Python\_Scripting\_Computational\_Science\_C01

p25

**Chapter 1 Introduction**

In this introductory chapter we first look at some arguments why scripting is a promising programming style for computational scientists and engineers and how scripting differs from more traditional programming in Fortran, C, C++, and Java. The chapter continues with a section on how to set up your software environment such that you are ready to get started with the introduction to Python scripting in Chapter 2. Eager readers who want to get started with Python scripting as quickly as possible can safely jump to Chapter 1.2 to set up their environment and get ready to dive into examples in Chapter 2.

**1.1 Scripting versus Traditional Programming**

The purpose of this section is to point out differences between scripting and traditional programming. These are two quite different programming styles, often with different goals and utilizing different types of programming languages. Traditional programming, also often referred to as system programming, refers to building (usually large, monolithic) applications (systems) using languages such as Fortranl , C, C++, or Java. In the context of this book, scripting means programming at a high and flexible abstraction level, utilizing languages like Perl, Python, Ruby, Scheme, or Tel. Very often the script integrates operation system actions, text processing and report writing, with functionality in monolithic systems. There is a continuous transition from scripting to traditional programming, but this section will be more focused on the features that distinguish these programming styles.

Hopefully, the present section motivates the reader to get started with scripting in Chapter 2. Much of what is written in this section may make more sense after you have experience with scripting, so you are encouraged to go back and read it again at a later stage to get a more thorough view of how scripting fits in with other programming techniques.

**1.1.1 Why Scripting is Useful in Computational Science**

***Scientists Are on the Move****.* During the last decade, the popularity of scientific computing environments such as Maple, Mathematica, Matlab, and S- Plus/R has increased considerably. Scientists and engineers simply feel more productive in such environments. One reason is the simple and clean syntax of the command languages in these environments. Another factor is the tight integration of simulation and visualization: in Maple, Matlab, S-Plus/R and similar environments you can quickly and conveniently visualize what you just have computed.

***Build Your Own Environment****.* One problem with the mentioned environments is that they do not work, at least not in an easy way, with other types of numerical software and visualization systems. Many of the environment- specific programming languages are also quite simple or primitive. At this point scripting in Python comes in. Python offers the clean and simple syntax of the popular scientific computing environments, the language is very powerful, and there are lots of tools for *gluing* your favorite simulation, visualization, and data analysis programs the way you want. Phrased differently, Python allows you to build your own Matlab-like scientific computing environment, tailored to your specific needs and based on your favorite high- performance Fortran, C, or C++ codes.

***Scientific Computing Is More Than Number Crunching****.* Many computational scientists work with their own numerical software development and realize that much of the work is not only writing computationally intensive number-crunching loops. Very often programming is about shuffling data in and out of different tools, converting one data format to another, extracting numerical data from a text, and administering numerical experiments involving a large number of data files and directories. Such tasks are much faster to accomplish in a language like Python than in Fortran, C, C++, or Java. Chapter 3 presents lots of examples in this context.

***Graphical User Interfaces****.* GUIs are becoming increasingly more important in scientific software, but (normally) computational scientists and engineers have neither the interest nor the time to read thick books about GUI programming. What you need is a quick "how-to" description of wrapping GUIs to your applications. The Tk-based GUI tools available through Python make it easy to wrap existing programs with a GUI. Chapter 6 provides an introduction.

***Demos****.* Scripting is particularly attractive for building demos related to teaching or project presentations. Such demos benefit greatly from a GUI, which offers input data specification, calls up a simulation code, and visualizes the results. The simple and intuitive syntax of Python encourages users to modify and extend demos on their own, even if they are newcomers to Python.

Some relevant demo examples can be found in Chapters 2.3, 6.2, 7.2, 11.4, and 12.3.

***Modern Interfaces to Old Simulation Codes****.* Many Fortran and C programmers want to take advantage of new programming paradigms and languages, but at the same time they want to reuse their old well-tested and efficient codes. Instead of migrating these codes to C++, recent Fortran versions, or Java, one can wrap the codes with a scripting interface. Calling Fortran, C, or C++ from Python is particularly easy, and the Python interfaces can take advantage of object-oriented design and simple coupling to GUIs, visualization, or other programs. Computing with your Fortran or C libraries from these interfaces can then be done either in short scripts or in a fully interactive manner through a Python shell. Roughly speaking, you can use Python interfaces to your existing libraries as a way of creating your own tailored problem solving environment. Chapter 5 explains how Python code can call Fortran, C, and C++.

***Unix Power on Windows****.* We also mention that many computational scientists are tied to and take great advantage of the Unix operating system. Moving to Microsoft Windows environments can for many be a frustrating process. Scripting languages are very much inspired by Unix, yet cross platform. Using scripts to create your working environment actually gives you to the power of Unix (and more!) also on Windows and Macintosh machines. In fact, a script-based working environment can give you the combined power of the Unix and Windows/Macintosh working styles. Many examples of operating system interaction through Python are given in Chapter 3.

***Python versus Matlab****.* Some readers may wonder why an environment such as Matlab or something similar (like Octave, Scilab, Rlab, Euler, Tela, Yorick) is not sufficient. Matlab is a *de facto* standard, which to some extent offers many of the important features mentioned in the previous paragraphs. Matlab and Python have indeed many things in common, including no declaration of variables, simple and convenient syntax, easy creation of GUIs, and gluing of simulation and visualization. Nevertheless, in my opinion Python has some clear advantageous over Matlab and similar environments:

* - the Python programming language is more powerful,  
     
  - the Python environment is completely open and made for integration with external tools,
* - a complete toolbox/module with lots of functions and classes can be contained in a single file (in contrast to a bunch of M-files),  
     
  - transferring functions as arguments to functions is simpler,
* - nested, heterogeneous data structures are simple to construct and use,
* - object-oriented programming is more convenient,
* - interfacing C, C++, and Fortran code is better supported and therefore simpler,

- scalar functions work with array arguments to a larger extent (without modifications of arithmetic operators),

- the source is free and runs on more platforms.

Having said this, we must add that Matlab has significantly more comprehensive numerical functionality than Python (linear algebra, ODE solvers, optimization, time series analysis, image analysis, etc.). The graphical capabilities of Matlab are also more convenient than those of Python, since Python graphics relies on external packages that must be installed separately. There is an interface pymat that aIlows Python programs to use Matlab as a computational and graphics engine (see Chapter 4.4.3). At the time of this writing, Python's support for numerical computing and visualization is rapidly growing, especiaIly through the SciPy project (see Chapter 4.4.2).

**1.1.2 Classification of Programming languages**

It is convenient to have a term for the languages used for traditional scientific programming and the languages used for scripting. We propose to use *type- safe languages* and *dynamically typed languages,* respectively. These terms distinguish the languages by the flexibility of the variables, i.e. , whether variables must be declared with a specific type or whether variables can hold data of any type. This is a clear and important distinction of the functionality of the two classes of programming languages.

Many other characteristics are candidates for classifying these languages. Some speak about compiled languages versus interpreted languages (Java complicates these matters, as it is type-safe, but have the nature of being both interpreted and compiled). Scripting languages and system programming languages are also very common terms [27], i.e., classifying languages by their typical associated programming style. Others refer to high-level and low-Ievel languages. High and low in this context implies no judgment of quality. High-Ievel languages are characterized by constructs and data types close to naturallanguage specifications of algorithms, whereas low-Ievellan- guages work with constructs and data types reflecting the hardware level. This distinction may weIl describe the difference between Perl and Python, as high-Ievellanguages, versus C and Fortran, as low-Ievellanguages. C++ and Java come somewhat in between. High-Ievellanguages are also often re- ferred to as very high-level languages, indicating the problem of choosing a common scale when measuring the level of languages.

Our focus is on programming style rather than on language. This book teaches *scripting* as a way of working and programming, using Python as the preferred computer language. A synonym for scripting could weIl be *high-level programming,* but the expression sometimes leaves a confusion about how to measure the level. Why I use the term scripting instead of just programming is explained in Chapter 1.1.16. Already now the reader may have in mind that I use the term scripting in a broader meaning than many others.

**1.1.3 Productive Pairs of Programming Languages**

***Unix and C****.* Unix evolved to be a very productive software development environment based on two programming tools of different nature: the classical system programming language C for CPU-critical tasks, often involving non- trivial data structures, and the Unix shell for gluing C programs to form new applications. With only a handful of basic C programs as building blocks, a user can solve a new problem by writing a tailored shell program combining existing tools in a simple way. For example, there is no basic Unix tool that enables browsing a sorted list of the disk usage in the directories of a user, but it is trivial to combine three C programs, du for summarizing disk usage, sort for sorting lines of text, and less for browsing text files, together with the pipe functionality of Unix shells, to build the desired tool as a one-line shell instruction:

**du -a $HOME I sort -rn I less**

**In** this way, we glue three programs that are in principle completely indepen- dent of each other. This is the power of Unix in a nutshell. Without the gluing capabilities of Unix shells, we would need to write a tailored C program, of a much larger complexity, to solve the present problem.

A Unix command interpreter, or *shell* as it is normally called, provides a language for gluing applications. There are many shells: Bourne shell (sh) and C shell (csh) are classical, whereas Bourne Again shell (bash), Korn shell (ksh) , and Z shell (zsh) are popular modern shells. A program written in a shell is often referred to as a *script.* Although the Unix shells have many useful high-level features that contribute to keep the size of scripts small, the shells are quite primitive programming languages, at least when viewed by modern programmers.

C is a low-Ievel language, often claimed to be designed for computers and not humans. However, low-Ievel system programming languages like C and Fortran 77 were introduced as alternatives to the much more low-Ievel assembly languages and have been successful for making computationally fast code, yet with a reasonable abstraction level. Fortran 77 and C give nearly complete control of memory usage and CPU-critical program segments, but the amount of details at a low code level is unfortunately huge. The need for programming tools that increase the human productivity led to a devel- opment of more powerful languages, both for classical system programming and for scripting.

***C++ and VisualBasic****.* Under the Windows family of operating systems, cfficient program development evolved as a combination of the type-safe lan- guage C++ for elassical system programming and the VisualBasic language for scripting. C++ is a richer (and much more complicated) language than C and supports working with high-level abstractions through concepts like *object-oriented* and *generic programming.* VisualBasic is also a richer lan- guage than Unix shells.

***Java****.* Especially for tasks related to Internet programming, Java is taking over as the preferred language for building large software systems. Many regard JavaScript as some kind of scripting companion in Web pages. PHP and Java are also a popular pair. However, Java is much of a self-contained language, and being simpler and safer to apply than C++, it has become very popular and widespread for elassical system programming. A promising scripting companion to Java is Jython, the Java implementation of Python.

***Modern Scripting Languanges****.* During the last decade several powerful dy- namically typed languages have emerged and developed to a mature state. Bash, Perl, Python (and Jython), Ruby, Scheme, and Tel are examples of general-purpose, modern, widespread languages that are popular for script- ing tasks. PHP is a related language, but more specialized towards making Web applications.

**1.1.4 Gluing Existing Applications**

Dynamically typed languages are often used for gluing stand-alone applica- tions (typically coded in a type-safe language) and offer for this purpose rich interfaces to operating system functionality, file handling, and text process- ing. A relevant example for computational scientists and engineers is gluing a simulation program, a visualization program, and perhaps a data analysis program, to form an easy-to-use tool for problem solving. Running a program, grabbing and modifying its output, and directing data to another program are central tasks when gluing applications, and these tasks are easier to ac- complish in a language like Python than in Fortran, C, C++, or Java. A script that glues existing components to form a new application often needs a *graphical user interface* (GUI), and adding a GUI is normally a simpler task in dynamically typed languages than in the type-safe languages.

There are basically two ways of gluing existing applications. The simplest approach is to launch stand-alone programs and let such programs commu- nicate through files. This is exemplified already in Chapter 2.3. The other more sophisticated way of gluing consists in letting the script call functions in the applications. This can be done through direct calls to the functions and using pointers to transfer data structures between the applications. AI- ternatively, one can use alayer of, e.g., CORBA or COM objects between the script and the applications. The latter approach is very flexible as the applications can easily run on different machines, but data structures need to be copied between the applications and the script. Passing large data structures by pointers in direct calls of functions in the applications therefore seems at- tractive for high-performance computing. The topic is treated in Chapters 9 and 10.

**1.1.5 Scripting Yields Shorter Code**

Powerful dynamically typed languages, such as Python, support numerous high-level constructs and data structures enabling you to write programs that are significantly shorter than programs with corresponding functionality coded in Fortran, C, C++, or Java. In other words, more work is done (on average) per statement. A simple example is reading an *apriori* unknown number of real numbers from a file, where several numbers may appear at one line and blank lines are permitted. This task is accomplished by two Python statements2 :

**F =open(filename, 'r'); n =F.read().split()**

Trying to do this in Fortran, C, C++, or Java requires at least a loop, and in some of the languages several statements needed for dealing with a variable number of reals per line.

As another example, think about reading a complex number expressed in a text format like (-3.1,4) . We can easily extract the real part -3.1 and the imaginary part 4 from the string (-3. 1 ,4) using a *regular expression,* also when optional whitespace is included in the text format. Regular expressions are particularly weIl supported by dynamically typed languages. The relevant Python statements read3

**m = re.search(r'\(\s\*([~,]+)\s\*,\s\*([~,]+)\s\*\)', ' (-3.1,4) ')**

**re, im = [float(x) for x in m.groups()]**

We can alternatively strip off the parenthesis and then split the string '-3.1,4' with respect to the comma character:

**m = , (-3.1,4) '.stripO[l:-l]**

**re, im = [float(x) for x in m.split(',')]**

This solution applies string operations and a convenient indexing syntax instead of regular expressions. Extracting the real and imaginary numbers in Fortran or C code requires many more instructions, doing string searching and manipulations at the character array level.

The special text of comma-separated numbers enclosed in parenthesis, like (-3.1,4), is a valid textual representation of a standard list (tuple) in

2 Do not try to understand the details of the statements. The size of the code is what matters at this point. The meaning of the statements will he evident from Chapter 2.

3 The code examples may look cryptic for a novice, hut the meaning of the sequence of strange characters (in the regular expressions) should he evident from reading just a few pages in Chapter 8.2.

Python. This allows us in fact to convert the text to a list variable and from there extract the list elements by a very simple code:

**re, im = eval('(-3.1, 4)')**

The ability to convert textual representation of lists (ineluding nested, het- erogeneous lists) to list variables is a very convenient feature of scripting. In Python you can have a variable q holding, e.g., a list of various data and say s=str(q) to convert q to astring sand q=eval(s) to convert the string back to a list variable again. This feature makes writing and reading non-trivial data structures trivial, which we demonstrate in Chapter 8.3.l.

Ousterhout's article [27J about scripting refers to several examples where the code-size ratio and the implementation-time ratio between type-safe lan- guages and the dynamically typed Tellanguage vary from 2 to 60, in favor of Tel. For example, the implementation of a database application in C++ took two months, while the reimplementation in Tel, with additional functional- ity, took only one day. A database library was implemented in C++ during aperiod of 2-3 months and reimplemented in Tel in about one week. The Tel implementation of an application for displaying oil well curves required two weeks of labor, while the reimplementation in C needed three months. Another application, involving a simulator with a graphical user interface, was first implemented in Tel, requiring 1600 lines of code and one week of labor. A corresponding Java version, with less functionality, required 3400 lines of code and 3-4 weeks of programming.

**1.1.6 Efficiency**

Scripts are first compiled to hardware-independent byte-code and then the byte-code is *interpreted.* Type-safe languages, with the exception of Java, are compiled in the sense that all code is nailed down to hardware-dependent machine instructions before the program is executed. The interpreted, high- level, flexible data structures used in scripts imply a speed penalty, especially when traversing data structures of some size [6J.

However, for a wide range of tasks, dynamically typed languages are ef- ficient enough on today's computers. A factor of 10 slower code might not be crucial when the statements in the scripts are executed in a few seconds or less, and this is very often the case. Another important aspect is that dynamically typed languages can sometimes give you optimal efficiency. The previously shown one-line Python code for splitting a file into numbers calls up highly optimized C code to perform the splitting. You need to be a very elever C programmer to beat the efficiency of Python in this example. The same operation in Perl runs even faster, and the underlying C code has been optimized by many people around the world over a decade so your chances of creating something more efficient are most probably zero. A consequence is that in the area of text processing, dynamically typed languages will often provide optimal efficiency both from a human and a computer point of view.

Another attractive feature of dynamically typed languages is that they were designed for migrating CPU-critical code segments to C, C++, or For- tran. This can often resolve bottlenecks, especially in numerical computing. If you can solve your problem using, for example, fixed-size, contiguous arrays and traverse these arrays in a C, C++, or Fortran code, and thereby uti- lize the compilers' sophisticated optimization techniques, the compiled code will run much faster than the similar script code. The speed-up we are talk- ing about here can easily be a factor of 100 (Chapters 9 and 10 presents examples).

**1.1.7 Type-Specification (Declaration) of Variables**

Type-safe languages require each variable to be explicitly declared with a specific type. The compiler makes use of this information to control that the right type of data is combined with the right type of algorithms. Some refer to statically typed and strongly typed languages. Static, being opposite of dynamic, means that a variable's type is fixed at compiled time. This distinguishes, e.g. , C from Python. Strong versus weak typing refers to if something of one type can be automatically used as another type, i.e., if implicit type conversion can take place. Variables in Perl may be weakly typed in the sense that

**$b ='1.2'; $c =5.1\*$b**

is valid: $b gets converted from astring to a float in the multiplication. The same operation in Python is not legal, astring cannot suddenly act as a float4 .

The advantage of type-safe languages is less bugs and safer programming, at a cost of decreased flexibility. In large projects with many programmers the static typing certainly helps managing complexity. Nevertheless, reuse of code is not always weIl supported by static typing since a piece of code only works with a particular type of data. Object-oriented and especially generic programming provide important tools to relax the rigidity of a statically typed environment.

In dynamically typed languages variables are not declared to be of any type, and there are no *apriori* restrictions on how variables and functions are combined. When you need a variable, simply assign it a value - there is no need to mention the type. This gives great flexibility, but also undesired side effects from typing errors. Fortunately, dynamically typed languages usually perform extensive run-time checks (at a cost of decreased efficiency, of course) for consistent use of variables and functions. At least experienced program- mers will not be annoyed by errors arising from the lack of static typing: they will easily recognize typos or type mismatches from the run-time messages. The benefits of no explicit typing is that a piece of code can be applied in many contexts. This reduces the amount of code and thereby the number of bugs.

Here is an example of a generic Python function for dumping a data structure with a leading text:

**def debug(leading\_text, variable):**

**if os.environ.get('MYDEBUG', '0') == '1':**

**print leading\_text, variable**

The function performs the print action only if the environment variable MYDEBUG is defined and has the value '1'. By adjusting MYDEBUG in the op- erating system environment one can turn on and off the output from debug in any script.

The main point here is that the debug function actually works with any built-in data structure. We may send integers, floating-point numbers, com- plex numbers, arrays, and nested heterogeneous lists of user-defined objects (provided these have defined how to print themselves). With three lines of code we have made a very convenient tool. Such quick and useful code devel- opment is typical for scripting.

In asense, templates in C++ mimics the nature of dynamically typed languages. The similar function in C++ reads

**template <class T>**

**void debug(std::ostream& 0,**

**const std::string& leading\_text,**

**const T& variable)**

**{**

**c h a r \* c = getenv("MYDEBUG");**

**bool defined = false;**

**if (c != NULL) { // if MYDEBUG is defined ...**

**if (std:: string(c) "1") { / / if MYDEBUG**

**is true ... 0« leading\_text«""« variable « std::endl;**

**defined = true; }**

**}**

**} }**

In Fortran, C, and Java one needs to make different versions of debug for different types of the variable variable.

Object-oriented programming is also used to parameterize types of vari- ables. In Java or C++ we could write the debug function to work with ref- erences variable of type A and call a (virtual) print function in A objects.

The **debug** function would then work with all instances variable of subclasses of A. This requires us to explicitly register a special type as subclass of A, which implies some work. The advantage is that we (and the compiler) have full control of what types that are allowed to be sent to debug. The Python debug function is much quicker to write and use, but we have no control of the type of variables that we try to print. For the present example this is irrelevant, but in large systems unintended transactions of objects may be critical. Static typing may then help, at the cost quite some extra work.

**1.1.8 Flexible Function Interfaces**

Problem solving environments such as Maple, Mathematica, Matlab, and S-Plus/R have simple-to-use command languages. One particular feature of these command languages, which enhances user friendliness, is the possibility of using *keyword* or *named* arguments in function calls. As an illustration, consider a typical plot session5

**f = calculate( ... ) # calculate something**

**plot(f)**

Whatever we calculate is stored in f , and plot accepts f variables of different types. In the simple plot(f) call, the function relies on default options for axis, labels, etc. More control is obtained by adding parameters in the plot call, e.g.,

**plot(f, label= ' elevation' , xrange=[O,10])**

Here we specify a label to mark the curve and the extent of the *x* axis. Arguments with a name, say label, and a value, say ,elevation', are called keyword or named arguments. The advantage of such arguments is three-fold: (i) the user can specify just a few arguments and rely on default values for the rest, (ii) the sequence of the arguments is arbitrary, and (iii) the keywords help to document and explain the call. The more experienced user will often need to fine tune a plot, and in that case a range of additional arguments can be specified, for instance something like

**plot(f, label=' elevation' , xrange=[O,10], title='V ariable bottom', linetype='dashed', linecolor='red', yrange=[-l,l])**

Python offers keyword arguments in functions, exact1y as explained here. The plot calls are in fact written with Python syntax (but the plot function itself is not a built-in Python feature: it is here supposed to be some user-defined function) .

An argument can be of different types inside the plot function. Con- sider, for example, the xrange parameter. One could offer the specification of this parameter in several ways: (i) as a list [xmin,xmax], (ii) as astring 'xmin:xmax', or (iii) as a single floating-point number xmax, assuming that the minimum value is zero. These three cases can easily be dealt with inside the plot function, because Python enables checking the type of xrange (the details are explained in Chapter 3.2.10).

Some functions, debug in Chapter 1.1.7 being an example, accept any type of argument, but Python issues run-time error messages when an operation is incompatible with the supplied type of argument. The plot function above accepts only a limited set of argument types and could convert different types to a uniform representation (floating-point numbers xmin and xmax) within the function.

The nature and functionality of Python give you a full-fledged, advanced programming language at disposal, with the clean and easy-to-use interface syntax that has obtained great popularity through environments like Maple and Matlab. The function programming interface offered by type-safe lan- guages is more comprehensive, less flexible, and less user friendly. Having said this, we should add that user friendliness has, of course, many aspects and depends on personal taste. Static typing and comprehensive syntax may provide a reliability that some people find more user friendly than the pro- gramming style we advocate in this text.

**1.1.9 Interactive Computing**

Many of the most popular computational environments, such as Maple, Mat- lab, and S-Plus/R, offer interactive computing. The user can type a com- mand and immediately see the effect of it. Previous commands can quickly be recalled and edited on the fly. Since mistakes are easily discovered and corrected, interactive environments are ideal for exploring the steps of a computational problem. When all details of the computations are clear, the commands can be collected in a file and run as a program.

Python offers an interactive shell, which provides the type of interactive environment just described. A very simple session could do some basic calculations:

**» > from math import \***

**»> w=l**

**»> sin(w\*2.5)\*cos(1+w\*3)**

**-0.39118749925811952**

The first line gives us access to functions like sin and cos. The next line defines a variable w, which is used in the computations in the proceeding line. User input follows after the »> prompt, while the result of a command is printed without any prompt.

A less trival session could involve integrals of the Bessel functions *Jn(x):*

**»> from scipy.special import jn**

**»> def myfunc(x):**

**return jn(n,x)**

**»> from scipy import integrate**

**»> n=2**

**» > integrate.quad(myfunc, 0, 10)**

**(0 . 98006581161901407, 9.1588489241801687e-14)**

**»> n=4**

**» > integrate.quad(myfunc, 0, 10) (0.86330705300864041, 1 . 0255758932352094e-13)**

Bessel functions, together with lots of other mathematical functions, can be imported from a library scipy. special. We define a function, here just *Jn(x),* import an integration module from scipy, and call a numerical integration routine6 . The result of the call are two numbers: the value of the integral and an estimation of the numerical error. These numbers are echoed in the interactive shell. We could alternatively store the return values in variables and use these in further calculations:

**» > v, e = integrate.quad(myfunc, 0, 10)**

**»> q =v\*exp(-0.02\*140)**

**»>q**

**3.05589193585e-05**

Since previous commands are reached by the up-arrow key, we can easily fetch and edit an n assignment and re-run the corresponding integral computation. There are Python modules for efficient array computing and for visualization so the interactive shell may act as an alternative to other interactive scientific computing environments.

**1.1.10 Creating Code at Run Time**

Since scripts are interpreted, new code can be generated while the script is running. This makes it possible to build tailored code, a function for in- stance, depending on input data in a script. A very simple example is a script that evaluates mathematical formulas provided as input to the script. For example, in a GUI we may write the text 'sin(1.2\*x) + x\*\*a' as a rep- resentation of the mathematical function *f(x)* = sin(1.2x) + *x*a . If x and a are assigned values, the Python script can grab the string and execute it as Python code and thereby evaluate the user-given mathematical expres- sion (see Chapters 6.1.10,8.6.10, and 11.2.1 for details). This run-time code generation provides a flexibility not offered by compiled, type-safe languages.

As another example, consider an input file to a program with the syntax

**a =1.2**

**no of iterations =100**

**solution strategy = 'implicit'**

**cl =0**

**c2 =0.1**

**A=4**

**c3 = StringFunction('A\*sin(x)')**

The following generic Python code segment reads the file information and creates Python variables a, no\_of\_iterations, solution\_strategy, cl, c2, A, and c3 with the values as given in the file (!):

**file = open('inputfile.dat', 'r')**

**for line in file:**

**variable, value = [word.strip() for word in line.split('=')]**

**# variable names cannot contain blanks; replace space by \_**

**variable = variable.replace(' " '\_')**

**pycode = variable + '=' + value**

**exec pycode**

Moreover, c3 is in fact a function c3(x) as specified in the file (see Chap- ters 8.6.10 or 12.2.1 to see what the StringFunction tool really is). The pre- sented code segment handles any such input file, regardless of the number of and name of the variables. This is a striking example on the usefulness and power of run-time code generation.

Our general tool for turning input file commands into variables in a code can be extended with support for physical units. With some more code (the details appear in Chapter 11.4.10) we could read a file with

**a=1.2km**

**c2 = 0.1 MPa**

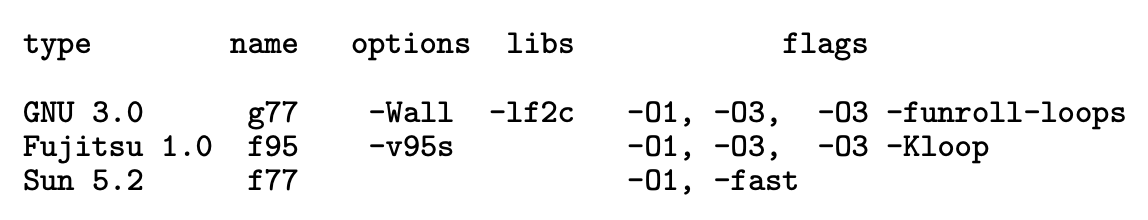
**A=4s**

Here, a may be converted from km to m, c2 may be converted from MPa to bar, and A may be kept in seconds. Such convenient handling of units cannot be exaggerated - most computational scientists and engineers know how much confusion that may arise from unit conversion.

1.1.11 Nested Heterogeneous Data Structures

Fortran, C, C++, and Java programmers will normally represent tabular data by plain arrays. In a language like Python, one can very often reach a hetter solution by tailoring some flexible built-in data structures to the problem at hand. As an example, suppose you want to automate a test of compilers for a particular program you have. The purpose of the test is to run through several types of compilers and various combinations of compiler flags to find the optimal combination of compiler and flags (and perhaps also hardware). This is a very useful (but boring) thing to do when heavy scientific computations lead to large CPU times.

We could set up the different compiler commands and associated flags by means of a table:



For each compiler, we have information about the vendor and the version (type), the name of the compiler program (name), some standard options and required libraries (options and libs), and a list of compiler Hag combinations (e.g., we want to test the GNU g77 compiler with the options -01, -03, and finally -03 -funroll-loops).

How would you store such information in a program? An array-oriented programmer could think of creating a two-dimensional array of strings, with seven columns and as many rows as we have compilers. Unfortunately, the missing entries in this array call for special treatments inside loops over com- pilers and options. Another inconvenience arises when adding more Hags for a compiler as this requires the dimensions of the array to be explicitly changed and also most likely some special coding in the loops.

In a language like Python, the compiler data would naturally be repre- sented by a dictionary, also called hash or associative array. These are ragged arrays indexed by strings instead of integers. In Python we would store the GNU compiler data as

compiler\_data['GNU'] ['type'] = 'GNU 3.0'

compiler\_data['GNU'] ['name'] = 'g77'

compiler\_data['GNU'] ['options'] = '-W all' compiler\_data['GNU']['libs'] = '-lf2c'

compiler\_data['GNU']['test'] = '-W all'

compiler\_data['GNU'] ['flags'] = ('-01','-03','-03 -funroll-loops')

Note that the entries are not of the same type: the ['GNU'] ['flags'] entry is a list of strings, whereas the other entries are plain strings. Such heteroge- neous data structures are trivially created and handled in dynamically typed languages since we do not need to specify the type of the entries in a data structure. The loop over compilers can be written as

**for compiler in compiler\_data:**

**c = compiler\_data[compiler] # ’GNU’, ’Sun’, etc.**

**cmd = ’ ’.join([c[’name’], c[’options’], c[’libs’]])**

**for flag in c[flags]:**

**oscmd = ’ ’.join([cmd, flag, ’ -o app ’, files])**

**os.system(oscmd)**

**<run program and measure CPU time>**

Adding a new compiler or new Hags is a matter of inserting the new data in the compiler\_data dictionary. The loop and the rest of the program remain the same. Another strength is the ease of inserting compiler\_data or parts of it into other data structures. We might, for example, want to run the compiler test on different machines. A dictionary test is here indexed by the machine name and holds a list of compiler data structures:

c = compiler\_data # abbreviation

test['ella.simula.no'] test['tva.ifi.uio.no'] test['pico.uio.no']

(c['GNU'], c['Fujitsu'])

(c['GNU'], c['Sun'], c['Portland']) (c['GNU'], c['HP'], c['Fujitsu'])

The Python program can run through the test array, log on to each machine, run the loop over different compilers and the loop over the flags, compile the application, run it, and measure the CPU time.

A real compiler investigation of the type outlined here is found in the src/app/wavesim2D/F77 directory of the software associated with the book.

**1.1.12 GUI Programming**

Modern applications are often equipped with graphical user interfaces. GUI programming in C is extremely tedious and error-prone. Some libraries pro- viding higher-Ievel GUI abstractions are available in C++ and Java, but the amount of programming is still more than what is needed in dynamically typed languages like Perl, Python, Ruby, and Tel. Many dynamically typed languages have bindings to the Tk library for GUI programming. An example from [27] will illustrate why Tk-based GUIs are easy and fast to code.

Consider a button with the text "Hello!", written in a 16-point Times font. When the user clicks the button, a message "hello" is written on standard output. The Python code for defining this button and its behavior can be written compactly as

**def out(): print 'hello' # the button calls this function**

**Button(root, text="Hello! ", font="Times 16", command=out) .packO**

Thanks to keyword arguments, the properties of the button can be specified in any order, and only the properties we want to control are apparent: there are more than 20 properties left unspecified (at their default values) in this example. The equivalent code using Java requires 7 lines of code in two func- tions, while with Microsoft Foundation Classes (MFC) one needs 25 lines of code in three functions [27]. As an example, setting the font in MFC leads to severallines of code:

**CFont\* fontPtr = new CFont();**

**fontPtr->CreateFont(16, 0, 0,0,700, 0, 0, 0, ANSI\_CHARSET,**

**OUT\_DEFAULT\_PRECIS,CLIP\_DEFAULT\_PRECIS, DEFAULT\_QUALITY,**

**DEFAULT\_PITCH IFLDONTCARE, "Times New Roman"); buttonPtr->SetFont(fontPtr);**

Static typing in C++ and Java makes GUI codes more complicated than in dynamically typed languages. (Some readers may at this point argue that GUI programming is seldom required as one can apply a graphical interface for developing the GUI. However, creating GUIs that are portable across Windows, Unix, and Mac normally requires some hand programming, and reusable scripting components based on, for instance, Tk and its extensions are in this respect an effective solution.)

Many people turn to dynamically typed languages for creating GUI ap- plications. If you have lots of text-driven applications, a short script can glue the existing applications and wrap them with a tailored graphical user inter- face. The recipe is provided in Chapter 6.2. In fact, the nature of scripting encourages you to write independent applications with flexible text-based in- terfaces and provide a GUI on top when needed, rather than to write huge stand-alone applications wired with complicated GUIs. The latter type of programs are hard to combine efficiently with other programs.

Dynamic Web pages, where the user fills in information and gets feedback, constitute a special kind of GUI of great importance in the Internet age. When the data processing takes place on the Web server, the communication between the user and the running program involves lots of text processing. Languages like Perl, PHP, Python, and Ruby have therefore been particularly popular for creating such server-side programs, and these languages offer very user-friendly modules for rapid development of Web applications. In fact, the recent "explosive" interest in scripting languages is very much related to their popularity and effectiveness in creating Internet applications. This type of programs are referred to as CGI scripts, and CGI programming is treated in Chapter 7.

**1.1.13 Mixed language Programming**

Using different languages for different tasks in a software system is often a sound strategy. Dynamically typed languages are normally implemented in C and therefore have well-documented recipes for how to extend the language with new functions written in C. Python can also be easily integrated with *C++* and Fortran. A special version of Python, called Jython, implements basic functionality in Java instead of C, and Jython thus offers a seamless integration of Python and Java.

Type-safe languages can also be combined with each other. However, call- ing C from Java is a more complicated task than calling C from Python. The initial design of the languages were different: Python was meant to be ex- tended with new C and *C++* software, whereas Fortran, C, *C++,* and Java were designed to build large applications in one language. This differing phi- losophy makes dynamically typed languages simpler and more flexible for multi-language programming. In Chapter 5 we shall encounter two tools, F2PY and SWIG, which (almost) automatically make Fortran, C, and *C++* code callable from Python.

Multi-language programming is of particular interest to the computa- tional scientist or engineer who is concerned with numerical efficiency. Using Python as the administrator of computations and visualizations, one can create a user-friendly environment with interactivity and high-level syntax, where computationally slow Python code is migrated to Fortran or *CjC++.* An example may illustrate the importance of migrating numerical code to Fortran or *CjC++.* Suppose you work with a very long list of floating- point numbers. Doing a mathematical operation on each item in this list is normally a very slow operation. The Python segment

**# x is a list**

**for i in range(len(x»: # i=O.1.2•...•n-1 n=len(x) is large**

**x[i] = sin(x [i])**

runs 20 times faster if the operation is implemented in Fortran 77 or C (the length of x was 5 million in my test). Since such mathematical operations are common in scientific computing, a special numerical package, called Numer- ical Python, was developed. This package offers a contiguous array type and optimized array operations implemented in C. The above loop over x can be coded like this:

**x = sin(x)**

where x is a Numerical Python array. The statement sin(x) invokes a C function, basically performing x [i] =sin(x [i]) for all entries x [i]. Such a loop, operating on data in a plain C array, is easy to optimize for a compiler. There is some overhead of the statement x=sin(x) compared to a plain Fortran or C code, so the Numerical Python statement runs only 13 times faster than the equivalent plain Python loop.

You can easily write your own C, C++, or Fortran code for efficient computing with a Numerical Python array. The combination of Python and Fortran is particularly simple. To illustrate this, suppose we want to migrate the loop

**for i in range(l.len(u)-l.l): # n=1.2 •...• n-2 n=len(u)**

**u\_new[i] = u[i] + c\*(u[i-1] - 2\*u[i] + u[i+1])**

to Fortran. Here, u and u\_new are Numerical Python arrays and c is a given floating-point number. We write the Fortran routine as

**subroutine diffusion(c. u\_new. u. n) integer n. i**

**real\*8 u(O:n-1). u\_new(O:n-1). c**

**Cf2py intent(in. out) u\_new do i = 1. n-2**

**u\_new(i) = u(i) + c\*(u(i-1) - 2\*u(i) + u(i+1» end do**

**return end**

This routine is placed in a file diffusion.f. Using the tool F2PY, we can create a Python interface to the Fortran function by a single command:

**f2py -c -m f77comp diffusion.f**

The result is a compiled Python module, named f77comp, whose diffusion function can be called:

**from f77comp import diffusion**

**<create and init u and u\_new (Numerical Python arrays» c =0.7**

**for i in range(no\_of\_timesteps):**

**u\_new =diffusion(c, u\_new, u) # can omit the length n (!)**

F2PY makes an interface where the output argument u\_new in the diffusion function is returned, as this is the usual way of handling output arguments in Python.

With this example you should understand that Numerical Python arrays look like Python objects in Python and plain Fortran arrays in Fortran. (Doing this in C or C++ is a lot more complicated.)

**1.1.14 When to Choose a Dynamically Typed language**

Having looked at different features of type-safe and dynamically typed lan- guages, we can formulate some guidelines for choosing the appropriate type of language in a given programming project. A positive answer to one of the following questions [27J indicates that a type-safe language might be a good choice .   
- Does the application implement complicated algorithms and data struc- tures where low-Ievel control of implementational details is important?  
   
- Does the application manipulate large datasets so that detailed control of the memory handling is critical?  
   
- Are the application's functions well-defined and changing slowly?

- Will static typing be an advantage, e.g., in large development teams?  
   
Dynamieally typed languages are most appropriate if one of the next char- acteristics are present in the project.  
   
- The application's main task is to connect together existing components. The application includes a graphieal user interface.  
 - The application performs extensive text manipulation.  
 - The design of the application code is expected to change significantly.  
   
- The CPU-time intensive parts of the application are located in small program segments, and if necessary, these can be migrated to C, C++, or Fortran.

- The application can be made short if it operates heavily on (possibly het- erogeneous, nested) list or dietionary structures with automatie memory administration.

- The application is supposed to communicate with Web servers.

- The application should run without modifications on Unix, Windows, and Macintosh computers, also when a GUr is included.

The last two features are supported by Java as well.

The optimal programming tool often turns out to be a combination of type-safe and dynamically typed languages. You need to know both classes of languages to determine the most efficient tool for a given subtask in a programming project.

**1.1.15 Why Python?**

Assuming that you have experience with programming in some type-safe lan- guage, this book aims at upgrading your knowledge about scripting, focusing on the Python language. Python has many attractive features that in my view makes it stand out from other dynamicaBy typed languages:

* Python is easy to learn because of the very clean syntax,
* - extensive built-in run-time checks help to detect bugs and decrease de- velopment time,

* programming with nested, heterogeneous data structures is easy,

* object-oriented programming is convenient,

* there is support for efficient numerical computing, and

* the integration of Python with C, C++, Fortran, and Java is very weIl supported.

If you come from Fortran, C, C++, or Java, you will probably find the following features of scripting with Python particularly advantageous:

1. Since the type of variables and function arguments are not explicitly writ- ten, a code segment has a larger application area and a better potential for reuse.
2. There is no need to administer dynamic memory: just create variables when needed, and Python will destroy them automaticaBy.
3. Keyword arguments give increased call flexibility and help to document the code.
4. The ease of setting up and working with arbitrarily nested, heterogeneous lists and dictionaries often avoids the need to write your own classes to represent non-trivial data structures.
5. Any Python data structure can be dumped to the screen or to file with a single command, a highly convenient feature for debugging or saving data between executions.
6. Gur programming at a high level is easily accessible.
7. Python has many advanced features appreciated by C++ programmers: classes, single and multiple inheritance, templates7 , namespaces, and op- erator overloading.
8. Regular expressions and associated tools simplify reading and interpret- ing text considerably.
9. The clean Python syntax makes it possible to write code that can be read and understood by a large audience, even if they do not have much experience with Python.
10. The interactive Python shell makes it easy to test code segments before writing them into a source code. The shell can also be utilized for gaining a high level of interactivity in an application.
11. Although dynamically typed languages are often used for smaller codes, Python's module and package system makes it weIl suited for large-scale development projects.
12. Python is much more dynamic8 than compiled languages, meaning that you can, at run-time, generate code, add new variables to classes, etc.
13. Program development in Python is faster than in Fortran, C, C++, or Java, thus making Python weIl suited for rapid prototyping of new appli- cations. Also in dual programming (programming two independent ver- sions of an application, for debugging and verification purposes), rapid code generation in Python is an attractive feature.

Most of these points imply much shorter code and thereby faster develop- ment time. You will most likely adopt Python as the preferred programming language and turn to type-safe languages only when strictly needed.

Once you know Python, it is easy to pick up the basics of Perl. To encour- age and help the reader in doing so, there is a companion note [15] having the same organization and containing the same examples as the introduc- tory Python material in Chapters 2 and 3. The companion note also covers a similar introduction to scripting with TcljTk.

**1.1.16 Script or Program?**

The term script was originally used for a set of interactive operating sys- tem commands put in a file, that is, the script was a way of automating otherwise interactive sessions. Although this is still an important application when writing code in an advanced language like Python, such a language is often also used for much more complicated tasks. Are we then writing scripts or programs? The Perl FAQ9 has a question "Is it a Perl program or a Perl script?" .The bottom line of the answer, which applies equally weIl in a Python context, is that it does not matter what term we uselO .

**In** a scientific computing context I prefer to distinguish between scripts and programs. The programs we traditionally make in science and engineer- ing are often large and computationally intensive, involving complicated data structures. The implementation is normally in a low-Ievellanguage like For- tran 77 or C, with an associated demanding debugging and verification phase. Extending such programs is non-trivial and require experts. The programs in this book, on the other hand, have more an administering nature, they are written in a language supporting commands at a significantly higher level than in Fortran and C (also higher than C++ and Java), the programs are short and commonly under continuous development to optimize your work- ing environment. Using the term script distinguishes such programs from the common numerically intensive codes that are so dominating in science and engineering.

Many people use scripting as a synonym for gluing applications as one typically performs in Unix shell scripts, or for collecting some commands in a primitive, tailored command-Ianguage associated with a specific monolithic system. This flavor of "scripting" often points in the direction of very sim- plified programming that anyone can do. My meaning of scripting is much wider, and is a programming style recognized by

1. gluing stand-alone applications, operating system commands, and other scripts,

2. flexible use of variables and function arguments as enabled by dynamic typing,

3. flexible data structures (e.g.,nestedheterogeneouslistsjdictionaries),reg- ular expressions, and other features that make the code compact and "high level" .

1.**2 Preparations for Working with This Book**

This book makes lots of references to complete source codes for scripts de- scribed in the text. All such scripts are available in electronic form, packed in a single file, which can be downloaded from the author's web page

**http://www.simula.no/ hpl/scripting**

Unpacking the file should be done in some directory, say scripting under your home directory, unless others have already made the software available on your computer system.

Along with this book we also distribute a package called scitools, which contains a set of useful Python modules and scripts for scientific work. There are numerous references to scitools throughout the text so you should down- load the package from the address above.

The following Unix commands perform the necessary tasks of installing both the book examples and the scitools package in a subdirectory scripting under your home directory:

cd $HOME

mkdir scripting

cd scripting

firefox http://www.simula.no/ hpl/scripting

# download TCSE3-3rd-examples.tar.gz and scitools.tar.gz gunzip TCSE3-3rd-examples.tar.gz scitools.tar.gz

tar xvf TCSE3-3rd-examples.tar

rm TCSE3-3rd-examples.tar

tar xvf scitools.tar

rm scitools.tar

On Windows machines you can use WinZip to pack out the compressed tarfiles.

Packing out the tarfiles results in two subdirectories, src and scitools. The former tarfile also contains a file doc.html (at the same level as src). The doc.html file provides convenient access to lots of manuals, man pages, tutorials, etc. You are strongly recommended to add this file as a bookmark in your browser. There are lots of references to doc.html throughout this book. The bibliography at the end of the book contains quite few items – most of the references needed throughout the text have been collected in doc.html instead. The rapid change of links and steady appearance of new tools makes it difficult to maintain the references in a static book.

The reader must set an environment variable $scripting equal to the root of the directory tree containing the examples and documentation associated with the present book. For example, in a Bourne Again shell (Bash) start-up file, usually named .profile or .bashrc, you can write

**export scripting=$HOME/scripting**

and in C shell-like start-up files (.cshrc or .tcshrc) the magic line is

**setenv scripting $HOME/scripting**

Of course, this requires that the scripting directory, referred to in the pre- vious subsection, is placed in your home directory as indicated.

Mac OS X users can just follow the Unix instructions to have the Python tools running on a Mac. For some of the tools used in this book Mac users need to have X11 installed.

In Windows 2000/XP/Vista, environment variables are set interactively in a dialog. Right-click My Computer, then click Properties, choose the Advanced tab, and click Environment Variables. Click New to add a new environment variable with a name and a value, e.g., scripting as name and

**C:\Documents and Settings\hpl\My Documents\scripting**

as value. An alternative method is to define environment variables in the C:\autoexec.bat file if you have administrator privileges (note that this is the only method in Windows 95/98/ME). The syntax is set name=value on one line.

Note the following: All references in this text to source code for scripts are relative to the $scripting directory. As an example, if a specific script is said to be located in src/py/intro, it means that it is found in the directory

**$scripting/src/py/intro**

Two especially important environment variables are PATH and PYTHONPATH. The operating system searches in the directories contained in the PATH vari- able to find executable files. Similarly, Python searches modules to be im- ported in the directories contained in the PYTHONPATH variable. For running the examples in the present text without annoying technical problems, you should set PATH and PYTHONPATH as follows in your Bash start-up file:

**export PYTHONPATH=$scripting/src/tools:$scripting/scitools/lib PATH=$PATH:$scripting/src/tools:$scripting/scitools/bin**

C shell-like start-up files can make use of the following C shell code:

**setenv PYTHONPATH $scripting/src/tools:$scripting/scitools/lib**

**set path=( $path $scripting/src/tools $scripting/scitools/bin )**

As an alternative, you can go to the scitools directory and run setup.py to install tools from this book (see Appendix A.1.5).

In the examples on commands in set-up files elsewhere in the book we apply the Bash syntax. The same syntax can be used also for Korn shell (ksh) and Z shell (zsh) users. If you are a TC shell (tcsh) user, you therefore need to translate the Bash statements to the proper TC shell syntax. The parallel examples shown so far provide some basic information about the translation.

On Windows you can set PATH to

**%PATH%;%scripting%\src\tools;%scripting%\scitools\bin**

and PYTHONPATH to

**%scripting%\src\tools;%scripting%\scitools\lib**

The second path, after ;, is not necessary if you use setup.py to install scitools properly (see Appendix A.1.5).

On Unix systems with different types of hardware, compiled programs can conveniently be stored in directories whose names reflect the type of hardware the programs were compiled for. We suggest to introduce an environment variable MACHINE\_TYPE and set this to, e.g., the output of the uname command:

**export MACHINE\_TYPE=‘uname‘**

A directory $scripting/$MACHINE\_TYPE/bin for compiled programs must be made, and this directory must be added to the PATH variable:

**PATH=$PATH:$scripting/$MACHINE\_TYPE/bin**

If you employ the external software set-up suggested in Appendix A.1, the contents of the PATH and PYTHONPATH environment variables must be extended, see pages 678 and 682.

There are numerous utilities you need to successfully run the examples and work with the exercises in this book. Of course, you need Python and many of its modules. In addition, you need Tcl/Tk, Perl, ImageMagick, to mention some other software. Appendix A.1.9 describes test scripts in the src/tools directory that you can use to find missing utilities.

Right now you should try to run the command

**python $scripting/src/tools/test\_allutils.py**

on a Unix machine, or

**python "%scripting%\src\tools\test\_allutils.py"**

on a Windows machine. If these commands will not run, the scripting en- vironment variable is not properly defined (log out and in again and retry). When successfully run, test\_allutils.py will check if you have everything you need for this book on the computer.