Python\_Scripting\_Computational\_Science\_C02

p50

Chapter 2

Getting Started with Python Scripting

This chapter contains a quick and efficient introduction to scripting in Python with the aim of getting you started with real projects as fast as possible. Our pedagogical strategy for achieving this goal is to dive into examples of relevance for computational scientists and dissect the codes line by line.

The present chapter starts with an extension of the obligatory “Hello, World!” program. The next example covers reading and writing data from and to files, implementing functions, storing data in lists, and traversing list structures. Thereafter we create a script for automating the execution of a simulation and a visualization program. This script parses command- line arguments and performs some operating system tasks such as removing and creating directories. The final example concerns converting a data file format and involves programming with a convenient data structure called dictionary. A more thorough description of the various data structures and program constructions encountered in the introductory examples appears in Chapter 3, together with lots of additional Python functionality.

You are strongly encouraged to download and install the software associated with this book and set up your environment as described in Chapter 1.2 before proceeding. All Python scripts referred to in this introductory chapter are found in the directory **src/py/intro** under the root reflected by the scripting environment variable.

In the work with exercises you may need access to reference manuals. The file **$scripting/doc.html** is a good starting point so you should bookmark this page in your favorite browser. Chapter 3.1.1 provides information on recommended Python documentation to acompany the present book.

2.1 A Scientific Hello World Script

It is common to introduce new programming languages by presenting a trivial program writing “Hello, World!” to the screen. We shall follow this tradition when introducing Python, but since we deal with scripting in a computational science context, we have extended the traditional Hello World program a bit: A number is read from the command line, and the program writes the sine of this number along with the text “Hello, World!”. Providing the number 1.4 as the first command-line argument yields this output of the script:

**Hello, World! sin(1.4)=0.985449729988**

This Scientific Hello World script will demonstrate

– how to work with variables,

– how to initialize a variable from the command line,

– how to call a math library for computing the sine of a number, and

– how to print a combination of numbers and plain text.

The complete script can take the following form in Python:

#!/usr/bin/env python

import sys, math # load system and math module

r = float(sys.argv[1]) # extract the 1st command-line argument

s = math.sin(r)

print "Hello, World! sin(" + str(r) + ")=" + str(s)

2.1.1 Executing Python Scripts

Python scripts normally have the extension .py, but this is not required. If the code listed above is stored in a file hw.py, you can execute the script by the command

**python hw.py 1.4**

This command specifies explicitly that a program python is to be used to interpret the contents of the hw.py file. The number 1.4 is a command-line argument to be fetched by the script.

For the python hw.py ... command to work, you need to be in a console window, also known as a terminal window on Unix, and as a command prompt or MS-DOS prompt on Windows. The Windows habit of double-clicking on the file icon does not work for scripts requiring command-line information, unless you have installed PythonWin.

In case the file is given execute permission1 on a Unix system, you can also run the script by just typing the name of the file:

./hw.py 1.4

or

**hw.py 1.4**

if you have a dot (.) in your path2.

On Windows you can write just the filename hw.py instead of python hw.py

if the .py is associated with a Python interpreter (see Appendix A.2). When you do not precede the filename by python on Unix, the first line of the script is taken as a specification of the program to be used for interpreting the script. In our example the first line reads

1 This is achieved by the Unix command chmod a+x hw.py.

2 There are serious security issues related to having a dot, i.e., the current working directory, in your path. Check out the site policy with your system administrator.

**#!/usr/bin/env python**

This particular heading implies interpretation of the script by a program named python. In case there are several python programs (e.g., different Python versions) on your system, the first python program encountered in the directories listed in your PATH environment variable will be used3. Executing ./hw.py with this heading is equivalent to running the script as python hw.py. You can run **src/py/examples/headers.py** to get a text explaining the syntax of headers in Python scripts. For a Python novice there is no need to understand the first line. Simply make it a habit to start all scripts with this particular line.

2.1.2 Dissection of the Scientific Hello World Script

The first real statement in our Hello World script is

**import sys, math**

meaning that we give our script access to the functions and data structures in the system module and in the math module. For example, the system module sys has a list argv that holds all strings on the command line. We can extract the first command-line argument using the syntax

**r = sys.argv[1]**

Like any other Python list (or array), sys.argv starts at 0. The first element, sys.argv[0], contains the name of the script file, whereas the rest of the elements hold the arguments given to the script on the command line.

As in other dynamically typed languages there is no need to explicitly declare variables with a type. Python has, however, data structures of different types, and sometimes you need to do explicit type conversion. Our first script illustrates this point. The data element sys.argv[1] is a string, but r is supposed to be a floating-point number, because the sine function expects a number and not a string. We therefore need to convert the string sys.argv[1] to a floating-point number:

r = float(sys.argv[1])

Thereafter, math.sin(r) will call the sine function in the math module and return a floating-point number, which we store in the variable s.

At the end of the script we invoke Python’s print function:

print "Hello, World! sin(" + str(r) + ")=" + str(s)

The print function automatically appends a newline character to the output string. Observe that text strings are concatenated by the **+** operator and that the floating-point numbers r and s need to be converted to strings, using the **str** function, prior to the concatenation (i.e., addition of numbers and strings is not supported).

We could of course work with r and s as string variables as well, e.g.,

**r = sys.argv[1]**

**s = str(math.sin(float(r)))**

**print "Hello, World! sin(" + r + ")=" + s**

Python will abort the script and report run-time errors if we mix strings and floating-point numbers. For example, running

r = sys.argv[1]

s = math.sin(r) # sine of a string...

results in

Traceback (most recent call last): File "./hw.py", line 4, in ?

s = math.sin(r)

TypeError: illegal argument type for built-in operation

So, despite the fact that we do not declare variables with a specific type, Python performs run-time checks on the type validity and reports inconsistencies.

The **math** module can be imported in an alternative way such that we can avoid prefixing mathematical functions with math:

# import just the sin function from the math module:

from math import sin

# or import all functions in math:

from math import \*

s = sin(r)

Using **import math** avoids name clashes between different modules, e.g., the sin function in math and a sin function in some other module. On the other hand, from math import \* enables writing mathematical expressions in the familiar form used in most other computer languages.

The string to be printed can be constructed in many different ways. A popular syntax employs variable interpolation, also called variable substitution. This means that Python variables are inserted as part of the string. In our original hw.py script we could replace the output statement by

print "Hello, World! sin(%(r)g)=%(s)12.5e" % vars()

The syntax %(r)g indicates that a variable with name r is to be substituted in the string, written in a format described by the character g. The **g** format implies writing a floating-point number as compactly as possible, i.e., the output space is minimized. The text %(s)12.5e means that the value of the variable s is to be inserted, written in the 12.5e format, which means a floating-point number in scientific notation with five decimals in a field of total width 12 characters. The final **% vars()** is an essential part of the string syntax, but there is no need to understand this now4. An example of the output is

**Hello, World! sin(1.4)= 9.85450e-01**

A list of some common format statements is provided on page 80.

Python also supports the output format used in the popular “printf” family of functions in C, Perl, and many other languages. The names of the variables do not appear inside the string but are listed after the string:

print "Hello, World! sin(%g)=%12.5e" % (r,s)

If desired, the output text can be stored in a string prior to printing, e.g.,

output = "Hello, World! sin(%g)=%12.5e" % (r,s)

print output

This demonstrates that the printf-style formatting is a special type of string specification in Python5.

Exercise 2.1. Become familiar with the electronic documentation.

Write a script that prints a uniformly distributed random number between −1 and 1. The number should be written with four decimals as implied by the **%.4f** format.

To create the script file, you can use a standard editor such as Emacs or Vim on Unix-like systems. On Windows you must use an editor for pure text files – Notepad is a possibility, but I prefer to use Emacs or the “IDLE” editor that comes with Python (you usually find IDLE on the start menu, choose File–New Window to open up the editor). IDLE supports standard key bindings from Unix, Windows, or Mac (choose Options–Configure IDLE... and Keys to specify the type of bindings).

The standard Python module for generation of uniform random numbers is called random. To figure out how to use this module, you can look up the description of the module in the Python Library Reference [37]. Load the file $scripting/doc.html into a web browser and click on the link Python Library Reference: Index. You will then see the index of Python functions, modules, data structures, etc. Find the item “random (standard module)” in the index and follow the link. This will bring you to the manual page for the random module. In the bottom part of this page you will find information about functions for drawing random numbers from various distributions (do not use the classes in the module, use plain functions). Also apply pydoc to look up documentation of the random module: just write pydoc random on the command line.

Remark: Do not name the file with this script random.py. This will give a name clash with the Python module random when you try to import that module (your own script will be imported instead). ⋄

2.2 Working with Files and Data

Let us continue our Python encounter with a script that has some relevance for the computational scientist or engineer. We want to do some simple mathematical operations on data in a file. The tasks in such a script include reading numbers from a file, performing numerical operations on them, and then writing the new numbers to a file again. This will demonstrate

– file opening, reading, writing, and closing,

– how to define and call functions,

– loops and if-tests, and

– how to work with lists and arrays.

We shall also show how Python can be used for interactive computing and how this can be combined with a debugger for detecting programming errors.

**2.2.1 Problem Specification**

Suppose you have a data file containing a curve represented as a set of (x, y) points and that you want to transform all the y values using some function f(y). That is, we want to read the data file with (x,y) pairs and write out a new file with (x,f(y)) pairs. Each line in the input file is supposed to contain one x and one y value. Here is an example of such a file format:

0.0 3.2

0.5 4.3

1.0 8.3333

2.5 -0.25

The output file should have the same format, but the f(y) values in the second column are to be written in scientific notation, in a field of width 12 characters, with five decimals (i.e., the number −0.25 is written as -2.50000E-01).

The script, called datatrans1.py, can take the input and output data files as command-line arguments. The usage is hence as follows:

**python datatrans1.py infile outfile**

Inside the script we need to do the following tasks:

1. read the input and output filenames from the command line,

2. open the input and output files,

3. define a function f(y),

4. for each line in the input file:

(a) read the line,

(b) extract the x and y values from the line,

(c) apply the function f to y,

(d) write out x and f(y) in the proper format.

First we present the complete script, and thereafter we explain in detail what is going on in each statement.

2.2.2 The Complete Code

#!/usr/bin/env python import sys, math

try:

infilename = sys.argv[1]; outfilename = sys.argv[2]

except:

print "Usage:",sys.argv[0], "infile outfile"; sys.exit(1)

ifile = open( infilename, ’r’) # open file for reading ofile = open(outfilename, ’w’) # open file for writing

def myfunc(y):

if y >= 0.0:

return y\*\*5\*math.exp(-y)

else:

return 0.0

# read ifile line by line and write out transformed values: for line in ifile:

pair = line.split()

x = float(pair[0]); y = float(pair[1]) fy = myfunc(y) # transform y value ofile.write(’%g %12.5e\n’ % (x,fy))

ifile.close(); ofile.close()

The script is stored in src/py/intro/datatrans1.py. Recall that this path is relative to the scripting environment variable, see Chapter 1.2.

2.2.3 Dissection

The most obvious difference between Python and other programming languages is that the indentation of the statements is significant. Looking, for example, at the for loop, a programmer with background in C, C++, Java, or Perl would expect braces to enclose the block inside the loop. Other languages may have other “begin” and “end” marks for such blocks. However, Python employs just indentation6.

The script needs two modules: sys and math, which we load in the top of the script. Alternatively, one can load a module at the place where it is first needed.

The next statement contains a try-except block, which is the preferred Python style for handling potential errors. We want to load the first two command-line arguments into two strings. However, it might happen that the user of the script failed to provide two command-line arguments. In that case, subscripting the sys.argv list leads to an index out of bounds error, which causes Python to report this error and abort the script. This may not be exactly the behavior we want: if something goes wrong with extracting command-line arguments, we assume that the script is misused. Our recovery from such misuse consists of printing a usage message before terminating the script. In the implementation, we first try to execute some statements in a try block, and then we recover from a potential error in an except block:

try:

infilename = sys.argv[1]; outfilename = sys.argv[2]

except:

print "Usage:",sys.argv[0], "infile outfile"; sys.exit(1)

As soon as any error7 occurs in the try block, the program jumps to the except block. This is recognized as exception handling in Python, a topic which is covered in more detail in Chapter 8.8.

The name of the script being executed is stored in sys.argv[0], and this information is used in the usage message. Calling the function sys.exit aborts the script. Any integer argument to the sys.exit function different from 0 signifies exit due to an error. The value of the integer argument to sys.exit is available in the environment that executes the script and can be used to check if the execution of the script was successful. For example, in a Unix environment, the variable $? contains the value of the argument to sys.exit. If $? is different from 0, the execution of the last command was unsuccessful.

Observe that more than one Python statement can appear at the same line if a semi-colon is used as separator between the statements. You do not need to end a statement with semi-colon if there is only one statement on the line.

* 6 A popular Python slogan reads “life is happier without braces”. I am not com- pletely sure – no braces imply nicely formatted code, but you must be very careful with the indentation when inserting if tests or loops in the middle of a block. Using a Python-aware editor (like Emacs) to adjust indentation of large blocks of code has been essential for me.  
     
  7 We have for simplicity at this introductory stage just tested for any error in the except block. See Exercise 2.7 for comments and how the error testing should be improved.

A file is opened by the open function, taking the filename as first argument and a read/write indication (’r’ or ’w’) as second argument:

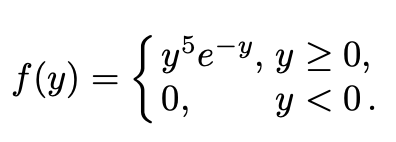
**ifile = open( infilename, ’r’) # open file for reading**

**ofile = open(outfilename, ’w’) # open file for writing**

The open function returns a Python file object that we use for reading from or writing to a file.

At this point we should mention that there is no difference between single and double quotes when defining strings. That is, ’r’ is the same as "r". This is true also in printf-style formatted strings and when using variable interpo- lation. There are other ways of specifying strings as well, and an overview is provided on page 95.

The next block of statements regards the implementation of a function



Such a function, here called myfunc, can in Python be coded as

def myfunc(y):

if y >= 0.0:

return y\*\*5\*math.exp(-y)

else:

return 0.0

A shorter syntax is also possible:

**def myfunc(y):**

**return (y\*\*5\*math.exp(-y) if y >= 0 else 0.0)**

Any function in Python must be defined before it can be called. The file is read line by line using the following construction:

**for line in ifile:**

**# process line**

Python code written before version 2.2 became available applies another con- struction for reading a file line by line:

**while 1:**

**line = ifile.readline()**

**if not line: break # jump out of the loop # process line**

This construction is still useful in many occasions. Each line is read using the file object’s readline function. When the end of the file is reached, readline returns an empty string, and we need to jump out of the loop using a break statement. The termination condition is hence inside the loop, not in the while test (actually, the while 1 implies a loop that runs forever, unless there is a break statement inside the loop).

The processing of a line consists of splitting the text into an x and y value, modifying the y value by calling myfunc, and finally writing the new pair of values to the output file. The splitting of a string into a list of words is accomplished by the split operation

pair = line.split()

Python string objects have many built-in functions, and split is one of them. The split function returns in our case a list of two strings, containing the x and y values. The variable pair is set equal to this list of two strings. However, we would like to have x and y available as floating-point numbers, not strings, such that we can perform numerical computations. An explicit conversion of the strings in pair to real numbers x and y reads

x = float(pair[0]); y = float(pair[1])

We can then transform y using our mathematical function myfunc: fy = myfunc(y)

Thereafter, we write x and fy to the output file in a specified format: x is written as compactly as possible (%g format), whereas fy is written in scientific notation with 5 decimals in a field of width 12 characters (%12.5e format):

ofile.write(’%g %12.5e\n’ % (x, fy))

One should notice a difference between the print statement (for writing to standard output) and a file object’s write function (for writing to files): print automatically adds a newline at the end of the string, whereas write dumps the string as is. In the present case we want each pair of curve points to appear on separate lines so we need to end each string with newline, i.e., \n.

2.2.4 Working with Files in Memory

Instead of reading and processing lines one by one, scripters often load the whole file into a data structure in memory as this can in many occasions simplify further processing. In our next version of the script, we want to (i) read the file into a list of lines, (ii) extract the x and y numbers from each line and store them in two separate floating-point arrays x and y, and (iii) run through the x and y arrays and write out the transformed data pairs. This version of our data transformation example will hence introduce some basic concepts of array or list processing. In a Python context, array and list often mean the same thing, but we shall stick to the term list. We reserve the term array for data structures that are based on an underlying contiguous memory segment (i.e., a plain C array). Such data structures are available in the Numerical Python package and are well suited for efficient numerical computing. A taste is given in Chapters 2.2.5 and 2.2.6, while Chapter 4.1 contains more comprehensive information.

Loading the file into a list of lines is performed by the statement

**lines = ifile.readlines()**

Storing the x and y values in two separate lists can be realized with the following loop:

**x = []; y = [] # start with empty lists for line in lines:**

**xval, yval = line.split() x.append(float(xval)); y.append(float(yval))**

The first line creates two empty lists x and y. One always has to start with an empty list before filling in entries with the append function (Python will give an error message in case you forget the initialization). The statement for line in lines sets up a loop where, in each pass, line equals the next entry in the lines list. Splitting the line string into its individual words is accomplished as in the first version of the script, i.e., by line.split(). However, this time we illustrate a different syntax: individual variables xval and yval are listed on the left-hand side of = and assigned values from the sequence of elements in the list on the right-hand side. The next line in the loop converts the strings xval and yval to floating-point variables and appends these to the x and y lists.

Running through the x and y lists and transforming the y values can be implemented as a C-style for loop over an index:

**for i in range(0, len(x), 1):**

**fy = myfunc(y[i]) # transform y value**

**ofile.write(’%g %12.5e\n’ % (x[i], fy))**

The **range(from, to, step)** function returns a set of integers, here to be used as loop counters, starting at from and ending in to-1, with steps as indicated by step. Calling range with only one value implies the very frequently en- countered case where from is 0 and step is 1. Utilizing range with a just single argument, we could in the present example write for i in range(len(x)).

The complete alternative version of the script appears in datatrans2.py in the directory src/py/intro.

If your programming experience mainly concerns Fortran and C, you prob- ably see already now that Python programs are much shorter and simpler because each statement is more powerful than what you are used to. You might be concerned with efficiency, and that topic is dealt with in the next paragraph.

2.2.5 Array Computing

Sometimes we want to load file data into arrays in a script and perform nu- merical computing with the arrays. We exemplified this in the datatrans2.py script in Chapter 2.2.4. However, there are Python tools that allows more ef- ficient and convenient “Matlab-style” array computing. These tools are based on an extension to Python, called Numerical Python, or just NumPy, which is presented in Chapter 4. In the present section we shall just indicate how we can load array data in a file into NumPy arrays and compute with them.

In the datatrans2.py script we have the file data in lists x and y. These can be turned into NumPy arrays by the statements

**from numpy import \***

**x = array(x); y = array(y) # convert lists to efficient arrays**

Using the file reading tools from Chapter 4.3.6, available through the module scitools.filetable, we can read tabular numerical data in a file directly into NumPy arrays more compactly than we managed in the datatrans2.py script:

**import scitools.filetable**

**f = open(infilename, ’r’)**

**x, y = scitools.filetable.read\_columns(f) f.close()**

Here, x and y are NumPy arrays holding the first and second column of data in the file, respectively.

We may now compute directly with the x and y arrays, e.g., scale the x coordinates by a factor of 10 and transform the y values according to the formula 2y + 0.1 · sin x:

**x = 10\*x**

**y = 2\*y + 0.1\*sin(x)**

These statements are more compact and much more efficient than writing the equivalent loops with indexing:

for i in range(len(x)):

x[i] = 10\*x[i]

for i in range(len(x)):

y[i] = 2\*y[i] + 0.1\*sin(x[i])

We can also compute y with the aid of a function:

**def transform(x, y):**

**return 2\*y + 0.1\*sin(x) y = transform(x, y)**

This **transform** function works with both scalar and array arguments. With Numerical Python available, (most) arithmetic expressions work with scalars and arrays. However, our myfunc function from the datatrans1.py script in Chapter 2.2.2 does unfortunately not work with array arguments because of the if test. The cause of this problem and a remedy is explained in detail in Chapter 4.2.

Writing the x and new y back to a file again can also utilize the tools from from Chapter 4.3.6:

**f = open(outfilename, ’w’) scitools.filetable.write\_columns(f, x, y) f.close()**

Here is another typical action, where we generate a coordinate array in the script and compute curves:

**x = linspace(0, 1, 1001) # 0.0, 0.001, 0.002, ..., 1.0 y1 = sin(2\*pi\*x)**

**y2 = y1 + 0.2\*sin(30\*2\*pi\*x)**

Many more details about such array computing are found in Chapter 4. We can also quickly plot the data:

**from scitools.easyviz import \***

**plot(x, y1, ’b-’, x, y2, ’r-’, legend=(’sine’, ’sine w/noise’),**

**title=’plotting arrays’, xlabel=’x’, ylabel=’y’) hardcopy(’tmp1.ps’) # dump plot to file**

You can type pydoc scitools.easyviz to get more information about Easyviz, a unified interface to various popular plotting packages. Easyviz offers a sim- ple Matlab-like interface to curve plotting, see Chapter 4.3.3.

The statements above are collected in a script called datatrans3.py. A modified script, where the arrays can be sent to a version of the myfunc function from datatrans1.py, is realized as datatrans3a.py.

2.2.6 Interactive Computing and Debugging

**IPython**. Instead of collecting Python statements in scripts and executing the scripts, you can run commands interactively in a Python shell. There are many types of Python shells, and all of them make Python behave much like interactive computing environments such as IDL, Maple, Mathematica, Matlab, Octave, Scilab, and S-PLUS/R. I recommend to use a particularly powerful Python shell called IPython. Just write ipython on the command line to invoke this shell. After the In [1]: prompt you can execute any valid Python statement or get the result of any valid Python expression. Here are some examples on using the shell as calculator:

In [1]:3\*4-1

Out[1]:11

In [2]:from math import \* In [3]:x = 1.2

In [4]:y = sin(x)

In [5]:x

Out[5]:1.2

**In [6]:y Out[6]:0.93203908596722629**

**In [7]:\_ + 1 Out[7]:1.93203908596722629**

Observe that just writing the name of a variable dumps its value to the screen. The \_ variable holds the last output, \_\_ holds the next last output, and \_X holds the output from input command no. X.

Help on Python functionality is available as

**In [8]:help math.floor In [9]:help str.split**

With the arrows you can recall previous commands. In the session above, we can hit the up arrow four times to recall the assignment to x, edit this command to x=0.001, hit the up arrow four times to recall the computation of y and press return to re-run this command, and then write y to see the result (sin 0.001).

**Invoking a Debugger**. With the run command you can also execute script files inside IPython:

**In [1]:run datatrans3.py .datatrans\_infile tmp1**

This is very useful if errors arise because IPython can automatically in- voke Python’s standard debugger pdb when an exception is raised. Let us demonstrate the principle by inserting a possibly erroneous statement in the datatrans3.py file (the file with the error is named datatrans3\_err.py):

**def f(x):**

**p = x+1**

**p[10] = 0**

**return p**

**x = f(x)**

If the array x has length less than 11, the assignment to p[10] will raise an exception (IndexError). Write

**In [1]:%pdb on**

to make IPython invoke the debugger automatically after an exception is raised. When we run the script and an exception occurs, we get a nice printout that illustrates clearly the call sequence leading to the exception. In the present case we see that the exception arises at the line p[10] = 0, and we can thereafter dump the contents of p and check its length. The session looks like this:

I**n [23]:run datatrans3\_err.py .datatrans\_infile tmp1 /some/path/src/py/intro/datatrans3\_err.py**

**19**

**20 ---> 21 22**

**p[10] = 0**

**return p**

**f(x) # leads to an exception**

**if len(x) < 11**

**in f(x)**

**if len(x) < 11**

**IndexError:**

**> /some/path/src/py/intro/datatrans3\_err.py(19)f() -> p[10] = 0**

**(Pdb) print p**

**[ 2. 3. 4. 5.1]**

**(Pdb) len(p)**

**4**

After the debugger’s (Pdb) prompt, writing print var or just p var prints the contents of the variable var. This is often enough to uncover bugs, but pdb is a full-fledged debugger that allows you to execute the code statement by statement, or set break points, view source code files, examine variables, execute alternative statements, etc. You use run -d to start the pdb debugger in IPython:

**In [24]:run -d datatrans3.py .datatrans\_infile tmp1 ...**

**(Pdb)**

At the (Pdb) prompt you can run pdb commands, say s or step for executing one statement at a time, or the alternative n or next command which does the same as s except that pdb does not step into functions (just the call is performed). Here is a sample session for illustration:

**(Pdb) s**

**> /home/work/scripting/src/py/intro/datatrans3.py(11)?() -> import sys**

**(Pdb) s**

**> /home/work/scripting/src/py/intro/datatrans3.py(12)?() -> try:**

**(Pdb) s**

**> /home/work/scripting/src/py/intro/datatrans3.py(13)?() -> infilename = sys.argv[1]; outfilename = sys.argv[2] ...**

**(Pdb) s**

**> /home/work/scripting/src/py/intro/datatrans3.py(20)?() -> x, y = scitools.filetable.read\_columns(f)**

**(Pdb) n**

**> /home/work/scripting/src/py/intro/datatrans3.py(21)?()**

**x =**

**23 x = 10\*x**

/some/path/src/py/intro/datatrans3\_err.py 17 def f(x):

18 ---> 19 20

21 x =

p=x+1

p[10] = 0

return p

-> f.close()

(Pdb) x

Out[25]:array([ 0.1, 0.2, 0.3, 0.4])

A nice introduction to pdb is found in Chapter 9 of the Python Library Ref- erence (you may follow the link from the pdb item in the index). I encourage you to learn some basic pdb commands and use pdb on or run -d as illustrated above – this makes debugging Python scripts fast and effective.

A script can also invoke an interactive mode at the end of the code such that you can examine variables defined, etc. This is done with the -i option to run (or python -i on the command line):

In [26]:run -i datatrans2.py .datatrans\_infile tmp1

In [27]:y

Out[27]:[1.1000000000000001, 1.8, 2.2222200000000001, 1.8]

This technique is useful if you need an initialization script before you start with interactive work.

IPython can do much more than what we have outlined here. I therefore recommend you to browse through the documentation (comes with the source code, or you can follow the link in doc.html) to see the capabilities of this very useful tool for Python programmers.

**IDLE**. The core Python source code comes with a tool called IDLE (Inte- grated DeveLopment Environment) containing an interactive shell, an editor, a debugger, as well as class and module browsers. The interactive shell works much like IPython, but is less sophisticated. One feature of the IDLE shell and editor is very nice: when you write a function call, a small window pops up with the sequence of function arguments and a help line. The IDLE debug- ger and editor are graphically coupled such that you can watch a step-by-step execution in the editor window. This may look more graphically appealing than using IPython/pdb when showing live demos. More information about the capabilities and usage of IDLE can be obtained by following the “Intro- duction to IDLE” link in doc.html.

There are several other IDEs (Integrated Development Environments) for Python offering editors, debuggers, class browsers, etc. The doc.html file contains a link to a web page with an overview of Python IDEs.

2.2.7 Efficiency Measurements

You may wonder how slow interpreted languages, such as Python, are in comparison with compiled languages like Fortran, C, or C++. I created an input file with 100,000 data points8 and compared small datatrans1.py-like programs in the dynamically typed languages Python, Perl, and Tcl with

8 The script described in Exercise 8.7 on page 356 is convenient for this purpose.

similar programs in the compiled languages C and C++. Setting the execu- tion time of the fastest program (0.9 s) to one time unit, the time units for the various language implementations were as follows9.

C, I/O with fscanf/fprintf: 1.0; Python: 4.3; C++, I/O with fstream: 4.0; C++, I/O with ostringstream: 2.6; Perl: 3.1; Tcl: 10.7. These timings re- flect reality in a relevant way: Perl is somewhat faster than Python, and com- piled languages are not dramatically faster for this type of program. A spe- cial Python version (datatrans3b.py) utilizing Numerical Python and TableIO runs faster than the best C++ implementation (see Chapter 4.3.6 for details of implementations and timings).

One can question whether the comparison here is fair as the scripts make use of the general split functions while the C and C++ codes read the num- bers consecutively from file. Another issue is that the large data set used in the test is likely to be stored in binary format in a real application. Working with binary files would make the differences in execution speed much smaller.

The efficiency tests are automated in datatrans-eff.sh (Bourne shell script) or datatrans-eff.py (Python version) so you can repeat them on other computers.

2.2.8 Exercises

**Exercise 2.2**. Extend Exercise 2.1 with a loop.

Extend the script from Exercise 2.1 such that you draw n random uni- formly distributed numbers, where n is given on the command line, and compute the average of these numbers. ⋄

**Exercise 2.3**. Find five errors in a script.

The file src/misc/averagerandom2.py contains the following Python code:

#!/usr/bin/ env python import sys, random

def compute(n):

i = 0; s = 0

while i <= n:

s += random.random()

i += 1

return s/n

n = sys.argv[1]

print ’average of %d random numbers is %g" % (n, compute(n))

There are five errors in this file – find them! ⋄

9 These and other timing tests in the book were mostly performed with an IBM X30 laptop, 1.2 GHz and 512 Mb RAM, running Debian Linux, Python 2.3, and gcc 3.3.

**Exercise 2.4**. Basic use of control structures.

To get some hands-on experience with writing basic control structures and functions in Python, we consider an extension of the Scientific Hello World script hw.py from Chapter 2.1. The script is now supposed to read an arbitrary number of command-line arguments and write the natural logarithm of each number to the screen. For example, if we provide the command-line arguments

1.0 -0.9 2.1

the script writes out

ln(1) = 0

ln(-0.9) is illegal ln(2.1) = 0.741937

Implement four types of loops over the command-line entries:

– a for r in sys.argv[1:] loop (i.e., a loop over the entries in sys.argv, starting with index 1 and ending with the last valid index),  
   
– a for loop with an integer counter i running over the relevant indices in sys.argv (use the range function to generate the indices),  
   
– a while loop with an integer counter running over the relevant indices in sys.argv,  
   
– an “infinite” while 1: loop of the type shown on page 35, with an integer counter and a try-except block where we break out of the loop when sys.argv[i] is an illegal operation.  
   
Look up the documentation of the math module in the Python Library Ref- erence (index “math”) to see how to compute the natural logarithm of a number. Since the bodies of the loops are quite similar, you should collect the common statement in a function (say print\_ln(r), which converts r to a float object, tests on r>0 and prints the appropriate strings). ⋄  
   
**Exercise 2.5**. Use standard input/output instead of files.  
   
Modify the datatrans1.py script such that it reads its numbers from stan- dard input, sys.stdin, and writes the results to standard output, sys.stdout. You can work with sys.stdin and sys.stdout as the ordinary file objects you already have in datatrans1.py, except that you do not need to open and close them.  
   
You can feed data into the script directly from the terminal window (after you have started the script, of course) and terminate input with Ctrl-D. Alternatively, you can send a file into the script using a pipe, and if desired, redirect output to a file:  
   
**cat inputfile | datatrans1stdio.py > res**   
(datatrans1stdio.py is the name of the modified script.) A suitable input file for testing the script is src/py/intro/.datatrans\_infile. ⋄

**Exercise 2.6**. Read streams of (x, y) pairs from the command line.

Modify the datatrans1.py script such that it reads a stream of (x, y) pairs from the command line and writes the modified pairs (x,f(y)) to a file. The usage of the new script, here called datatrans1b.py, should be like this:

python datatrans1b.py tmp.out 1.1 3 2.6 8.3 7 -0.1675

resulting in an output file tmp.out:

1.1 2.6 7

1.20983e+01

9.78918e+00

0.00000e+00

Hint: Run through the sys.argv array in a for loop and use the range function with appropriate start index and increment. ⋄

**Exercise 2.7**. Test for specific exceptions.

Consider the datatrans1.py script with a typo (sys.arg) in the try block:

**try:**

**infilename = sys.arg[1]; outfilename = sys.argv[2]**

**except:**

**print "Usage:",sys.argv[0], "infile outfile"; sys.exit(1)**

Run this script and observe that whatever you write as filenames, the script aborts with the usage message. The reason is that we test for any exception in the except block. We should rather test for specific exceptions, i.e., the type of errors that we want to recover from in the try block. In the present case we are worried about too few command-line arguments. Read about exceptions in Chapter 8.8 and figure out how the except block is to be modified. Run the modified script and observe the impact of the typo.

Extend the script with an appropriate try-except block around the first open statement. You should test for a specific exception caused by a non- existing input file.

Finally, it is a good habit to write error messages to standard error (sys.stderr) and not standard output (where the print statements go). Make the corresponding modifications of the print statements.

**Exercise 2.8**. Sum columns in a file.

Extend the datatrans1.py script such that you can read a file with an arbitrary number of columns of real numbers. Find the average of the numbers on each line and write to a new file the original columns plus a final column with the averages. All numbers in the output file should have the format 12.6f. ⋄

**Exercise 2.9**. Estimate the chance of an event in a dice game.

What is the probability of getting at least one 6 when throwing two dice? This question can be analyzed theoretically by methods from probability theory (see the last paragraph of this exercise). However, a much simpler and much more general alternative is to let a computer program “throw” two dice a large number of times and count how many times a 6 shows up. Such type of computer experiments, involving uncertain events, is often called Monte Carlo simulation (see also Exercise 4.14).

Create a script that in a loop from 1 to n draws two uniform random numbers between 1 and 6 and counts how many times p a 6 shows up. Write out the estimated probability p/float(n) together with the exact result 11/36. Run the script a few times with different n values (preferably read from the command line) and determine from the experiments how large n must be to get the first three decimals (0.306) of the probability correct.

Use the **random** module to draw random uniformly distributed integers in a specified interval.

The exact probability of getting at least one 6 when throwing two dice can be analyzed as follows. Let A be the event that die 1 shows 6 and let B be the event that die 2 shows 6. We seek P (A ∪ B), which from probability theory equals P(A) + P(B) − P(A ∩ B) = P(A) + P(B) − P(A)P(B) (A and B are independent events). Since P(A) = P(B) = 1/6, the probability becomes 11/36 ≈ 0.306. ⋄

**Exercise 2.10**. Determine if you win or loose a hazard game.

Somebody suggests the following game. You pay 1 unit of money and are allowed to throw four dice. If the sum of the eyes on the dice is less than 9, you win 10 units of money, otherwise you loose your investment. Should you play this game?

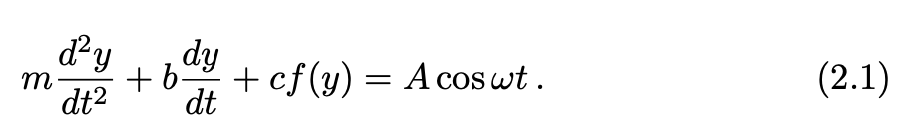
Hint: Use the simulation method from Exercise 2.9. ⋄ 2.3 Gluing Stand-Alone Applications

2.3 ​​Gluing Stand-Alone Applications

One of the simplest yet most useful applications of scripting is automation of manual interaction with the computer. Basically, this means running stand- alone programs and operating system commands with some glue in between. The next example concerns automating the execution of a simulation code and visualization of the results. Such an example is of particular value to a computational scientist or engineer. The simulation code used here in- volves an oscillating system, i.e., solution of an ordinary differential equation, whereas the visualization is a matter of plotting a time series. The mathe- matical simplicity of this application allows us to keep the technical details of the simulation code and the visualization process at a minimum.

2.3.1 The Simulation Code

**Problem Specification**. We consider an oscillating system, say a pendu- lum, a moored ship, or a jumping washing machine. The one-dimensional back-and-forth movement of a reference point in the system is supposed to be adequately described by a function y(t) solving the ordinary differential equation



This equation usually arises from Newton’s second law (or a variant of it: the equation of angular momentum). The first term reflects the mass times the acceleration of the system, the bdy/dt term denotes damping forces, cf(y) is a spring-like force, while Acosωt is an external oscillating force applied to the system. The parameters m, b, c, A, and ω are prescribed constants. Engineers prefer to make a sketch of such a generic oscillating system using graphical elements as shown in Figure 2.1.

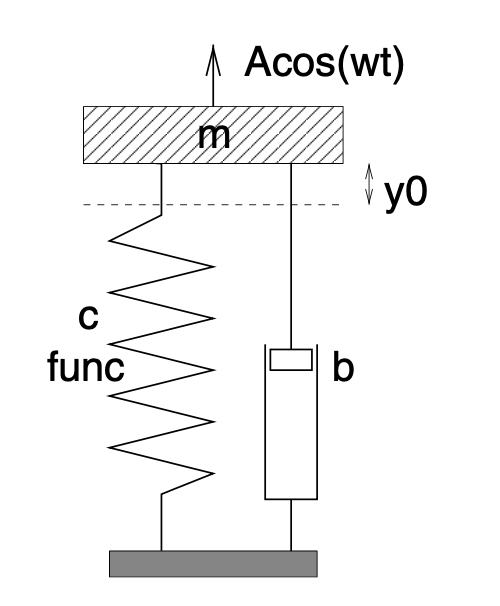
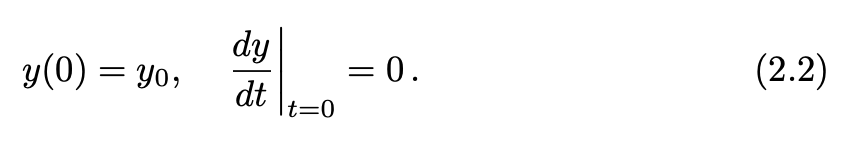


Fig. 2.1. Sketch of an oscillating system. The goal is to compute how the vertical position y(t) of the mass changes in time. The symbols correspond to the names of the variables in and the options to the script performing simulation and visualiza- tion of this system.

Along with the differential equation we need two initial conditions:



This means that the system starts from rest with an initial displacement y0. For simple choices of f(y), in particular f(y) = y, mathematical solution techniques for (2.1) result in simple analytical formulas for y(t), but in general a numerical solution procedure must be applied for solving (2.1). Here we assume that there exists a program oscillator which solves (2.1) using appro- priate numerical methods10. This program computes y(t) when 0 ≤ t ≤ tstop, and the solution is produced at discrete times 0, Δt, 2Δt, 3Δt, and so forth. The Δt parameter controls the numerical accuracy. A smaller value results in a more accurate numerical approximation to the exact solution of (2.1).

**Installing the Simulation Code**. A Fortran 77 version of the oscillator code is found in the directory src/app/oscillator/F77. Try to write oscillator and see if the cursor is hanging (waiting for input). If not, you need to compile, link, and install the program. The Bourne shell script make.sh, in the same di- rectory as the source code, automates the process on Unix system. Neverthe- less, be prepared for platform- or compiler-specific edits of make.sh. The exe- cutable file oscillator is placed in a directory $scripting/$MACHINE\_TYPE/bin, which must be in your PATH variable. Of course, you can place the executable file in any other directory in PATH.

If you do not have an F77 compiler, you can look for implementations of the simulator in other languages in subdirectories of src/app/oscillator. For example, there is a subdirectory C-f2c with a C version of the F77 code automatically generated by the f2c program (an F77 to C source code trans- lator). Since most numerical codes are written in compiled high-performance languages, like Fortran or C, we think it is a point to work with such type of simulation programs in the present section. However, there is also a directory src/app/oscillator/Python containing a Python version, oscillator.py, of the simulator. Copy this file to $scripting/$MACHINE\_TYPE/bin/oscillator if you work on a Unix system and do not get the compiled versions to work properly. Note that the name of the executable file must be oscillator, not oscillator.py, exactly as in the Fortran case, otherwise our forthcoming script will not work. On Windows there is no need to move oscillator.py, see Appendix A.2.

**Simulation Code Usage**. Our simulation code oscillator reads the following parameters from standard input, in the listed order: m, b, c, name of f(y) function, A, ω, y0, tstop, and Δt. The valid names of the implemented f(y) functions are y for f(y) = y, siny for f(y) = siny, and y3 for f(y) = y−y3/6 (the first two terms of a Taylor series for sin y).

The values of the input parameters can be conveniently placed in a file (say) prms:

1.0

0.7

5.0

y

5.0

6.28

0.2

10 Our implementations of oscillator employ a two-stage Runge-Kutta scheme.

30.0 0.05

The program can then be run as

**oscillator < prms**

One may argue that the program is not very user friendly: missing the correct order of the numbers makes the input corrupt. However, the purpose of our script is to add a more user-friendly handling of the input data and avoid the user’s direct interaction with the oscillator code.

The output from the oscillator program is a file sim.dat containing data points (ti, y(ti)), i = 0, 1, 2, . . ., on the solution curve. Here is an extract from such a file:

0.0500 0.2047

0.1000 0.2167

0.1500 0.2328

0.2000 0.2493

0.2500 0.2621

0.3000 0.2674

0.3500 0.2621

0.4000 0.2437

2.3.2 Using Gnuplot to Visualize Curves

The data are easily visualized using a standard program for displaying curves. We shall apply the freely available Gnuplot11 program, which runs on most platforms. One writes gnuplot to invoke the program, and thereafter one can issue the command

**plot ’sim.dat’ title ’y(t)’ with lines**

A separate window with the plot will now appear on the screen, containing the (x, y) data in the file sim.dat visualized as a curve with label y(t).

A PostScript file with the plot is easily produced in Gnuplot:

**set term postscript eps monochrome dashed ’Times-Roman’ 28**

**set output ’myplot.ps’**

followed by the plot command. The plot is then available in the file myplot.ps and ready for inclusion in a report. If you want the output in the PNG format with colored lines, the following commands do the job:

**set term png small**

**set output ’myplot.png’**

11 Exercise2.14explainshoweasyitistoreplaceGnuplotbyMatlabintheresulting script. Exercise 11.1 applies the BLT graph widget from Chapter 11.1.1 instead.

The resulting file **myplot.png** is suited for inclusion in a web page. The vi- sualization of the time series in hardcopy plots is normally improved when reducing the aspect ratio of the plot. To this end, one can try

**set size ratio 0.3 1.5, 1.0**

prior to the plot command. This command should not be used for screen plots. We refer to the Gnuplot manual (see link in doc.html) for more infor- mation on what the listed Gnuplot commands mean and the various available options.

Instead of operating Gnuplot interactively one can collect all the com- mands in a file, hereafter called Gnuplot script. For example,

gnuplot cmd

runs Gnuplot with the commands in the file cmd in a Unix environment. The Gnuplot option -persist is required if we want the plot window(s) on the screen to be visible after the commands in cmd are executed. A standard X11 option -geometry can be used to set the geometry of the window. In the present application with time series it is convenient to have a wide window, e.g., 800 × 200 pixels as specified by the option -geometry 800x200.

Gnuplot behaves differently on Windows and Unix. For example, the name of the Gnuplot script file must be GNUPLOT.INI on Windows, and the existence of such a file implies that Gnuplot reads its commands from this file. I have made two small scripts (see page 687) that comes with this book’s software and makes the gnuplot command behave in almost the same way on Win- dows and Unix. The major difference is that some of the command-line argu- ments on Unix have no effect on Windows. The previously shown examples on running Gnuplot can therefore be run in Windows environments without modifications. This allows us to make a cross-platform script for simulation and visualization.

**2.3.3 Functionality of the Script**

Our goal now is to simplify the user’s interaction with the oscillator and gnuplot programs. With a script simviz1.py it should be possible to adjust the m, b, Δt, and other mathematical parameters through command-line options, e.g.,

**-m 2.3 -b 0.9 -dt 0.05**

The result should be PostScript and PNG plots as well as an optional plot on the screen. Since running the script will produce some files, it is convenient to create a subdirectory and store the files there. The name of the subdirectory and the corresponding files should be adjustable as a command-line option to the script.

Let us list the complete functionality of the script:

1. Set appropriate default values for all input variables.
2. Run through the command-line arguments and set script variables ac- cordingly. The following options should be available: -m for m, -b for b, -c for c, -func for the name of the f(y) function, -A for A, -w for ω, -dt for Δt, -tstop for tstop, -noscreenplot for turning off the plot on the screen12, and -case for the name of the subdirectory and the stem of the filenames of all generated files.
3. Remove the subdirectory if it exists. Create the subdirectory and change the current working directory to the new subdirectory.
4. Make an appropriate input file for the oscillator code.
5. Run the oscillator code.
6. Make a file with the Gnuplot script, containing the Gnuplot commands for making hardcopy plots in the PostScript and PNG formats, and (op- tionally) a plot on the screen.

7. Run Gnuplot.

2.3.4 The Complete Code

#!/usr/bin/env python import sys, math

# default values of input parameters:

m = 1.0; b = 0.7; c = 5.0; func = ’y’; A = 5.0; w = y0 = 0.2; tstop = 30.0; dt = 0.05; case = ’tmp1’ screenplot = True

# read variables from the command line, one by one: while len(sys.argv) > 1:

2\*math.pi

option = sys.argv[1]; if option == ’-m’:

m = float(sys.argv[1]); elif option == ’-b’:

b = float(sys.argv[1]); elif option == ’-c’:

c = float(sys.argv[1]); elif option == ’-func’: func = sys.argv[1];

elif option == ’-A’:

A = float(sys.argv[1]);

elif option == ’-w’:

w = float(sys.argv[1]);

elif option == ’-y0’:

y0 = float(sys.argv[1]);

elif option == ’-tstop’:

tstop = float(sys.argv[1]);

elif option == ’-dt’:

del sys.argv[1]

del sys.argv[1]

del sys.argv[1]

del sys.argv[1]

del sys.argv[1]

del sys.argv[1]

del sys.argv[1]

del sys.argv[1]

del sys.argv[1]

12 Avoiding lots of graphics on the screen is useful when running large sets of ex- periments as we exemplify in Chapter 2.4.

dt = float(sys.argv[1]); elif option == ’-noscreenplot’:

screenplot = False elif option == ’-case’: case = sys.argv[1];

del sys.argv[1]

del sys.argv[1]

else:

print sys.argv[0],’: invalid option’,option sys.exit(1)

# create a subdirectory:

d = case

import os, shutil

if os.path.isdir(d):

shutil.rmtree(d)

os.mkdir(d)

os.chdir(d)

# name of subdirectory

# does d exist?

# yes, remove old directory # make new directory d

# move to new directory d

# make input file to the program:

f = open(’%s.i’ % case, ’w’)

# write a multi-line (triple-quoted) string with # variable interpolation:

f.write("""

%(m)g

%(b)g

%(c)g

%(func)s

%(A)g

%(w)g

%(y0)g

%(tstop)g

%(dt)g

""" % vars())

f.close()

# run simulator:

cmd = ’oscillator < %s.i’ % case # command to run failure = os.system(cmd)

if failure:

print ’running the oscillator code failed\n%s\n%s’ % \ (cmd, output); sys.exit(1)

# make file with gnuplot commands:

f = open(case + ’.gnuplot’, ’w’)

f.write("""

set title ’%s: m=%g b=%g c=%g f(y)=%s A=%g w=%g y0=%g dt=%g’; """ % (case, m, b, c, func, A, w, y0, dt))

if screenplot:

f.write("plot ’sim.dat’ title ’y(t)’ with lines;\n")

f.write("""

set size ratio 0.3 1.5, 1.0;

# define the postscript output format:

set term postscript eps monochrome dashed ’Times-Roman’ 28; # output file containing the plot:

set output ’%s.ps’;

# basic plot command:

plot ’sim.dat’ title ’y(t)’ with lines;

# make a plot in PNG format:

set term png small;

set output ’%s.png’;

plot ’sim.dat’ title ’y(t)’ with lines;

""" % (case, case))

f.close()

# make plot:

cmd = ’gnuplot -geometry 800x200 -persist ’ + case + ’.gnuplot’ failure = os.system(cmd)

if failure:

print ’running gnuplot failed\n%s\n%s’ % \ (cmd, output); sys.exit(1)

You can find the script in **src/py/intro/simviz1.py**.

2.3.5 Dissection

After a standard opening of Python scripts, we start with assigning ap- propriate default values to all variables that can be adjusted through the script’s command-line options. The next task is to parse the command-line arguments. This is done in a while loop where we look for the option in sys.argv[1], remove this list element by a del sys.argv[1] statement, and thereafter assign a value, the new sys.argv[1] entry, to the associated vari- able:

# read variables from the command line, one by one: while len(sys.argv) > 1:

option = sys.argv[1]; if option == ’-m’:

m = float(sys.argv[1]); elif option == ’-b’:

b = float(sys.argv[1]);

... else:

del sys.argv[1]

del sys.argv[1]

del sys.argv[1]

print sys.argv[0],’: invalid option’,option sys.exit(1)

The loop is executed until there are less than two entries left in sys.argv (recall that the first entry is the name of the script, and we need at least one option to continue parsing).

We remark that Python has built-in alternatives to our manual parsing of command-line options: the getopt and optparse modules, see Chapter 8.1.1. Exercise 8.1 asks you to use getopt or optparse in simviz1.py. An alternative tool is developed in Exercise 8.2.

The next step is to remove the working directory d if it exists (to avoid mixing old and new files), create the directory, and move to d. These operating system tasks are offered by Python’s os, os.path, and shutil modules:

d = case

import os, shutil

if os.path.isdir(d):

shutil.rmtree(d)

os.mkdir(d)

os.chdir(d)

# name of subdirectory

# does d exist?

# yes, remove old directory # make new directory d

# move to new directory d

Then we are ready to execute the simulator by running the command

**oscillator < case.i**

where case.i is an input file to oscillator. The filestem case is set by the -case option to the script. Creating the input file is here accomplished by a multi-line Python string with variable interpolation:

f = open(’%s.i’ % case, ’w’) f.write("""

%(m)g

%(b)g

%(c)g

%(func)s

%(A)g

%(w)g

%(y0)g

%(tstop)g

%(dt)g

""" % vars())

f.close()

Triple quoted strings """...""" can span several lines, and newlines are pre- served in the output.

Running an application like oscillator is conveniently done by the func- tion os.system:

cmd = ’oscillator < %s.i’ % case # command to run failure = os.system(cmd)

if failure:

print ’running %s failed’ % cmd; sys.exit(1)

Something went wrong with the command if the function returns a value different from zero13. There are several alternative ways to run a program from a Python script, see Chapter 3.1.3.

Having run the simulator, we are ready for producing plots of the solu- tion. This requires running Gnuplot with a file containing all the relevant commands. First we write the file, this time using a multi-line (triple double quoted) string with standard printf-style formatting:

f.write("""

set title ’%s: m=%g b=%g c=%g f(y)=%s A=%g w=%g y0=%g dt=%g’; """ % (case, m, b, c, func, A, w, y0, dt))

if screenplot:

f.write("plot ’sim.dat’ title ’y(t)’ with lines;\n") f.write("""

set size ratio 0.3 1.5, 1.0;

# define the postscript output format:

set term postscript eps monochrome dashed ’Times-Roman’ 28; # output file containing the plot:

set output ’%s.ps’;

13 Note that if failure is equivalent to if failure != 0.

# basic plot command plot ’sim.dat’ title # make a plot in PNG set term png small; set output ’%s.png’; plot ’sim.dat’ title """ % (case, case)) f.close()

’y(t)’ with lines; format:

’y(t)’ with lines;

Gnuplot accepts comments starting with #, which we here use to make the file more readable. In the next step we run Gnuplot and check if something went wrong:

cmd = ’gnuplot -geometry 800x200 -persist ’ + case + ’.gnuplot’ failure = os.system(cmd)

if failure:

print ’running gnuplot failed’; sys.exit(1)

Let us test the script:

**python simviz1.py -m 2 -case tmp2**

The results are in a new subdirectory tmp2 containing, among other files, the plot tmp2.ps, which is displayed in Figure 2.2. To kill a Gnuplot window on the screen, you can type ’q’ when window is in focus.

With the simviz1.py script at our disposal, we can effectively perform numerical experiments with the oscillating system model since the interface is so much simpler than running the simulator and plotting program manually. Chapter 2.4 shows how to run large sets of experiments using the simviz1.py script inside a loop in another script.

Fig. 2.2. A plot of the solution y(t) of (2.1) as produced by the simviz1.py script.

2.3.6 Exercises

**Exercise 2.11**. Generate an HTML report from the simviz1.py script. Extend the simviz1.py script such that it writes an HTML file containing the values of the physical and numerical parameters, a sketch of the system (src/py/misc/figs/simviz2.xfig.t.gif is a suitable file), and a PNG plot of the solution. In case you are not familiar with writing HTML code, I have made a quick introduction, particularly relevant for this exercise, in the file

src/misc/html-intro/oscillator.html

In Python, you can conveniently generate HTML pages by using multi-line (triple quoted) strings, combined with variable interpolation, as outlined be- low:

htmlfile.write("""

<html>

...

The following equation was solved: <center>

%(m)gDDy + %(b)gDy + %(c)g%(func)s = %(A)gcos(%(w)g\*t), y(0)=%(y0)g, Dy(0)=0

</center>

with time step %(dt)g for times in the interval [0,%(tstop)g].

...

<img src="%(case)s.png">

...

</html>

""" % vars())

It is recommended to design and write the HTML page manually in a separate file, insert the HTML text from the file inside a triple-quoted Python string, and replace relevant parts of the HTML text by variables in the script.

**Exercise 2.12**. Generate a LATEX report from the simviz1.py script.

Extend the simviz1.py script so that it writes a LATEX file containing the values of the physical and numerical parameters, a sketch of the system (src/misc/figs/simviz.xfig.eps is a suitable file), and a PostScript plot of the solution. LATEX files are conveniently written by Python scripts using triple quoted raw strings (to preserve the meaning of backslash). Here is an example:

latexfile.write(r"""

%% Automatically generated LaTeX file \documentclass[11pt]{article}

...

The following equation was solved:

\[ %(m)g\frac{d 2 y}{dt 2} + %(b)\frac{dy}{dt} + %(c)g%(lfunc)s

= %(A)g\cos (%(w)gt), \quad

y(0)=%(y0)g, \frac{dy(0)}{dt}=0\]

with time step $\Delta t = %(dt)g$ for times in the interval $[0,%(tstop)g]$.

...

\end{document}

""" % vars())

The lfunc variable holds the typesetting of func in LATEX (e.g., lfunc is r’\sin y’ if func is siny).

It is smart to write the LATEX page manually in a separate file, insert the LATEX text from the file inside a triple-quoted Python string, and replace parts of the LATEX text by variables in the script.

Comments in LATEX start with %, but this character is normally used for formatting in the write statements, so a double % is needed to achieve the correct formatting (see the first line in the output statement above – only a single % appears in the generated file).

Note that this exercise is very similar to Exercise 2.11. ⋄

**Exercise 2.13**. Compute time step values in the simviz1.py script.

The value of Δt, unless set by the -dt command-line option, could be chosen as a fraction of T , where T is the typical period of the oscillations. T will be dominated by the period of free vibrations in the system, 2π/ c/m, or the period of the forcing term, 2π/ω. Let T be the smallest of these two values and set Δt = T/40 if the user of the script did not apply the -dt option. (Hint: use 0 as default value of dt to detect whether the user has given dt or not.) ⋄

**Exercise 2.14**. Use Matlab for curve plotting in the simviz1.py script.

The plots in the simviz1.py script can easily be generated by another plotting program than Gnuplot. For example, you can use Matlab. Some possible Matlab statements for generating a plot on the screen, as well as hardcopies in PostScript and PNG format, are listed next.

load sim.dat % read sim.dat into a matrix sim plot(sim(:,1),sim(:,2)) % plot 1st column of sim as x, 2nd as y legend(’y(t)’)

title(’test1: m=5 b=0.5 c=2 f(y)=y A=1 w=1 y0=1 dt=0.2’)

outfile = ’test1.ps’; print(’-dps’, outfile) outfile = ’test1.png’; print(’-dpng’, outfile)

The name of the case is in this example taken as test1. The plot statements can be placed in an M-file (Matlab script) with extension .m. At the end of the M-file one can issue the command pause(30) to make the plot live for 30 seconds on the screen. Thereafter, it is appropriate to shut down Matlab by the exit command. The pause command should be omitted when no screen plot is desired.

Running Matlab in the background without any graphics on the screen can be accomplished by the command

matlab -nodisplay -nojvm -r test1

if the name of the M-file is test1.m. To get a plot on the screen, run

matlab -nodesktop -r test1 > /dev/null &

Here, we direct the output from the interactive Matlab terminal session to the “trash can” /dev/null on Unix systems. We also place the Matlab exe- cution in the background (&) since screen plots are associated with a pause command (otherwise the Python script would not terminate before Matlab has terminated).

Modify a copy of the simviz1.py script and replace the use of Gnuplot by Matlab. Hint: In printf-like strings, the character % must be written as %%, because % has a special meaning as start of a format specification. Hence, Matlab comments must start with %% if you employ printf-like strings or variable interpolation when writing the M-file.

2.4 Conducting Numerical Experiments

Suppose we want to run a series of different m values, where m is a physical parameter, the mass of the oscillator, in Equation (2.1). We can of course just execute the simviz1.py script from Chapter 2.3 manually with different values for the -m option, but here we want to automate the process by cre- ating another script loop4simviz1.py, which calls simiviz1.py inside a loop over the desired m values. The loop4simviz1.py script can have the following command-line options:

**m\_min m\_max dm [ options as for simviz1.py ]**

The first three command-line arguments define a sequence of m values, start- ing with m\_min and stepping dm at a time until the maximum value m\_max is reached. The rest of the command-line arguments are supposed to be valid options for simviz1.py and are simply passed on to that script.

Besides just running a loop over m values, we shall also let the script

– generate an HTML report with plots of the solution for each m value and a movie reflecting how the solution varies with increasing m,  
   
– collect PostScript plots of all the solutions in a compact file suitable for printing, and  
   
– run a loop over any input parameter to the oscillator code, not just m.

**2.4.1 Wrapping a Loop Around Another Script**

We start the loop4simviz1.py script by grabbing the first three command-line arguments:

try:

m\_min = float(sys.argv[1])

m\_max = float(sys.argv[2])

dm = float(sys.argv[3])

except IndexError:

print ’Usage:’,sys.argv[0],\

’m\_min m\_max m\_increment [ simviz1.py options ]’ sys.exit(1)

The next command-line arguments are extracted as sys.argv[4:]. The sub- script [4:] means index 4, 5, 6, and so on until the end of the list. These list items must be concatenated to a string before we can use them in the execution command for the simviz1.py script. For example, if sys.argv[4:] is the list [’-c’,’3.2’,’-A’,’10’], the list items must be combined to the string ’-c 3.2 -A 10’. Joining elements in a list into a string, with a specified delimiter, here space, is accomplished by

**simviz1\_options = ’ ’.join(sys.argv[4:])**

We are now ready to make a loop over the m values. Unfortunately, the range function can only generate a sequence of integers, so a for loop over real-valued m values, like for m in range(...), will not work. A while loop is a more appropriate choice:

m = m\_min

while m <= m\_max:

case = ’tmp\_m\_%g’ % m

cmd = ’python simviz1.py %s -m %g -case %s’ % \

(simviz1\_options, m, case) failure = os.system(cmd)

m += dm

Inside the loop, we let the case name of each experiment reflect the value of m. Using this name in the -case option after the user-given options ensures that our automatically generated case name overrides any value of -case provided by the user.

Notice that we run the simviz1.py script by writing python simviz1.py. This construction works safely on all platforms. The simviz1.py file must in this case be located in the same directory as the loop4simviz1.py script, otherwise we need to write the complete filepath of simviz1.py, or drop the python “prefix” and put the simviz1.py script in a directory contained in the PATH variable.

2.4.2 Generating an HTML Report

To make the script even more useful, we could collect the various plots in a common document. For example, all the PNG plots could appear in an HTML14 file for browsing. This is achieved by opening the HTML file, writing a header and footer before and after the while loop, and writing an IMG tag with the associated image file inside the loop:

html = open(’tmp\_mruns.html’, ’w’) html.write(’<HTML><BODY BGCOLOR="white">\n’)

m = m\_min

while m <= m\_max:

case = ’tmp\_m\_%g’ % m

cmd = ’python simviz1.py %s -m %g -case %s’ % \

(simviz1\_options, m, case) failure = os.system(cmd)

html.write(’<H1>m=%g</H1> <IMG SRC="%s">\n’ \

% (m, os.path.join(case, case+’.png’)))

m += dm html.write(’</BODY></HTML>\n’)

One can in this way browse through all the figures in tmp\_mruns.html using a standard web browser.

The previous code segment employs a construction

**os.path.join(case, case+’.png’)**

for creating the correct path to the PNG file in the case subdirectory. The os.path.join function joins its arguments with the appropriate directory sep- arator for the operating system in question (the separator is / on Unix, : on Macintosh, and \ on DOS/Windows, although / works well in paths inside Python on newer Windows systems).

We can also make a PostScript file containing the various PostScript plots. Such a file is convenient for compact printing and viewing of the experi- ments. A Perl script epsmerge (see doc.html for a link) merges Encapsulated PostScript files into a single file. For example,

**epsmerge -o figs.ps -x 2 -y 3 -par file1.ps file2.ps ...**

fills up a file figs.ps with plots file1.ps, file2.ps, and so on, such that each page in figs.ps has three rows with two plots in each row, as specified by the -x 2 -y 3 options. The -par option preserves the aspect ratio of the plots.

In the loop4simviz1.py script we need to collect the names of all the PostScript files and at the end execute the epsmerge command:

14 Check out src/misc/html-intro/oscillator.html and Exercise 2.11 if you are not fa- miliar with basic HTML coding.

psfiles = [] # plot files in PostScript format ...

m = m\_min

while m <= m\_max:

case = ’tmp\_m\_%g’ % m

... psfiles.append(os.path.join(case,case+’.ps’))

...

cmd = ’epsmerge -o tmp\_mruns.ps -x 2 -y 3 -par ’+’ ’.join(psfiles) failure = os.system(cmd)

To make the tmp\_mruns.ps file more widely accessible, we can convert the document to PDF format. A simple tool is the ps2pdf script that comes with Ghostview (gs):

**failure = os.system(’ps2pdf tmp\_mruns.ps’)**

The reader is encouraged to try the loop4simviz1.py script and view the resulting documents. It is quite amazing how much we have accomplished with just a few lines: any number of m values can be tried, each run is archived in a separate directory, and all the plots are compactly collected in documents for convenient browsing. Automating numerical experiments in this way increases the reliability of your work as larger sets of experiments are encouraged and there are no questions about which input parameters that produced a particular plot.

Exercise 2.15. Combine curves from two simulations in one plot.

Modify the simviz1.py script such that when func is different from y, the plot contains two curves, one based on computations with the func function and one based on computations with the linear counterpart (func equals y). It is hence easy to see the effect of a nonlinear spring force. The following one-line plot command in Gnuplot combines two curves in the same plot:

plot ’run1/sim.dat’ title ’nonlinear spring’ with lines, \ ’run2/sim.dat’ title ’linear spring’ with lines

The script in this exercise can be realized in two different ways. For example, you can stay within a copy of simviz1.py and run oscillator twice, with two different input files, and rename the data file sim.dat from the first run to another name (os.rename is an appropriate command for this purpose, cf. Chapter 3.4.4 on page 121). You can alternatively create a script on top of simviz1.py, that is, call simviz1.py twice, with different options, and then create a plot of the curves from the two runs. In this latter case you need to propagate the command-line arguments to the simviz1.py script.

2.4.3 Making Animations

Making Animated GIF Pictures. As an alternative to collecting all the plots from a series of experiments in a common document, as we did in the previous example, we can make an animation. For the present case, where we run through a sequence of m values, it means that m is a kind of time dimension. The resulting movie will show how the solution y(t) develops as m increases.

With the convert utility, which is a part of the ImageMagick package (see doc.html for links), we can easily create an animated GIF file from the collection of PNG plots15:

convert -delay 50 -loop 1000 -crop 0x0 \

plot1.png plot2.png plot3.png plot4.png ... movie.gif

One can view the resulting file movie.gif with the ImageMagick utilities display or animate:

display movie.gif

animate movie.gif

With display, you need to type return to move to the next frame in the animation. You can also display the movie in an HTML file by loading the animated GIF image as an ordinary image:

**<IMG SRC="movie.gif">**

When creating the animated GIF file in our script we need to be careful with the sequence of PNG plots. This implies that the script must make a list of all generated PNG files, in the correct order.

A more complicated problem is that the scale on the y axis in the plots must be fixed in the movie. Gnuplot automatically scales the axis to fit the maximum and minimum values of the current curve. Fixing the scale forces us to make modifications of simviz1.py. To distinguish the new from the old ver- sions, we call the new versions of the scripts simviz2.py and loop4simviz2.py. The reader should realize that the modifications we are going to make are small and very easily accomplished. This is a typical feature of scripting: just edit and run until you have an effective working environment.

The **simviz2.py** script has an additional command-line option -yaxis fol- lowed by two numbers, the minimum and maximum y values on the axis. The relevant new statements in simviz2.py are listed next.

# no specification of y axis in plots by default: ymin = None; ymax = None

...

elif option == ’-yaxis’:

ymin = float(sys.argv[1]); ymax = float(sys.argv[2]) del sys.argv[1]; del sys.argv[1]

...

# make gnuplot script:

...

if ymin is not None and ymax is not None:

f.write(’set yrange [%g:%g];\n’ % (ymin, ymax))

15 The -delay option controls the “speed” of the resulting movie. In this example -delay 50 means 50 · 0.1s = 0.5s between each frame.

The None value is frequently used in Python scripts to bring a variable into play, but indicate that its value is “undefined”. We can then use constructs like if ymin is None or if ymin is not None to test whether a variable is “undefined” or not.

The loop4simviz2.py script calls simviz2.py and produces the animated GIF file. A list pngfiles of PNG files can be built as we did with the PostScript files in loop4simviz1.py. Running convert to make an animated GIF image can then be accomplished as follows:

cmd = ’convert -delay 50 -loop 1000 -crop 0x0 %s tmp\_m.gif’\ % ’ ’.join(pngfiles)

failure = os.system(cmd)

**Making an MPEG Movie**. As an alternative to the animated GIF file, we can make a movie in the MPEG format. The script ps2mpeg.py (in src/tools) converts a set of uniformly sized PostScript files, listed on the command line, into an MPEG movie file named movie.mpeg. Inside our script we can write

failure = os.system(’ps2mpeg.py %s’ % ’ ’.join(psfiles))

We can easily create a link to the MPEG movie in the HTML file, e.g.,

html.write(’<H1><A HREF="movie.mpeg">MPEG Movie</A></H1>\n’)

2.4.4 Varying Any Parameter

Another useful feature of loop4simviz2.py is that we actually allow a loop over any of the real-valued input parameters to simviz1.py and simviz2.py, not just m! This is accomplished by specifying the option name (without the leading hyphen), the minimum value, the maximum value, and the increment as command-line arguments:

option\_name min max incr [ options as for simviz2.py ]

An example might be

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

This implies executing a set of experiments where the b parameter is varied. All the hardcoding of m as variable and part of filenames etc. in loop4simviz1.py must be parameterized using a variable holding the option name. This vari- able has the name option\_name and the associated numerical value is stored in value in the loop4simviz2.py script. For example, the value parameter runs from 0 to 2 in steps of 0.25 and option\_name equals b in the previous exam- ple on a specific loop4simviz2.py command. The complete loop4simviz2.py script appears next.

#!/usr/bin/env python

"""

As loop4simviz1.py, but here we call simviz2.py, make movies, and also allow any simviz2.py option to be varied in a loop. """

import sys, os

usage = ’Usage: %s parameter min max increment ’\

’[ simviz2.py options

try:

option\_name = sys.argv[1] min = float(sys.argv[2]) max = float(sys.argv[3]) incr = float(sys.argv[4])

except:

print usage; sys.exit(1)

]’ % sys.argv[0]

simviz2\_options = ’ ’.join(sys.argv[5:])

html = open(’tmp\_%s\_runs.html’ % option\_name, ’w’) html.write(’<HTML><BODY BGCOLOR="white">\n’) psfiles = [] # plot files in PostScript format pngfiles = [] # plot files in PNG format

value = min

while value <= max:

case = ’tmp\_%s\_%g’ % (option\_name, value)

cmd = ’python simviz2.py %s -%s %g -case %s’ % \

(simviz2\_options, option\_name, value, case) print ’running’, cmd

failure = os.system(cmd)

psfile = os.path.join(case,case+’.ps’)

pngfile = os.path.join(case,case+’.png’) html.write(’<H1>%s=%g</H1> <IMG SRC="%s">\n’ \

% (option\_name, value, pngfile)) psfiles.append(psfile)

pngfiles.append(pngfile)

value += incr

cmd = ’convert -delay 50 -loop 1000 %s tmp\_%s.gif’ \

% (’ ’.join(pngfiles), option\_name)

print ’converting PNG files to animated GIF:\n’, cmd failure = os.system(cmd)

html.write(’<H1>Movie</H1> <IMG SRC="tmp\_%s.gif">\n’ % \

option\_name)

cmd = ’ps2mpeg.py %s’ % ’ ’.join(psfiles)

print ’converting PostScript files to an MPEG movie:\n’, cmd failure = os.system(cmd)

os.rename(’movie.mpeg’, ’tmp\_%s.mpeg’ % option\_name) html.write(’<H1><A HREF="tmp\_%s.mpeg">MPEG Movie</A></H1>\n’ \

% option\_name) html.write(’</BODY></HTML>\n’)

html.close()

cmd = ’epsmerge -o tmp\_%s\_runs.ps -x 2 -y 3 -par %s’ \

% (option\_name, ’ ’.join(psfiles)) print cmd

failure = os.system(cmd)

failure = os.system(’ps2pdf tmp\_%s\_runs.ps’ % option\_name)

Note that all file and directory names generated by this script start with tmp\_ so it becomes easy to clean up all files from a sequence of experiments (in Unix you can just write rm -rf tmp\_\*).

With this script we can perform many different types of numerical exper- iments. Some examples on command-line arguments to loop4simviz2.py are given below.

– study the impact of increasing the mass:

m 0.1 6.1 0.5 -yaxis -0.5 0.5 -noscreenplot

– study the impact of increasing the damping:

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

– study the impact of increasing a nonlinear spring force:

c 5 30 2 -yaxis -0.7 0.7 -b 0.5 -func siny -noscreenplot

For example, in the experiment involving the spring parameter c you get the following files, which can help you in understanding how this parameter affects the y(t) solution:

tmp\_c.gif

tmp\_c.mpeg

tmp\_c\_runs.html

tmp\_c\_runs.ps

tmp\_c\_runs.pdf

# animated GIF movie

# MPEG movie

# browsable HTML document with plots and movies # printable PostScript document with plots

# PDF version of tmp\_c\_runs.ps

The reader is strongly encouraged to run, e.g., one of the three suggested experiments just shown and look at the generated HTML and PostScript files as this will illustrate the details explained in the text. Do not forget to clean up all the tmp\* files after having played around with the loop4simviz2.py script.

A more general and advanced tool for running series of numerical exper- iments, where several parameters may have multiple values, is presented in Chapter 12.1.

**Other Applications**. From the example with the oscillator simulations in this section you should have some ideas of how scripting makes it easy to run, archive, and browse series of numerical experiments in your application areas of interest. More complicated applications may involve large directory trees and many nested HTML files, all automatically generated by a steering script. Those who prefer reports in LATEX format can easily adapt our example on writing HTML files (see Exercise 2.12 for useful hints). With Numerical Python (Chapter 4) you can also conveniently load simulation results into the Python script for analysis and further processing.

You may well stop reading at this point and start exploring Python script- ing in your own projects. Since the book is thick, there is much more to learn and take advantage of in computational science projects, but the philosophy of the simviz1.py and loop4simviz2.py examples has the potential of making a significant impact on how you conduct your investigations with a computer.

2.5 File Format Conversion

The next application is related to the file writing and reading example in Chapter 2.2. The aim now is to read a data file with several time series stored column-wise and write the time series to individual files. Through this project we shall learn more about list and file processing and meet a useful data structure called dictionary (referred to as hash, HashMap, or associative array in other languages). We shall also collect parts of the script in reusable functions, which can be called when the script file is imported as a module in other scripts.

Here is an example of the format of the input file with several time series:

some comment line

1.5

measurements model1 model2

* 0.0 0.1 1.0
* 0.1 0.1 0.188
* 0.2 0.2 0.25

The first line is a comment line. The second line contains the time lag Δt in the forthcoming data. Names of the time series appear in the third line, and thereafter the time series are listed in columns. We can denote the i-th time series by yi(kΔt), where k is a counter in time, k = 0,1,2,...,m. The script is supposed to store the i-th time series in a file with the same name as the i-th word in the headings in the third line, appended with a extension .dat. That file contains two columns, one with the time points kΔt and the other with the yi(kΔt) values, k = 0, 1, . . . , m. For example, when the script acts on the file listed above, three new files measurements.dat, model1.dat, and model2.dat are created. The file model1.dat contains the data

0 0.1

1.5 0.1

3 0.2

Most plotting programs can read and visualize time series stored in this simple two-column format.

**2.5.1 A Simple Read/Write Script**

The program flow of the script is listed below.

1. Open the input file, whose name is given as the first command-line argu- ment. Provide a usage message if the command-line argument is missing.

2. Read and skip the first (comment) line in the input file.

3. Extract Δt from the second line.

1. Read the names of the output files by splitting the third line into words. Make a list of file objects for the different files.
2. Read the rest of the file, line by line, split the lines into the yi values and write each value to the corresponding file together with the current time value.

The resulting script can be built of constructions met earlier in this book. The reader is encouraged to examine the script code as a kind of summary of the material so far.

#!/usr/bin/env python

import sys, math, string

usage = ’Usage: %s infile’ % sys.argv[0]

try:

infilename = sys.argv[1]

except:

print usage; sys.exit(1)

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

# read first comment line (no further use of it here): line = ifile.readline()

# next line contains the increment in t values: dt = float(ifile.readline())

# next line contains the name of the curves: ynames = ifile.readline().split()

# list of output files: outfiles = []

for name in ynames:

outfiles.append(open(name + ’.dat’, ’w’))

t = 0.0 # t value

# read the rest of the file line by line: for line in ifile:

yvalues = line.split()

if len(yvalues) == 0: continue # skip blank lines for i in range(len(outfiles)):

outfiles[i].write(’%12g %12.5e\n’ % \

(t, float(yvalues[i])))

t += dt

for file in outfiles: file.close()

The source is found in src/py/intro/convert1.py. You can test it with the input file .convert\_infile1 located in the same directory as the script.

**2.5.2 Storing Data in Dictionaries and Lists**

We shall make a slightly different version of the script in order to demonstrate some other widely used programming techniques and data structures. First we load all the lines of the input file into a list of lines:

f = open(infilename, ’r’); lines = f.readlines(); f.close()

The Δt value is found from lines[1] (the second line). The yi(kΔt) values are now to be stored in a data structure y with two indices: one is the name of the time series, as found from the third line in the input file, and the other is the k counter. The Python syntax for looking up the 3rd value in a time series having the name model1 reads y[’model1’][2]. Technically, y is a dictionary of lists of floats. One can think of a dictionary as a list indexed by a string. The index is called a key. Each entry in our dictionary y is a list of floating-point values. The following code segment reads the names of the time series curves and initializes the data structure y:

# the third line contains the name of the time series: ynames = lines[2].split()

# store y data in a dictionary of lists of floats: y = {} # declare empty dictionary

for name in ynames:

y[name] = [] # empty list (of y values of a time series)

# load data from the rest of the lines: for line in lines[3:]:

yvalues = [float(x) for x in line.split()]

if len(yvalues) == 0: continue # skip blank lines i = 0 # counter for yvalues

for name in ynames:

y[name].append(yvalues[i]); i += 1

The syntax lines[3:] means the sublist of lines starting with index 3 and continuing to the end, making it very convenient to iterate over a part of a list. The statement

yvalues = [float(x) for x in line.split()]

splits line into words, i.e. list of strings, and then converts this list to a list of floating-point numbers by applying the function float to each word. More information about this compact element-by-element manipulation of lists appears on page 88. The continue statement, here executed if the line is blank (i.e., the yvalues list is empty), drops the rest of the loop and continues with the next iteration.

The final loop above needs a counter i for indexing yvalues. A nicer syntax is

for name, yvalue in zip(ynames, yvalues): y[name].append(yvalue)

The zip construction allows iterating over multiple lists simultaneously with- out using explicit integer indices (see also page 87).

At the end of the script we write the t and y values to file:

for name in y: # run through all keys in y ofile = open(name+’.dat’, ’w’)

for k in range(len(y[name])):

ofile.write(’%12g %12.5e\n’ % (k\*dt, y[name][k])) ofile.close()

We remark that we have no control of the order of the keys when we iter- ate through them in the first for loop. This modified version of convert1.py is called convert2.py and found in the directory src/py/intro. A more ef- ficient version, utilizing NumPy arrays, is suggested in Exercise 4.10. More information on dictionary operations is listed in Chapter 3.2.5.

**2.5.3 Making a Module with Functions**

The previous script, convert2.py, reads a file, stores the data in the file in a convenient data structure, and then dumps these data to a set of files. It could be convenient to increase the flexibility such that we can read the file into data structures, then optionally compute with these data structures, and finally dump the data structures to new files. Such flexibility requires us to do two things. First, we need to structure the script code in two functions performing the principal actions: loading data and dumping data. Second, we need to enable these functions to be called from another script. In this other script, we must import the functions from a module.

Collecting Statements in Functions. The statements in convert2.py associ- ated with loading the file data into a dictionary of lists can be collected in a function load\_data. We let the name of the file to read be an argument to the function, and at the end we return the y dictionary of lists, plus the time increment dt, to the calling code:

def load\_data(filename):

f = open(filename, ’r’); lines = f.readlines(); f.close() dt = float(lines[1])

ynames = lines[2].split()

y = {}

for name in ynames: # make y a dictionary of (empty) lists

y[name] = []

for line in lines[3:]:

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

if len(yvalues) == 0: continue # skip blank lines for name, value in zip(ynames, yvalues):

y[name].append(value)

return y, dt

The **load\_data** function returns two variables. This might look strange for programmers coming from Fortran, C/C++, and Java. In those languages multiple output variables from functions are transferred via function argu- ments, while in Python all output variables are (usually) returned as shown above. The calling code will typically assign the result of the function call to two variables:

y, dt = load\_data(filename)

Chapter 3.3 contains more information on Python functions and how to han- dle input and output arguments.

The function for dumping the dictionary of lists to files simply contains the last for loop in convert2.py:

def dump\_data(y, dt):

# write out 2-column files with t and y[name] for each name: for name in y.keys():

ofile = open(name+’.dat’, ’w’) for k in range(len(y[name])):

ofile.write(’%12g %12.5e\n’ % (k\*dt, y[name][k])) ofile.close()

**Making a Module**. To use these functions in other scripts, we should make a module containing the two functions. This is easy: we just put the two functions in a file, say convert3.py. We can then use this module convert3 as follows in another script:

import convert3

y, timestep = convert3.load\_data(’.convert\_infile1’) convert3.dump\_data(y, timestep)

Having split the load and dump phases, we may add operations on the y data in this script. For small computations we may well iterate over the list, but for more heavy computations with large amounts of data, we should convert each list in y to a NumPy array and use NumPy functions for efficient computations (see Chapters 2.2.5 and 4).

Instead of writing a script that applies the convert3 module, we may use the module in an interactive Python shell, such as IPython or the IDLE shell (see Chapter 2.2.6). Typically, we would call load\_data as an interactive statement and then interactively inspect y and compute with its entries.

**Extending the Module with a Script**. We showed above how to write a short script for calling up the main functionality in the convert3 module. This script is just an alternative implementation of the convert2.py script. However, the application script is tightly connected to the convert3 module, and Python therefore offers the possibility to let a file act as either a module or a script: if it is imported it is a module, and if the file is executed it is a script. The convention is to add the application script in an if block at the end of the module file:

f \_\_name\_\_ == ’\_\_main\_\_’:

usage = ’Usage: %s infile’ % sys.argv[0] import sys

try:

infilename = sys.argv[1] except:

print usage; sys.exit(1) y, dt = load\_data(infilename) dump\_data(y, dt)

The \_\_name\_\_ variable is always present in a Python program or module. If the file is executed as a script, \_\_name\_\_ has the value ’\_\_main\_\_’. Otherwise, the file is imported as a module, and the if test evaluates to false. With this if block we both show how the module functions can be used and we provide a working script which performs the same steps as the “flat” script convert2.py. The reader is referred to Appendix B.1 for more information on building and using Python modules.

2.5.4 Exercises

**Exercise 2.16**. Combine two-column data files to a multi-column file.

Write a script inverseconvert1.py that performs the “inverse process” of convert1.py (or convert2.py). For example, if we first apply convert1.py to the specific test file .convert\_infile1 in src/py/intro, which looks like

some comment line

1.5

tmp-measurements tmp-model1 tmp-model2

* 0.0 0.1 1.0
* 0.1 0.1 0.188
* 0.2 0.2 0.25

we get three two-column files tmp-measurements.dat, tmp-model1.dat, and tmp-model2.dat. Running

python inverseconvert1.py outfile 1.5 \

tmp-measurements.dat tmp-model1.dat tmp-model2.dat

should in this case create a file outfile, almost identical to .convert\_infile1; only the first line should differ (inverseconvert1.py can write anything on the first line). For simplicity, we give the time step parameter explicitly as a command-line argument (it could also be found from the data in the files).

Hint: When parsing the command-line arguments, one needs to extract the name model1 from a filename model1.dat stored in a string (say) s. This can be done by s[:-4] (all characters in s except the last four ones). Chapter 3.4.5 describes some tools that allow for a more general solution to extracting the name of the time series from a filename. ⋄

**Exercise 2.17**. Read/write Excel data files in Python.

Spreadsheet programs, such as Microsoft Excel, can store their data in a file using the so-called CSV (Comma Separated Values) data format. The row in the spreadsheet is written as one line in the file with all column values separated by commas. Here is an example, found as src/misc/excel\_data.csv:

"E=10 Gpa, nu=0.3, eps=0.001",,,, "run 2",,,,

,,,,

,,,,

"x","model 1","model 2",,"measurements" ,,,,

0,1.1,1,,1.1

2,1.3,1.2,,1.3

3,1.7,1.5,,1.8

One could think of reading such comma-separated files in Python simply by applying line.split(’,’) constructions. Explain why that will fail in the present case. Fortunately, Python has a module csv that can be used to read and write files in the CSV format and hence enable data exchange with spreadsheet programs. The construction

import csv

f = open(filename, ’r’) reader = csv.reader(f) for row in reader:

gives access to each row in the spreadsheet as a list row, where the elements contain the data in the corresponding columns. Read the excel\_data.csv file and print out the row list to see how the data are represented in Python. Then extract the data in the columns in separate lists, subtract the model1 and measurements data to form a new list, say errors. We want to write a new file in the CSV format containing the x and errors data in the first two columns of a spreadsheet. The cvs module enables data writing by

f = open(filename, ’w’) writer = csv.writer(f) writer.writerows(rows)

where rows is a list of list such that rows[i][j] holds the data in row i and column j. Load the new CVS file into a spreadsheet program like Openoffice or Excel and examine the data. ⋄