’, True) # add key in a >>> print a

{’install’: ’no’, ’copy’: True, ’build’: ’yes’}

Doing the same with a tuple, which is an immutable object, is not successful:

* >>> a = (’a’, ’b’, ’c’)

>>> somefunc(a, 1, ’surprise’)  
 ...  
 TypeError: object doesn’t support item assignment  
   
See also comments on mutable and immutable types on page 85.  
 Instances of user-defined classes can also be modified in-place. Here is an outline of how we can change a class instance argument in a call by reference  
   
fashion:  
   
class A:  
 def \_\_init\_\_(self, value):  
   
self.int = value  
 self.dict = {’a’: self.int, ’b’: ’some string’}  
   
 def modify(x):

x.int = 2

* x.dict[’b’] = ’another string’  
     
  a1 = A(4)  
   modify(a1)  
   print ’int=%d dict=%s’ % (a1.int, a1.dict)  
     
  The print statement results in  
   int=2 dict={’a’: 4, ’b’: ’another string’}  
     
  showing that the data in the a1 instance have been modified by the modify function.  
     
  Our next example concerns a swap function that swaps the contents of two variables. A Fortran programmer may attempt to write something like  
     
  >>> def swap(a, b):  
   tmp = b; b = a; a = tmp;  
     
  >>> a = 1.2; b = 1.3;  
   >>> swap(a, b)  
   >>> a, b # has a and b been swapped? (1.2, 1.3) # no...

The a and b inside swap initially hold references to objects containing the numbers 1.2 and 1.3, respectively. Then, the local variables a and b are rebound to other float objects inside the function. At return the local a and b are destroyed and no effect of the swapping is experienced in the calling code. The right way to implement the swap function in Python is to return the output variables, in this cased a swapped pair12:

>>> def swap(a, b):

return b, a # return tuple (b, a)

>>> a = 1.2; b = 1.3;

>>> a, b = swap(a, b)

>>> a, b # has a and b been swapped? (1.3, 1.2) # yes!

3.3.5 Treatment of Input and Output Arguments

Chapter 3.3.4 outlines some ways of performing call by reference in Python. We should mention that the Pythonic way of writing functions aims at us- ing function arguments for input variables only. Output variables should be returned. Even in the cases we send in a list, dictionary, or class instance to a function, and modifications to the variable will be visible outside the function, the modified variable is normally returned. There are of course ex- ceptions from this style. One frequent case is functions called by os.path.walk or find (see Chapter 3.4.7). The return value of those functions is not handled by the calling code so any update of the user-defined argument must rely on call by reference.

Consider a function for generating a list of n random normally distributed numbers in a function. Fortran programmers would perhaps come up with the solution

def ngauss(r, n):

for i in range(n):

r[i] = random.gauss(0,1)

r = [0.0]\*10 # make list of 10 items, each equal to 0.0 ngauss(r, len(r))

This works well, but the more Pythonic version creates the list inside the function and returns it:

def ngauss(n):

return [random.gauss(0,1) for i in range(n)]

r = ngauss(10)

12 This swap operation is more elegantly expressed directly as b,a=a,b or (b,a)=(a,b) or [b,a]=[a,b] instead of calling a swap function.

There is no efficiency loss in returning a possibly large data structure, since only the reference to the structure is actually returned. In case a function produces several arrays, say a, b, and c, these are just returned as a tuple (a,b,c). We remark that for large n one should in the present example apply Numerical Python to generate a random array, see Chapter 4.3.1. Such a solution runs 25 times faster than ngauss.

Multiple Lists as Arguments. Sending several lists or dictionaries to a func- tion poses no problem: just send the variables separated by commas. We men- tion this point since programmers coming from Perl will be used to working with explicit reference variables when sending multiple arrays or hashes to a subroutine.

3.3.6 Function Objects

Lambda Functions. Python offers anonymous inline functions known as lambda functions. The construction

lambda <args>: <expression>

is equivalent to a function with <args> as arguments and <expression> as

return value:

def somefunc(<args>):

return <expression>

For example,

lambda x, y, z: 3\*x + 2\*y - z

is a short cut for

def somefunc(x, y, z): return 3\*x + 2\*y - z

Lambda functions can be used in places where we expect variables. Say we have a function taking another function as argument:

def fill(a, f):

n = len(a); dx = 1.0/(n-1) for i in range(n):

x = i\*dx

a[i] = f(x)

A lambda function can be used for the f argument: fill(a, lambda x: 3\*x\*\*4)

This is equivalent to

def somefunc(x):

return 3\*x\*\*4

fill(a, somefunc)

Callable Instances. Functions can also be represented as methods in class in- stances. A particular useful construction is instances with a \_\_call\_\_ method, as explained on page 100. Such instances can be called as ordinary functions and store extra information in attributes.

3.4 Working with Files and Directories

Python has extensive support for manipulating files and directories. Although such tasks can be carried out by operating system commands from Chap- ter 3.1.3, the built-in Python functions for file and directory manipulation work in the same way on Unix, Windows, and Macintosh. Chapter 3.4.1 contains Python functionality for listing files (i.e., the counterparts to the Unix ls and Windows dir commands). Chapter 3.4.2 describes how to test whether a filename reflects a standard file, a directory, or a link, and how to extract the age and size of a file. Chapter 3.4.3 explains how to remove files and directories, while copying and renaming files are the subjects of Chap- ter 3.4.4 Splitting a complete filepath into the directory part and the filename part is described in Chapter 3.4.5. Finally, Chapters 3.4.6 and 3.4.7 deal with creating directories and moving around in directory trees and processing files.

3.4.1 Listing Files in a Directory

Suppose you want to obtain a list of all files, in the current directory, with extensions .ps or .gif. The glob module is then convenient:

import glob

filelist = glob.glob(’\*.ps’) + glob.glob(’\*.gif’)

This action is referred to as file globbing. The glob function accepts filename specifications written in Unix shell-style wildcard notation. You can look up the documentation of the module fnmatch (used for wildcard matching) to see an explanation of this notation.

To list all files in a directory, use the os.listdir function:

files = os.listdir(r’C:\hpl\scripting\src\py\intro’) # Windows files = os.listdir(’/home/hpl/scripting/src/py/intro’) # Unix # fully cross platform:

files = os.listdir(os.path.join(os.environ[’scripting’],

’src’, ’py’, ’intro’))

files = os.listdir(os.curdir) # all files in the current dir.

files = glob.glob(’\*’) + glob.glob(’.\*’) # equiv. to last line

ile Types

The functions isfile, isdir, and islink in the os.path module are used to test if a string reflects the name of a regular file, a directory, or a link:

print myfile, ’is a’,

if os.path.isfile(myfile):

print ’plain file’ if os.path.isdir(myfile):

print ’directory’

if os.path.islink(myfile):

print ’link’

You can also find the age of a file and its size:

time\_of\_last\_access = os.path.getatime(myfile) time\_of\_last\_modification = os.path.getmtime(myfile) size = os.path.getsize(myfile)

Time is measured in seconds since January 1, 1970. To get the age in, e.g., days since last access, you can say

import time # time.time() returns the current time age\_in\_days = (time.time()-time\_of\_last\_access)/(60\*60\*24)

More detailed information about a file is provided by the os.stat function and various utilities in the stat module:

import stat

myfile\_stat = os.stat(myfile) size = myfile\_stat[stat.ST\_SIZE] mode = myfile\_stat[stat.ST\_MODE] if stat.S\_ISREG(mode):

print ’%(myfile)s is a regular file with %(size)d bytes’ %\ vars()

We refer to the Python Library Reference for complete information about the stat module.

Testing read, write, and execute permissions of a file can be performed by the os.access function:

if os.access(myfile, os.W\_OK):

print myfile, ’has write permission’

if os.access(myfile, os.R\_OK | os.W\_OK | os.X\_OK):

print myfile, ’has read, write, and execute permission’

Such tests are very useful in CGI scripts (see Chapter 7.2).

3.4.3 Removing Files and Directories

Single files are removed by the os.remove function, e.g., os.remove(’mydata.dat’)

An alias for os.remove is os.unlink (which coincides with the traditional Unix and Perl name of a function for removing files). Removal of a collection of files, say all \*.ps and \*.gif files, can be done in this way:

for file in glob.glob(’\*.ps’) + glob.glob(’\*.gif’): os.remove(file)

A directory can be removed by the rmdir command provided that the directory is empty. Frequently, one wants to remove a directory tree full of files, an action that requires the rmtree function from the shutil module13:

shutil.rmtree(’mydir’)

We can easily make a function remove for unified treatment of file and

directory removal. Typical usage may be

remove(’my.dat’) # remove a single file my.dat remove(’mytree’) # remove a single directory tree mytree

# remove several files/trees with names in a list of strings: remove(glob.glob(’\*.tmp’) + glob.glob(’\*.temp’)) remove([’my.dat’,’mydir’,’yourdir’] + glob.glob(’\*.data’))

Here is an implementation of the remove function: def remove(files):

"""Remove one or more files and/or directories."""

if isinstance(files, str): # is files a string?

files = [files] # convert files from a string to a list

if not isinstance(files, list): # is files not a list? <report error>

for file in files:

if os.path.isdir(file):

shutil.rmtree(file) elif os.path.isfile(file):

os.remove(file)

Here is a test of the flexibility of the remove function:

# make 10 directories tmp\_\* and 10 tmp\_\_\* files: for i in range(10):

os.mkdir(’tmp\_’+str(i))

f = open(’tmp\_\_’+str(i), ’w’); f.close()

remove(’tmp\_1’) # tmp\_1 is a directory remove(glob.glob(’tmp\_[0-9]’) + glob.glob(’tmp\_\_[0-9]’))

13 The corresponding Unix command is rm -rf mydir.

As a remark about the implementation of the remove function above, we realize that the test

if not isinstance(files, list):

is actually too strict. What we need is just a sequence of file/directory names to be iterated. Whether the names are stored in a list, tuple, or Numerical Python array is irrelevant. A better test is therefore

if not operator.isSequenceType(files): <report error>

3.4.4 Copying and Renaming Files

Copying files is done with the shutil module: import shutil

shutil.copy(myfile, tmpfile)

# copy last access time and last modification time as well: shutil.copy2(myfile, tmpfile)

# copy a directory tree: shutil.copytree(root\_of\_tree, destination\_dir, True)

The third argument to copytree specifies the handling of symbolic links: True means that symbolic linkes are preserved, whereas False implies that symbolic links are replaced by a physical copy of the file.

Cross-platform composition of pathnames is well supported by Python: os.path.join joins directory and file names with the right delimiter (/ on Unix and Mac OS X, and \ on Windows) and the variables os.curdir and os.pardir represent the current working directory and its parent directory, respectively. A Unix command like

cp ../../f1.c .

can be given the following cross-platform implementation in Python:

shutil.copy(os.path.join(os.pardir,os.pardir,’f1.c’), os.curdir)

The rename function in the os module is used to rename a file: os.rename(myfile, ’tmp.1’) # rename myfile to ’tmp.1’

This function can also be used for moving a file (within the same file system). Here myfile is moved to the directory d:

os.rename(myfile, os.path.join(d, myfile))

Moving files across file systems must be performed by a copy (shutil.copy2)

followed by a removal (os.remove): shutil.copy2(myfile, os.path.join(d, myfile))

os.remove(myfile)

The latter approach to moving files is the safest.

.4.5 Splitting Pathnames

Let fname be a complete path to a file, say /usr/home/hpl/scripting/python/intro/hw.py

Occasionally you need to split such a filepath into the basename hw.py and the directory name /usr/home/hpl/scripting/python/intro. In Python this is accomplished by

basename = os.path.basename(fname) dirname = os.path.dirname(fname)

# or

dirname, basename = os.path.split(fname)

The extension is extracted by the os.path.splitext function, root, extension = os.path.splitext(fname)

yielding ’.py’ for extension and the rest of fname for root. The extension without the leading dot is easily obtained by os.path.splitext(fname)[1][1:]. Changing some arbitrary extension of a file with name f to a new extension

ext can be done by

newfile = os.path.splitext(f)[0] + ext

Here is a specific example:

>>> f = ’/some/path/case2.data\_source’

>>> moviefile = os.path.basename(os.path.splitext(f)[0] + ’.mpg’) >>> moviefile

’case2.mpg’

3.4.6 Creating and Moving to Directories

The os module contains the functions mkdir for creating directories and chdir for moving to directories:

origdir = os.getcwd() # remember where we are newdir = os.path.join(os.pardir, ’mynewdir’) if not os.path.isdir(newdir):

os.mkdir(newdir) # or os.mkdir(newdir,’0755’) os.chdir(newdir)

...

os.chdir(origdir) # move back to the original directory os.chdir(os.environ[’HOME’]) # move to home directory

Suppose you want to create a new directory py/src/test1 in your home directory, but neither py, nor src and test1 exist. Instead of using three consecutive mkdir commands to make the nested directories, Python offers the os.makedirs command, which allows you to create the whole path in one statement:

os.makedirs(os.path.join(os.environ[’HOME’],’py’,’src’,’test1’))

3.4.7 Traversing Directory Trees

The call

os.path.walk(root, myfunc, arg)

traverses a directory tree root and calls myfunc(arg, dirname, files) for each directory name dirname, where files is a list of the filenames in dir (actually obtained from os.listdir(dirname)), and arg is a user-specified argument transferred from the calling code. Unix users will recognize that os.path.walk is the cross-platform Python counterpart to the useful Unix find command.

A trivial example of using os.path.walk is to write out the names of all files in all subdirectories in your home tree. You can try this code segment in an interactive Python shell to get a feeling for how os.path.walk works:

def ls(arg, dirname, files):

print dirname, ’has the files’, files

os.path.walk(os.environ[’HOME’], ls, None)

The arg argument is not needed in this application so we simply provide a None value in the os.path.walk call.

A suitable code segment for creating a list all files that are larger than 1 Mb in the home directory might look as follows:

def checksize1(arg, dirname, files): for file in files:

filepath = os.path.join(dirname, file) if os.path.isfile(filepath):

size = os.path.getsize(filepath) if size > 1000000:

size\_in\_Mb = size/1000000.0 arg.append((size\_in\_Mb, filename))

bigfiles = []

root = os.environ[’HOME’] os.path.walk(root, checksize1, bigfiles) for size, name in bigfiles:

print name, ’is’, size, ’Mb’

We now use arg to build a data structure, here a list of 2-tuples. Each 2-tuple holds the size of the file in megabytes and the complete file path. If arg is to be changed in the function called for each directory, it is essential that arg is a mutable data structure that allows in-place modifications (cf. the call by reference discussion in Chapter 3.3.4).

The dirname argument is the complete path to the currently visited di- rectory, and the names in files are given relative to dirname. The current working directory is not changed during the walk, i.e., the script “stays”

in the directory where the script was started. That is why we need to con- struct filepath as a complete path by joining dirname and file14. To change the current working directory to dirname, just call os.chdir(dirname) in the function that os.path.walk calls for each directory, and recall to set the cur- rent working directory back to its original value at the end of the function (otherwise os.path.walk will be confused):

def somefunc(arg, dirname, files):

origdir = os.getcwd(); os.chdir(dirname) <do tasks>

os.chdir(origdir)

os.path.walk(root, somefunc, arg)

As an alternative to os.path.walk, we can easily write our own function with a similar behavior. Here is a version where the user-provided function is called for each file, not each directory:

def find(func, rootdir, arg=None):

# call func for each file in rootdir

files = os.listdir(rootdir) # get all files in rootdir files.sort(lambda a, b: cmp(a.lower(), b.lower()))

for file in files:

fullpath = os.path.join(rootdir, file) if os.path.islink(fullpath):

pass # drop links...

elif os.path.isdir(fullpath):

find(func, fullpath, arg) # recurse into directory elif os.path.isfile(fullpath):

func(fullpath, arg) # file is regular, apply func else:

print ’find: cannot treat ’, fullpath

The find function above is available in the module scitools.misc. Contrary to the built-in function os.path.walk, our find visits files and directories in case-insensitive sorted order.

We could use find to list all files larger than 1 Mb:

def checksize2(fullpath, bigfiles): size = os.path.getsize(fullpath) if size > 1000000:

bigfiles.append(’%.2fMb %s’ % (size/1000000.0, fullpath))

bigfiles = []

root = os.environ[’HOME’] find(checksize2, root, bigfiles) for fileinfo in bigfiles:

print fileinfo

14 Perl programmers may be confused by this point since the find function in Perl’s File::Find package automatically moves the current working directory through the tree.

The arg argument represents great flexibility. We may use it to hold both input information and build data structures. The next example collects the name and size of all files, with some specified extensions, being larger than a given size. The output is sorted according to file size.

bigfiles = {’filelist’: [], # list of file names and sizes ’extensions’: (’.\*ps’, ’.tiff’, ’.bmp’),

’size\_limit’: 1000000, # 1 Mb

}

find(checksize3, os.environ[’HOME’], bigfiles)

def checksize3(fullpath, arg):

treat\_file = False

ext = os.path.splitext(fullpath)[1]

import fnmatch # Unix shell-style wildcard matching for s in arg[’extensions’]:

if fnmatch.fnmatch(ext, s):

treat\_file = True # fullpath has right extension

size = os.path.getsize(fullpath)

if treat\_file and size > arg[’size\_limit’]:

size = ’%.2fMb’ % (size/1000000.0) # pretty print arg[’filelist’].append({’size’: size, ’name’: fullpath})

# sort files according to size def filesort(a, b):

return cmp(float(a[’size’][:-2]), float(b[’size’][:-2])) bigfiles[’filelist’].sort(filesort) bigfiles[’filelist’].reverse() # decreasing size

for fileinfo in bigfiles[’filelist’]:

print fileinfo[’name’], fileinfo[’size’]

Note the function used to sort the list: each element in bigfiles[’filelist’] is a dictionary, and the size key holds a string where we must strip off the unit Mb (last two characters) and convert to float before comparison.

3.4.8 Exercises

Exercise 3.11. Automate execution of a family of similar commands.

The loop4simviz2.py script from Chapter 2.4 generates a series of direc- tories, with PostScript and PNG plots in each directory (among other files). Suppose you want to convert all the PNG files to GIF format. This can be accomplished by the convert utility that comes with the ImageMagick soft- ware:

convert png:somefile.png gif:somefile.gif

By this command, a PNG file somefile.png is converted to GIF format and stored in the file somefile.gif. Alternatively, you can use the Python Imaging Library (PIL):

import Image

# pngfile: filename for PNG file; giffile: filename for GIF file Image.open(pngfile).save(giffile)

Write a script for automating the conversion of many files. Input data to the script constitute of a collection of directories given on the command line. For each directory, let the script glob \*.png imagefiles and transform each imagefile to GIF format.

To test the script, you can generate some directories with PNG files by running loop4simviz2.py with the following command-line arguments:

b 0 2 0.25 -yaxis -0.5 0.5 -A 4 -noscreenplot

Run thereafter the automated conversion of PNG files to GIF format with command-line arguments tmp\_\* (loop4simviz2.py generates directories with names of the form tmp\_\*). ⋄

Exercise 3.12. Remove temporary files in a directory tree.

Computer work often involves a lot of temporary files, i.e., files that you need for a while, but that can be cleaned up after some days. If you let the name of all such temporary files contain the stem tmp, you can now and then run a clean-up script that removes the files. Write a script that takes the name of a directory tree as command-line argument and then removes all files (in this tree) whose names contain the string tmp.

Hint: Use os.path.walk to traverse the directory tree (see Chapter 3.4.7) and look up Chapter 3.2.8 to see how one can test if a string contains the substring tmp. Avoid giving the script a name containing tmp as the script may then remove itself! Also remember to test the script thoroughly, with the physical removal statement replaced by some output message, before you try it on a directory tree. ⋄

Exercise 3.13. Find old and large files in a directory tree.

Write a function that traverses a user-given directory tree and returns a list of all files that are larger than X Mb and that have not been accessed the last Y days, where X and Y are parameters to the function. Include an option in this function that moves the files to a subdirectory trash under /tmp (you need to create trash if it does not exist).

Hints: Use shutil.copy and os.remove to move the files (and not os.rename, it will not work for moving files across different filesystems). First build a list of all files to be removed. Thereafter, remove the files physically.

To test the script, you can run a script fakefiletree.py (in src/tools), which generates a directory tree (say) tmptree with files having arbitrary age (up to one year) and arbitrary size between 5 Kb and 10 Mb:

fakefiletree.py tmptree

If you find that fakefiletree.py generates too many large files, causing the disk to be filled up, you can take a copy of the script and modify the argu- ments in the maketree function call. Remember to remove tmptree when you have finished the testing. ⋄

Exercise 3.14. Remove redundant files in a directory tree.

Make a script cleanfiles.py that takes a root of a directory tree as ar- gument, traverses this directory tree, and for each file removes the file if the name is among a prescribed set of target names. Target names can be specified in Unix shell-style wildcard notation, for example,

\*tmp\* .\*tmp\* \*.log \*.aux \*.idx \* core a.out \*.blg

If the user has a file called .cleanrc in the home directory, assume that this file contains a list of target names, separated by whitespace. Use a default set of target names in the case the user does not have a .cleanrc file.

With the option --fake, the script should just write the name of the file to be removed to the screen but not perform the physical removal. The options --size X and --age Y cause the script to also write out a list of files that are larger than X Mb or older than Y weeks. The user can examine this list for later removal.

The script file should act both as a module and as an executable script (read about modules in Appendix B.1.1). For traversing the directory tree, use the find function from page 124, available in the scitools.misc module. Make a function add\_file for processing each file found by find:

def add\_file(fullpath, arg): """

Add the given fullpath, to arg[’rm\_files’] if fullpath matches one of the names in the arg[’targetnames’] list. The specification of names in targetnames follow the Unix shell-style wildcard notation (an example may be arg[’targetnames’]=[’tmp\*’, ’\*.log’, ’fig\*.\*ps’]). arg[’rm\_files’] contains pairs (fullpath, info), where info is a string containing the file’s size (in Mb)

and the age (in weeks). In addition, add fullpath to

the arg[’old\_or\_large\_files’] list if the size of the file is larger than arg[’max\_size’] (measured in Mb) or older than arg[’max\_age’] (measured in weeks).

"""

Make another function cleanfiles, employing find and add\_date, for printing the removed files and the old or large candidate files.

Hints: Exercises 3.12 and 3.13 might be a useful starting point. Use the fnmatch module to handle Unix shell-style wildcard notation. It is advan- tageous to store files for removal in a list and the large and/or old files in another list. When the traversal of the directory tree has terminated, files can be physically removed and lists can be printed. To test the script, generate a directory tree using the fakefiletree.py utility mentioned in Exercise 3.13 and comment out the os.remove call.

Exercises B.4–B.11 (starting on page 734) equip the useful cleanfiles.py script with good software engineering habits: user documentation, automatic verification, and a well-organized directory structure packed in a single file.

⋄

Exercise 3.15. Annotate a filename with the current date.

Write a function that adds the current date to a filename. For example, calling the function with the text myfile as argument results in the string myfile\_Aug22\_2010 being returned if the current date is August 22, 2010. Read about the time module in the Python Library Reference to see how information about the date can be obtained. Exercise 3.16 has a useful appli- cation of the function from the present exercise, namely a script that takes backup of files and annotates backup directories with the date. ⋄

Exercise 3.16. Automatic backup of recently modified files.

Make a script that searches some given directory trees for files with certain extensions and copies files that have been modified the last three days to a directory backup/copy-X in your home directory, where X is the current date. For example,

backup.py $scripting/src .ps .eps .tex .xfig tex private

searches the directories $scripting/src, tex, and private for files with exten- sions .ps, .eps, .tex, and .xfig. The files in this collection that have been modified the last three days are copied to $HOME/backup/copy-Aug22\_2010 if the current date is August 22, 2010 ($HOME denotes your home directory). Use the convention that command-line arguments starting with a dot denote extensions, whereas the other arguments are roots in directory trees. Make sure that the copy directory is non-existent if no files are copied.

Store files with full path in the backup directory such that files with identical basenames do not overwrite each other. For example, the file with path $HOME/project/a/file1.dat is copied to

$HOME/backup/copy-Aug22\_2010/home/me/project/a/file1.dat

if the value of HOME equals /home/me.

Hint: Make use of Exercises 3.15, os.path.walk or find from Chapter 3.4.7,

and the movefiles function in scitools.misc (run pydoc to see a documenta- tion of that function).

The files in the backup directory tree can easily be transferred to a mem- ory stick or to another computer. ⋄

Exercise 3.17. Search for a text in files with certain extensions.

Create a script search.py that searches for a specified string in files with prescribed extensions in a directory tree. For example, running

search.py "Newton’s method" .tex .py

means visiting all files with extensions .tex and .py in the current directory tree and checking each file if it contains the string Newton’s method. If the string is found in a line in a file, the script should print the filename, the line number, and the line, e.g.,

someletter.tex:124: when using Newton’s method. This allows

Hint: Chapter 3.2.8 explains how to search for a string within a string. ⋄

Exercise 3.18. Search directories for plots and make HTML report.

Running lots of experiments with the simviz1.py and loop4simviz2.py scripts from Chapters 2.3 and 2.4 results in lots of directories with plots. To get an overview of the contents of all the directories you are asked to develop a utility that

* – traverses a directory tree,
* – detects if a directory contains experiments with the oscillator code (i.e., the directory contains the files sim.dat, case.i, case.png, and case.ps, where case is the name of the directory),
* – loads the case.i file data into a dictionary with parameter names and values,
* – stores the path to the PNG plot together with the dictionary from the previous point as a tuple in a list,
* – applies this latter list to generate an HTML report containing all the PNG plots with corresponding text information about the parameters.  
     
  Test the script on a series of directories as explained in the last paragraph of Exercise 3.11. ⋄  
     
  Exercise 3.19. Fix Unix/Windows Line Ends.  
     
  Text files on MS-DOS and Windows have \r\n at the end of lines, whereas Unix applies only \n. Hence, when moving a Unix file to Windows, line breaks may not be visible in certain editors (Notepad is an example). Similarly, a file written on a Windows system may be shown with a “strange character” at the end of lines in certain editors (in Emacs, each line ends with M). Python strips off the \r character at line ends when reading text files on Windows and adds the \r character automatically during write operations. This means that one can, inside Python scripts, always work with \n as line terminator. For this to be successful, files must be opened with ’r’ or ’w’, not the binary counterparts ’rb’ and ’wb’ (see Chapter 8.3.6).  
     
  Write a script win2unix for converting the line terminator \r\n to \n and another script unix2win for converting \n to \r\n. The scripts take a list of filenames and directory names on the command line as input. For each directory, all files in the tree are to get their line ends fixed. Hint: Open the files in ’rb’ and ’wb’ mode (for binary files) such that \r remains un- changed. Checking that a line ends in \r\n can be done by the code segments if line[-2:] == ’\r\n’ or if line.endswith(’\r\n’).  
     
  Remark. On Macintosh computers, the line terminator is \r. It is easy to write scripts that convert \r to and from the other line terminators. However, conversion from \r must be run on a Mac, because on Unix and Windows

the file object’s readline or readlines functions swallow the whole file as one line since no line terminator (\r\n or \n) is found on these platforms. See Lutz [20, Ch. 5] for more details about line conversions. ⋄