2

Python primer

This chapter provides a compressed introduction to the programming language Python. Python, named after the famous British television show 'Monty Python's Flying Circus' from the early 1970s, is a high-level programming and scripting language. The standard implementation of Python (named cPython) is an open source programming language defined by the Python Software Foundation (PSF)[1]. Python was invented in 1991 and is by now a robust and generally applicable programming language with a large user and development community. Of relevance to the present book, Python has very strong capabilities in for example scientific computing, multi-processing, data storage and plotting. Python provides functionality by a huge selection of software packages in many diverse fields.

2.1 Introduction

Python belongs to the group of programming languages referred to as scripting languages. A scripting language is normally defined by being a high-level programming language where the code is interpreted by another program (a Python interpreter in this case) during runtime. In Python the interpreter often calls optimised compiled functions to obtain reasonably high performance – for example, most Python versions call highly optimised routines for matrix-matrix multiplications or other linear algebra functionality. A scripting language is by nature an 'interpreted' language meaning that each line of code is read and executed one-by-one. It does not follow the usual 'edit-compile-link-run' cycle of compiled languages but boils down to an 'edit-run' cycle. Scripting languages have the root in the earlier days of computer technology where they were used for job control on large mainframe computers. This was a time before our interactive sessions where we in close to real time interact with computers. But despite the links to earlier days of computing, the Python language is highly popular and ranks among the ten most popular programming languages in several aspects [37].[2] Originally the stringent semantics of compiled languages such as C and Fortran was intended

[1] See the homepage `http://www.python.org/psf/`, which indicates the main mission of the PSF as (quote): "The mission of the Python Software Foundation is to promote, protect, and advance the Python programming language, and to support and facilitate the growth of a diverse and international community of Python programmers".

[2] A report by King [37] in the October 2011 version of *IEEE Spectrum* showed that Python ranked among the most popular of the approximately 600 existing programming languages in areas such as the TIOBE index, most book titles and most discussed. The TIOBE index is sort of a popularity metric of programming languages. See `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` for more information. Python ranked 8 in September 2011 and the same position in February 2014. The details are perhaps less interesting but it does show that the Python community is very large.

to help users – i.e. we have to carefully follow the rules of the programming language to avoid errors [46]. For example this included that variables should be declared at compile time and the type should be specified.

Python is a dynamically typed language where a value is bound to a name, which can be changed dynamically [74]. This means that we can have one place in a namespace where e.g. `x=10` (type: integer) and another place in the same namespace we may have `x='Hello'` (type: string). The Python interpreter at all times keeps track of the types of the variables. Python is also strongly typed in the sense that we are not allowed to apply operations that are incompatible with the types of variables involved [74]. Errors of this kind are caught at runtime where they raise an error. We should note that the 'incompatible types' error for a given operation does not necessarily mean that an operation such as `x+y` raises an error if `x` and `y` are of different type. For example if `x` is an integer and `y` is a complex floating point number then `x+y` is computed without problems. The `+` operator knows what to do if one input is of type integer and the other of type complex floating point. On the other hand, we are not allowed to do `10 + 'x'`, which raises a `TypeError` as the operator `+` is not supported for the variables/objects of type integer (`10`) and string (`'x'`). The Python interpreter keeps track of the types of all assigned variables and ensures that only allowed operations of the variables are performed. The important lessons to learn from Python are: 1) the variable type is set when a value is assigned to the variable; 2) a variable may change type in the same namespace; and 3) applying non-allowed operations of variables of given types are caught at runtime.[3]

Python is a general purpose high-level programming language invented by Guido Van Rossum in 1991 [15, 59]. It is by now a mature language which relies on modules[4] or packages[5] to provide functionality in many areas; scientific computing, file input/output, plotting, databases, multi-processing etc. In terms of functionality Python basically covers it all – it is also referred to as a 'batteries included' programming environment meaning that everything needed is also included or readily available [49, 68]. Python can be used both as a scripting and as a non-scripting language depending on context. It provides excellent interfacing to compiled modules in e.g. Fortran or C, and also packages to use

[3] The concepts of strongly and weakly typed programming languages are not well defined. To some programmers, Python is weakly typed because it does not enforce a given variable to hold values of a predefined type or to be compatible with an argument of a declared function. To others Python is strongly typed because it ensures that only allowed operations are performed on variables with a given type. A classical 1974 ACM paper by Liskov and Zilles [45] is very restrictive in the definition of strongly typed. By this definition Python is weakly typed but nowadays both the Python community (`python.org`) and recent references (e.g. [74]) sees Python as strongly typed. Due to the different definitions of strongly and weakly typed programming languages it is the advice to avoid using the terms to a greater extent.

[4] A Python module is a file with extension `.py`, which contains Python executable statements and/or callable objects. The latter includes Python functions which are objects that may pass one or more objects to the function, perform various manipulations on the objects and possibly return one or more objects by completion. A module may access objects in other modules. Each module has its own namespace – the namespace is the names of objects known to that module. This means that we can have two different modules each with a function of the same name. Due to the namespace idea of Python we can still easily handle which one we intend to use by selecting first the specific module and then the desired function inside that module. More information can be found on page 44.

[5] A Python package is a collection of Python modules, compiled extension files etc. which are tied together in a common namespace that makes all or parts of the package easily accessible. A bunch of modules in the same folder does not make a package – for those modules to form a package it is necessary to combine them with support files that ties the namespace together. More information can be found on page 44.

Graphics Processing Units (GPUs) are available (Open Computing Language (OpenCL)[6] and Compute Unified Device Architecture (CUDA)[7] based). Python is supported by numerous programmers from the entire globe, and its open nature means that help from others is never far away. Standard Python (referred to as cPython) is open source and is driven to a high degree by a highly competent community. Although the standard Python is open source, it is possible to get support, courses and consulting from privately held companies in the field[8]. Python is a cross-platform programming language with support for Microsoft Windows, UNIX/Linux, as well as Mac OSX.

Much valuable information can be found at 'Stack Overflow'.[9] A lot of Python questions and answers exist already and users can ask questions themselves. Before asking just make sure that you check if the question matches the requirements for the forum and that the answer is not immediately at hand via a simple Google search.

Python currently exists in two version types; a 2.x and a 3.x version. The 3.x version is not backwards compatible, which means that the choice between 2.x and 3.x is not necessarily obvious. Without doubt the 3.x version is the future of Python – the question is just when the future arrives. At the moment (and actually for quite some time), some important supporting functionality is not available for version 3.x. When this is combined with the fact that a fairly good 2.x to 3.x automatic code translator exists, the choice is most often to use 2.x. This is likely to change over time though as more and more packages are ported to version 3.x.

A vast amount of documentation exists for Python – both books and several Wikis. The quality of the documentation varies but in general a novice programmer comes a long way by one of the core Python books and then searching the Internet for specific challenges during development of new functionality. An enthusiastic community is always there to answer intelligent questions or solve non obvious challenges. Of books there are several of very high quality. For computational science in particular the books by Langtangen can be recommended [15, 41, 42]. Also, the comprehensive book by Hellmann [31] with examples from the Python standard library is valuable for the Python programmer. For novices the Python tutorial [72] is a good starting point. Further, the Python language reference [70], and the Python library reference [71] are essential. The Python tutorial, language reference and library reference are available for free at the homepage of the Python Software Foundation (PSF)[10]. Also, the essential reference by Beazley [5] is of high quality. Plotting (mainly two-dimensional) using the `matplotlib` package is covered in the book by Tosi [69]. 3D-plotting can be handled by the `Mayavi` package [64]. Optimisation is covered by e.g. Pyomo [30] but other tools exist in e.g. the standard Python `scipy` package.

New Python related requests or recommendations of various types exist as 'Python Enhancement Proposals' – many topics are covered here.[11] Regarding coding, a number of recommendations from the Python inventor exists in 'Style Guide for Python Code'.[12] This set of recommendations includes a lot of things from indentation, white space in expressions, version numbering etc. There are many rules/recommendations to follow and it may take

[6] See `http://www.khronos.org/opencl/`.
[7] See `http://www.nvidia.com/object/cuda_home_new.html`.
[8] See `http://wiki.python.org/moin/PythonConsulting`.
[9] See `http://stackoverflow.com`.
[10] The full documentation can be downloaded from `http://docs.python.org/download`.
[11] See `http://www.python.org/dev/peps/`.
[12] See `http://www.python.org/dev/peps/pep-0008/`.

Table 2.1: Short comparison of some MATLAB and Python characteristics.

MATLAB	Python
'lazy' pass-by-value	pass-by-reference
1 based indexing (Fortran)	0 based indexing (C)
column major	row major
# workers: ≤ 256	# workers: no limit
no namespace (path controlled)	namespace
no functions in scripts	functions in scripts
scripts, functions	scripts, modules, mixed
matrix oriented	array oriented

some time before a beginner has everything in place.

2.2 Python performance

A number of mostly older studies on the performance of Python and other programming languages have been made and is available on the Internet. Overall most studies found Python to be fairly slow in various aspects of computing [14]. In raw performance this analysis can hardly be said to be surprising. Scripting languages such as Matrix Laboratory by MathWorks, Inc. (MATLAB), AWK, Perl etc. are just slow when it comes to raw performance. MATLAB is the most relevant of these for scientific computing and it uses efficient libraries, a Just In Time (JIT) compiler and has been optimised through many years. Some characteristics of MATLAB versus Python are shown in Table 2.1. Even considering the strong points of MATLAB, Python is still being used for scientific computing and seems to be gaining in popularity – see e.g. [1, 18, 19, 48–50, 63, 68]. The question is then: why?

The reason is likely that scientific computing is a lot more than just number crunching. For large projects with thousands of lines of code, issues such as validation/testing, time-to-results, data storage, visualisation, maintenance, cross-platform support etc. play a large role when selecting a programming language. There are indications pointing towards programming in Python even for large software projects to rely on more programming languages – therefore, integration of different modules written in different programming languages is important.

William Scullin from Argonne National Laboratory gave a talk entitled "Python for High Performance Computing" at the *Python Conference* in Atlanta, USA in 2011 [66]. He said that Argonne supported three programming languages: C/C++, Fortran and Python. With Python they saw a shift from using 90% of the total time to develop code and 10% to run it to spending 10% on developing Python code and 90% of the total time to generate results. Obviously, the latter is the preferable situation in a laboratory where scientific results are the key outcome. In many applications where nested loops are unavoidable, Python can often not handle the 'innermost loop' in number crunching at sufficient speed. Python is, however, well suited to interface to modules compiled from e.g. C or Fortran. So code a reference and functioning system in plain Python, identify bottlenecks and handle these – if necessary. Do not use time to improve things that do not really need improving – this is a key lesson to learn in all scientific computing applications. As an example Scullin showed an application from

material science of 130 000 lines of code where more than 85 % of the code was written in Python. The main arguments put forward by Scullin as reasons for using Python was: 1) easy maintenance (they often change hardware); 2) 'batteries' included (everything included and virtually for free); 3) fast algorithm-to-results cycle (allowing more time on producing results than on development of code); and 4) active and helpful user community (large and dedicated global community means that help is always near). The 10–15 % of the code needing high performance was then coded in C or Fortran – Scullin mentioned that the team was moving more and more of the C code to Python. They moved as much as possible as long as it did not cost significantly in execution time. Thereby the code gets easier to maintain. Detailed information can be found in [21].

Other examples of Python used in science and engineering includes partial differential equations [48], (functional) magnetic resonance imaging [50], Geographic Information System (GIS) mapping etc. [68], and two-dimensional point dynamics (chaos) [4]. Also signal processing and visualisation [1], microscopy [40, 55], biology [73], quantum mechanical system simulation [51], agricultural system optimization [76], electromagnetic computations [47], probabilistic search [44], and in Very Large Scale Integration (VLSI) as a design capture component in digital electronics [11].

Handling the time critical part of an algorithm can be done in different ways in Python. The easiest and slowest is using plain cPython, and the fastest and most time-consuming is the use of C or Fortran. But there are also several possibilities in between these two limits. It is possible to use a JIT compiler, which is an easy way to improve performance in some cases. It is possible to use inline C embedded in the Python code, which can provide substantial speedup in many cases or using a tool called `cython` allows use of compiled C from the Python code. Even in the case where we need a full module to be implemented in C or Fortran there are excellent interfacing opportunities from the Python code.

So all in all the important lesson to learn is to implement the algorithm in standard plain cPython as the first step. If needed identify the critical parts and make these as fast as they need to be by using the different possibilities with a C or Fortran implementation as the most drastic. Besides this, always remember to make code, which is well tested, easy to maintain and is robust to strange user inputs. Finally, remember that the place where most can often be gained is by using an algorithm that can be implemented efficiently.

Example 2.1: A small test was made to compare MATLAB[13] and Python[14] with respect to execution time. Three tests were made: 1) Some matrix operations at element level for a 5000×5000 matrix size; 2) multiplication of four 2000×2000 real matrices; and 3) Eigenvalue Decomposition (EIG) of a 1000×1000 matrix. The results from computer Colfax (see page xxiii) were:

1. $\mathbf{A}^{\circ 2} - \mathbf{B}^{\circ 9} + 0.3 \cdot \mathbf{C}^{\circ 13} - 0.1 \cdot \mathbf{D}^{\circ 8}, \quad \mathbf{A}, \ldots, \mathbf{D} \in \mathbb{R}^{5000 \times 5000}$
   - MATLAB: 1.51 s.
   - Python: 7.18 s.
   - Python (JIT compiled via '`numexpr`'): 0.11 s.
2. $\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C} \cdot \mathbf{D}, \quad \mathbf{A}, \ldots, \mathbf{D} \in \mathbb{R}^{2000 \times 2000}$

[13] MATLAB version 8.3.0.532 (R2014a) running on Ubuntu 12.04.4 LTS kernel 3.5.0-47 (64 bit).
[14] Python version 2.7.8 from Anaconda 2.1.0 (x86_64) running on Ubuntu 12.04.4 LTS kernel 3.5.0-47 (64 bit).

- MATLAB: 0.75 s.
- Python: 4.22 s.
- Python (MKL): 0.75 s.

3. $\text{eig}(\mathbf{A}), \quad \mathbf{A} \in \mathbb{R}^{1000\times 1000}$

- MATLAB: 1.41 s.
- Python: 11.17 s.
- Python (MKL): 1.50 s.

where $\mathbf{A}^{\circ p}$ with $(\mathbf{A}^{\circ p})_{k,\ell} = (\mathbf{A})_{k,\ell}^{p} = A_{k,\ell}^{p}$ denotes the Hadamard power of the matrix $\mathbf{A}$ (element wise power). As this indicates it is almost impossible to make any solid conclusions. Some other tests were also made in the signal processing area and they seem to indicate that MATLAB is a bit faster. It is also essential how Python has been compiled and what libraries are used. The use of Math Kernel Library (MKL) seems to be superior compared to the default Automatically Tuned Linear Algebra Software (ATLAS) library provided for this system. Notice that the timings for Python with MKL and MATLAB is almost identical because MATLAB uses MKL if it is supported.

Code: **matlab_python**

2.3 Development environments and packages

When working with Python programming we need at least two core parts. The first is a development environment. This includes a Python interpreter, an editor to compose the code, and a command terminal which can run the interpreter together with the Python file. These parts may sometimes be integrated into one tool. On top of this we usually need to add functionality to the core Python interpreter. This is done via modules and packages. Modules are Python files including different kind of functions or functionality that can be used in our programs. Packages are basically collections of modules, which usually cover a wider scope. For example a package to assist when doing two-dimensional plotting or numerical array based computing.

2.3.1 Development environments

For a novice Python user everything may seem a bit confusing. There are several Python variants, several supporting Integrated Development Environments (IDEs), many packages of useful functionality in different areas and so on. The easiest way to start is to get a Python distribution which contains the most essential tools and packages. One such distribution named CANOPY[15] is made by the company Enthought. For academia, Enthought makes the package available free of charge provided the user registers. Another such distribution is Anaconda[16] by Continuum Analytics, which is also available free of charge. Some functionality and performance enhancement options are commercially available and free for academic users. Otherwise, Python binaries (and source) can be freely acquired where

[15] See `http://enthought.com`.
[16] See `http://www.continuum.io`.

several possibilities exist.[17] No matter if a standard version is installed or one with several packages included from start, it is quite simple to expand the installation with more packages and modules on a need-to-install basis. Installation of additional packages may depend on the base installation and refer to the documentation to find the correct way to expand your Python installation. If performance is very important the best option is likely to compile the packages for the specific hardware architecture. There may be significant difference in performance between various math libraries and it may be worth checking this in greater detail.

In terms of tools, an interactive `IPython` interpreter is a huge asset compared to a standard Python environment [58]. The `IPython` interpreter includes arrow-up/arrow-down memory, tab completion, interactive data visualisation, web-based notebook, interactive parallel computing etc. This makes `IPython` much easier to work with. Independent of operating system, the interactive environment is started from a terminal as:

```
$ ipython
IPython 1.1.0 -- An enhanced Interactive Python.
?         -> Introduction $and$ overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

`ipython` includes a number of 'magic' functions that e.g. provides a link to the operating system. A few examples of 'magic' functions:

- `%who`, `%whos`: Lists variables in the workspace as well as their type.
- `%run`: This corresponds to running the command '`$ python`' in a terminal window. This is an extremely useful command allowing the user to stay in `IPython` during script development.
- `%pwd`: Print working directory to the display.
- `%cd`: Change directory.
- `%timeit`: A very useful command to measure the time to run various Python commands, scripts or modules. Can for example be used like: `%timeit range(100000)`.
- `%dhist`: Directory history.

We can get more info on the 'magic' commands by executing '`%magic`' in the `IPython` environment. There are many 'magic' commands which can be used for many purposes. Many of these commands may save valuable time during development.

Another, and even more comprehensive development environment, is Python Integrated DeveLopment Environment (IDLE),[18] which is a complete but lightweight environment for the development of Python code. IDLE contains a multi-window text editor with Python colorising, automatic indentation, auto-completion etc. as well as a Python interpreter. IDLE is a cross platform tool developed entirely in Python.

[17] See `http://www.python.org/download`.

[18] For further information see the IDLE homepage at: `http://www.python.org/idle`.

For users with MATLAB experience the integrated development tool `spyder`[19] may be interesting. `spyder` includes the usual parts of an integrated development tool by having editor etc. but also integrated with the Python debugger allowing breakpoints, dynamic code introspection, direct warnings and errors etc. Also `pythonxy`[20] for Microsoft Windows operating systems, and the cross-platform `sage`[21] may be worth looking at. Both are based on many packages and provides lots of functionality.

Undoubtedly there are several ways to work with Python but during a software project a few things can be identified. Often there are many smaller issues that need to be examined such as finding and investigating different data structures, functionality of functions etc. Here it is quite common to have a tool such as `IPython` or similar running. To keep track of the desirable features along the way it is typical to have an editor open where the statements etc. are noted down for future reference. This means a minimum of: 1) a Python interpreter such as `IPython`; 2) a text editor such as emacs, vim, textmate etc.; and sometimes 3) a terminal window to run the small test scripts (or use the 'magic' function '`%run`'). The `IPython` notebook seems to gain in popularity as it integrates all these parts as well as for example integrated plotting and integrated text. The `IPython` notebook (started in a terminal window by: '`$ ipython notebook`') can be very useful during algorithm and code development due to its versatility and ease of use. The `IPython` notebook is an interactive web-based environment with a Python interpreter allowing a mixture of text, mathematical equations, plotting, Python code etc. in one document. The notebook uses a special file format, `.ipynb`, and the embedded Python code cannot directly be extracted from this. However, it is possible to save just the Python code-part from the notebook environment. The `IPython` notebook includes a fine and simple editor with tab completion, function help integrated, text highlighting, interactive plotting as well as documentation editing with e.g. titles, mathematical equations, which all encourage a logical code development structure.

Often, Python code development is separated into a number of segments of say 5–10 lines of code, each with a logical and distinct purpose. When using the `IPython` notebook it is convenient to develop the code in script-form like: 1) imports; 2) test signal definition; 3) some operations on the signals; 4) further operations to the output from step 3; 5) visualisation of signals; 6) final manipulation and formatting of signals. This typical approach to coding matches extremely well with the `IPython` notebook. The reason for this is that the `IPython` notebook environment uses a cell structure for the code (and text) meaning that the code can be passed to the Python interpreter on a cell-by-cell basis. This also means that it is very easy to step through small pieces of the code and inspect variables etc. to facilitate debugging. The use of cells means that a change in cell number n just demands re-running the code in cells $n, n+1, \ldots$. As the code in cells $1, \ldots, n-2, n-1$ has not been changed we do not need to re-run the code in these cells. This all makes the `IPython` environment easy and efficient to use for the typical code-run-debug cycle used in code development. It also means that once a piece of code has been debugged and validated, it can be pushed into to a function in a separate file and loaded into the `IPython` notebook as a module by an `import` statement and the next piece of code can be developed.

At the same time it is very easy to use the notebook to document the steps in the code development process where we can easily mix text, mathematical equations, plots etc. All

[19] See `http://code.google.com/p/spyderlib`.
[20] See `http://code.google.com/p/pythonxy`.
[21] See `http://www.sagemath.org` [26].

this serves a very useful part of the code development process – not least when numerous code developers are involved. The `IPython` notebook is also extremely useful for general dissemination purposes due to its combination of code, text and Python interpreter allowing for example a student to add missing pieces to an instructor developed code-base.

2.3.2 Packages

It is practically impossible to know all existing Python software packages and Table 2.3.2 only provides a very short list of some of the more relevant ones for scientific computing. In the autumn of 2011 there were more than 17 000 Python packages officially registered at `http://pypi.python.org/pypi`, and in January 2014 the number was above 39 000, which is truly impressive. Obviously not all are equally valuable or comprehensive but it still indicates that the Python community is very large and active.

Python, and hence also `IPython`, has a certain base functionality when started. This means that a working environment is available where memory management for instance is included. It also provides a core base of datatypes, control structures, user interaction etc. It is also so that for special functionality the user needs to import packages. For example it is necessary to import a package to get system access, a package to support numerical computation etc. This means that the base Python is fairly light and a user then picks whatever is needed for the given context. There are different 'variants' of Python – one example being `cython` [67], which includes static type declarations, subsequent cross-compiling to C, and finally a compilation to binary executable via a C-compiler. This can provide a substantial performance improvement, and `cython` is a popular tool when improvement of the performance of some Python code is needed.

2.4 Language features

Any programming language is obviously composed of a number of language specific constructs. This section briefly introduces some typical data types, control structures etc. as well as describing how function definitions are made and used.

2.4.1 Data types

Python is not much different from many other programming languages by having a number of different data types. Python is dynamically typed meaning that you do not need to declare variables to be of a specific type (as for example known from C) and a variable may change type in a program depending on the type of the value being assigned to the variable at different places in the code. Although you need not pre-specify the type of data for a certain variable this is of course not the same as saying that Python does not have data types – it does. But if you declare '`d = 2`', Python says the data type of '`d`' is an integer, and if you use '`d = 2.0`', then `d` is a float (specifically a double). Data types include integer, float, string, boolean, complex etc. – nothing unexpected here. There are also a number of different specialised datasets such as date, time, sets, tuples, dictionary, list, array, queue etc. One particularly important dataset is the `ndarray` from the much used `numpy` package for numerical computing [65]. The `ndarray` is the key base datatype when doing `numpy` based computations. The following provides a few examples of datatypes in Python:

Table 2.2: A fraction of the available scientific computing relevant Python packages beyond the standard libraries.

Package	Functionality
`numpy`	General purpose numerical math library
`scipy`	Scientific computing
`matplotlib`	2D plotting
`hdf5`	File format library for scientific data
`numexpr`	Fast evaluation of array expressions
`pil`	Python imaging library
`cv2`	Open source Computer Vision (OpenCV) wrapper implementing version 2.x
`scikits.learn`	Machine learning in Python [57]
`scikits.<···>`	Various `scipy` extensions
`sciproc`	Scientific multi-dimensional data handling
`nagpy`	Python to NAG math library interface
`pycuda`	Access to NVIDIA GPUs via the CUDA framework
`cython`	C extensions from Python
`fwrap`	Fortran 77/90/95 wrapper to Python
`f2py`	Fortran to Python interface
`pybrain`	Neural network
`mayavi`	3D plotting
`pandas`	Data analysis library
`radon`	Static code analyser for complexity etc.
`pyflakes`	Program analyser
`mkl`	Intel MKL
`nose`	Extended unit testing
`gitbox`	A simple Dropbox alternative
`sympy`	Symbolic math
`django`	Web database
`mysqldb`	Interface to the MySQL database
`pyopencl`	Access to Central Processing Units (CPUs) and GPUs via the OpenCL standard

```
>>> x = [1, 2, 3, 4]
>>> type(x)
list

>>> x = (1, 2, 3, 4)
>>> type(x)
tuple

>>> x = 'abc cba'
>>> type(x)
str

>>> import numpy
>>> x = numpy.random.normal(size=(2,4))
>>> type(x)
numpy.ndarray
```

and a dictionary can be used like:

```
>>> x = {'apple': 1111, 'banana': 2222, 'orange': 3333}
>>> type(x)
dict
>>> x['apple']
1111
>>> x['lemon'] = 4444
>>> x
{'orange': 3333, 'lemon': 4444, 'apple': 1111, 'banana': 2222}
>>> del x['orange']
>>> x
{'lemon': 4444, 'apple': 1111, 'banana': 2222}
>>> x.keys()
['lemon', 'apple', 'banana']
>>> type(x.keys())
list
```

These are just a few examples and the reader is referred to the reference and library manuals for other data types [70, 71].

There are a few things we should be aware of. Observe the following:

```
>>> a = [1, 2, 3, 4]
>>> b = a      # a and b are the same
>>> b[0] = -4
>>> a
[-4, 2, 3, 4]
>>> b
[-4, 2, 3, 4]
>>> id(a)
4317101016
>>> id(b)
4317101016
```

So if we use '`b = a`' they are literally the *same* – changing one of them is the same as changing both since both *a* and *b* refers to the same object (notice the same id). We can make a copy or form '`b = 1 * a`' in which case we get two different variables which from the start hold the same value but does not continue to do so if different operations are applied to the two. We can use the command `id` to get the identification tag of a variable and check that they are not alike. It is also possible to make an identical variable `c` by using `c = a[:]`. But this is not a generally applicable method as it cannot be used for all datatypes – we can use the '`copy.copy`' functionality, which is available after importing as '`import copy`'.

Another thing is integer division, which can be a unpleasant surprise in Python 2.x. Observe:

```
>>> 3 / 1
```

```
3
>>> 3 / 2
1
>>> 3.0 / 2
1.5
>>> float(3)/2
1.5
```

This can provide some surprising results. In Python 3.x this has been changed to the usually expected behaviour such that `3/2 = 1.5` but not in Python 2.x. It is possible, though, to import a module to have the same division principles as from Python 3.x as:

```
>>> from __future__ import division
>>> 3/1
3.0
>>> 3/2
1.5
>>> 3.0/2
1.5
```

This is one way to ensure what is likely the expected behaviour of division.

2.4.2 Control flow and loops

As with most programming languages Python allows various types of flow depending of certain conditions – `if`, `while`, `for` etc. which are also known from many other languages – 'switch' is not defined in Python though. Let us take a simple example:

```
>>> xset = range(5)
>>> type(xset)
list
>>> xset
[0, 1, 2, 3, 4]
>>> t = -20
>>> if t < min(xset):
        response = 'Low'
    elif t in xset:
        response = 'Mid'
    else:
        response = 'High'
>>> response
'Low'
```

As seen above the `if` statements can be combined with `elif` (else if) to provide a 'switch'-like behaviour known from other programming languages. A `for`-loop can be made in different ways:

```
>>> for x in range(3):
       print(x),
0
1
2

>>> for x in [1, 3, 5, 7]:
        print(x+2),
3
5
7
9
```

Often we need to traverse several lists jointly or keep track of the index. This can be handled elegant via the built-in functions `zip` and `enumerate`:

```
>>> for x, y in zip([1, 3, 5, 7], [-1, -3, -5, -7]):
        print(x + y),
0
0
0
0

>>> for k, x in enumerate([1, 3, 5, 7]):
        print((k, x)),
(0, 1)
(1, 3)
(2, 5)
(3, 7)
```

2.4.3 Functions

A *function* is a collection of Python statements operating on a number of input objects and which may return none, one or more objects. The functions are defined by use of the `def` keyword. A function normally resides in a file to be reusable – but it can also be defined directly in the Python workspace. A Python function is defined as shown in Algorithm 2.1.

Algorithm 2.1: Simple structure of a Python function.

```
1:  def Fun(in):
2:      """
3:      Functionality
4:      Input:
5:      Output:
6:      Raises:
7:      Example:
8:
9:      """
10:     ⋮
11:     return <Result>
```

In lines 2–9 in Algorithm 2.1 a docstring describes the functionality, input/output, explain potential raise of errors and provides an example of its use. This is normally the first thing done when developing a new function in Python. It is very important to have the functionality and input/output interface of the function well defined and having this in a docstring also means that we can use this to automatically create documentation via e.g. the `sphinx` package. In this example, lines 10–11 contains the Python statements including the return statement of the result using the `result` keyword.

Example 2.2: As a small example we make a function to compute $f(x) = x^2 + \pi$. This can be done in a Python interactive session (via e.g. `IPython`) as:

```
>>> import numpy

>>> def sqrpluspi(x):
...     res = x**2 + numpy.pi
...     print('{:}**2 + pi = {:10.4f}'.format(str(x), res))
```

```
...     return res

>>> y = sqrpluspi(1)
1**2 + pi =     4.1416

>>> y = sqrpluspi(4)
4**2 + pi =    19.1416

>>> y = sqrpluspi(7)
7**2 + pi =    52.1416
```

Since the function `sqrpluspi` is defined in the Python workspace only it cannot be used outside this instance of the workspace.

Code: **sqrpluspi**

It is also possible to use a variable number of inputs as well as inputs with default values for which the reader is referred to the documentation. Often it is necessary to have several calls to a function with different input. This can be handled by use of the built-in `map` function:

```
>>> def a_sqr_plus_b(a, b):
...     return a**2 + b

>>> print([a_sqr_plus_b(0, 4), a_sqr_plus_b(1 , 4), a_sqr_plus_b(2 , 4)])
[4, 5, 8]

>>> print(map(a_sqr_plus_b, range(3), [4]*3))
[4, 5, 8]
```

The `map` function applies a function to every item of one or more iterable object(s), e.g. lists, and returns a list of the result. Note that multiplication of a list means replicating it such that $[4]*3$ produces the same list as $[4,4,4]$ This way of using a function is convenient but the conceptual idea can also be applied for multiprocessing (see Section 6.3 on page 182).

2.4.4 Scripts, modules and packages

Scripts and modules refer to files, which contain Python code – and a package is a collection of modules to provide functionality in a certain domain. Scripts are basically intended as stand-alone Python source code for specific local purposes, and modules are typically more general functionalities intended to be used by other Python programs. There are a number of similarities and differences of how these are used and their types of recommended applications, which are described in the following.

A Python *script* is a file containing executable Python statements, objects, definitions etc. When a file containing Python source code is passed to a Python interpreter we basically get each line executed sequentially – line-by-line. A script is normally a stand-alone file to make some specific computation but where the content is not intended to be used in other contexts or by other Python programs. A script is normally executed by a Python interpreter directly via e.g. `$ python script.py` when we have the Python code in the file `script.py`. It is actually possible to import a script into a Python session, which means that any functions defined are available after the import. However, most practically usable scripts has some input/output and more seriously often by itself demands computations that may be very demanding. When only the functions of a script is intended to be used by other Python programs this kind of behaviour is obviously an undesired side effect, which should be avoided. Therefore, an import of a script is normally not intended nor recommended. We

should see scripts as stand-alone Python code files, which have some specific purpose and may use other modules to accomplish this purpose.

Example 2.3: We consider a very simple Python script file named `pscript.py`, which contains two functions and prints some results to the display. The code is:

```
"""
Docstring to describe the overall functionality of the script.

"""

import numpy

def f(k, x):
    """Computes f(k, x) = exp(k*x+pi)"""
    return numpy.exp(k*x + numpy.pi)

def g(x):
    """Computes g(x) = sin(cos(x**2 + pi/19)"""
    return numpy.sin(numpy.cos(x**2 + numpy.pi/19))

# Initialise and execute functions
k, x = -0.83, 1.89
r1 = f(k, x)
r2 = g(x)
print('result 1: {:10.4f}\nresult 2: {:10.4f}'.format(r1, r2))
```

In this approach to code development it is convenient to write the script with functions already from the start, which makes it easier to integrate code parts in the software project later on. The script can be executed by running '`$ python pscript.py`' in a terminal window, which is a convenient way to document smaller parts of ongoing investigations and pre-coding studies.

When using the script in the file `pscript.py` as intended we execute it as:

```
$ python pscript.py
result 1:     4.8206
result 2:    -0.7364
```

In principle we can also import the file with the content above into a Python interactive session. This can be done as follows when using `IPython`:

```
>>> import pscript
result 1:     4.8206
result 2:    -0.7364

>>> pscript.f?
Type:       function
String Form:<function f at 0x1035b0230>
File:       .../ch02_python/pscript/pscript.py
Definition: pscript.f(k, x)
Docstring:  Computes f(k, x) = exp(k*x+pi)

>>> pscript.f(1, 1)
62.902924281639983
>>> pscript.g(1)
0.38428360129276429

>>> r1
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-35a853e2b2b1> in <module>()
----> 1 r1
```

```
NameError: name 'r1' is not defined

>>> pscript.r1
4.8205722910136144
```

As we see above, the `import pscript` statement also executes the content of the `pscript.py` file. The two functions `f` and `g` in `script.py` are accessible in the interactive session via calls to `pscript.f` and `pscript.g`, respectively. Also, the docstring is displayed via the `IPython` command `pscript.<function>?`. However, the execution of the statements in lines 17–20 causes computations and an output in this case, which is normally not acceptable if we just wish to use the functions in another Python script or module. For this reason we should not import a script. Note also that the local variables `r1`, `r2`, `k` and `x` are accessible via `pscript.r1`, `pscript.r2`, `pscript.k` and `pscript.x`.

Code: **pscript**

A Python *module* is a file, which contains executable Python definitions and statements just like a script. Statements (such as `import`) are only executed once, which is the first time the module is imported. The module is intended to be used by other modules or scripts via an import, which makes all function definitions available to the importing session. When imported, a module is often (but not necessarily) designed to be silent in the sense that it does not by itself cause any directly visible input/output. When a Python interpreter reads Python code it executes everything in a line-by-line fashion – no matter if the file is a script being run directly by an interpreter or if it being imported into a Python session.[22]one variable defined before executing any code is the `__name__` variable. This variable is set to `"__main__"` (string). If the same source code file is imported into a Python session via an import it gets the name of the file name (excluding the `.py` extension) as a string. This difference combined with a conditional statement in the Python source code allows us to use a module in two different ways depending on how it is used.

Example 2.4: To illustrate the two different ways we can handle a file containing Python source code let us create a simple file named `nametest.py` with the content:

```
1 print("__name__:         {0}".format(__name__))
2 print("type(__name__):  {0}".format(type(__name__)))
```

Thus we print the content of the `__name__` variable as well as its type. So if we input the file `nametest.py` directly to a Python interpreter we get:

```
$ python nametest.py
__name__:         __main__
type(__name__):  <type 'str'>
```

If we open an `IPython` interactive Python session and import the `nametest.py` file we get:

```
>>> import nametest
__name__:         nametest
type(__name__):  <type 'str'>
```

[22]For example, 1) import statements are executed, which load the corresponding modules; 2) functions are evaluated and a reference objects to the functions are made linking them to the function names, and 3) other statements such as e.g. `print`, `if` are executed.

As we see from this we can distinguish between the two different ways we used the Python source code at runtime. This allows us to run the module as a script if we wish and as a module only making functions, imports etc. available for other Python scripts or modules.

Code: **nametest**

It is possible, via a special conditional, to make a module behave in two different ways depending on how it is used. This behaviour is often used as follows. The main purpose of a module is typically to provide some functionality via one or more function definitions. These are then available for other Python scripts or modules via an import statement. The script-like behaviour possible for a module under a conditional based on the `__name__` variable is often used to provide an example or a test of the module, which can be valuable both during code development and to a user of the module. This does not replace unit testing (see page 67) though.

Example 2.5: We continue with the same example as used for the script in Example 2.3 but here we wish to map the code to a module. We 'hide' the input/output statements behind a conditional based on the `__name__` variable as described in Example 2.4, which ignores that part of the code when it is imported and runs the code when executed as a script:

```
"""
Docstring to describe the overall functionality of the module.

"""

import numpy

def f(k, x):
    """Computes f(k, x) = exp(k*x+pi)"""
    return numpy.exp(k*x + numpy.pi)

def g(x):
    """Computes g(x) = sin(cos(x**2 + pi/19)"""
    return numpy.sin(numpy.cos(x**2 + numpy.pi/19))

if __name__ == '__main__':
    # Initialise and execute function when module is running as a script
    k, x = -0.83, 1.89
    r1 = f(k, x)
    r2 = g(x)
    print('result 1: {:10.4f}\nresult 2: {:10.4f}'.format(r1, r2))
```

We save this code in the file `pmodule.py`. When executing the file as a script we get:

```
$ python pmodule.py
result 1:     4.8206
result 2:    -0.7364
```

When we import the file with the content above into a Python interactive session, via e.g. `IPython`, we get:

```
>>> import pmodule

>>> pmodule.f?
Type:       function
String Form:<function f at 0x10352e2a8>
File:       .../ch02_python/pmodule/pmodule.py
Definition: pmodule.f(k, x)
Docstring:  Computes f(k, x) = exp(k*x+pi)

>>> pmodule.g?
Type:       function
```

```
String Form:<function g at 0x10352e1b8>
File:       .../ch02_python/pmodule/pmodule.py
Definition: pmodule.g(x)
Docstring:  Computes g(x) = sin(cos(x**2 + pi/19)

>>> pmodule.f(1, 1)
62.902924281639983

>>> pmodule.g(2)
-0.49702173952941486
```

As we see above, the `import pmodule` statement is silent in this case. The two functions `f` and `g` in `script.py` are accessible in the interactive session via calls to `pscript.f` and `pscript.g`, respectively. We can also use the `IPython` command `pmodule.<function>?` to get information on the function and observe that the docstrings are displayed as expected.

Overall we have demonstrated that the Python module `pmodule.py` acts in one way when imported and in a script-like behaviour when run as a script via feeding the file to a Python interpreter.

Code: **pmodule**

A Python *package* is a collection of modules, which jointly provide some functionality in a certain domain of application. It might be a collection of different modules to perform numerical integration or modules to perform different types of pattern recognition. Once a package has been imported into a Python interactive session or in a Python script or module, the different modules in the package can be accessed via the functions available in the namespace.

The namespace is, as the name indicates, a space holding names. The namespace links a name to the corresponding object (e.g. variable or function). Typically, we can see the namespace as a collection of all the names known to the Python interpreter. This part is the global namespace. If we for example have a module named `mymodule` and it has an object (function) `func1` then we access the object by first importing `mymodule` as `import mymodule`. After this we may access the object as `mymodule.func1`. The module `mymodule` is global to the interpreter whereas the object `func1` is local to `mymodule`. This also means that we may have multiple identical local names without this causing problems. The global name is used to separate such objects. All names linked to objects can be listed like for example:

```
>>> import numpy
>>> numpy.__dict__
    {...
    numpy.uint16,
    numpy.int16,
    numpy.bool_,
    numpy.complex128,
    numpy.float64,
    numpy.uint32,
    numpy.int32,
    numpy.string_,
    numpy.complex256,
    ...}
```

This example just lists a few of the hundreds of available names in the `numpy` package.

2.4.5 Exceptions and error handling

Exceptions are errors that appear when executing a statement – even when the statement is syntactically correct. This could for example be the statement `numpy.zeros(10, 10)` if we

have forgotten to first perform '`import numpy`'. But there are many ways we can see exceptions:

```
>>> R = numpy.zeros((10,10))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/Users/tl/<ipython-input-1-6f15bb67d12b> in <module>()
----> 1 R = numpy.zeros((10,10))

NameError: name 'numpy' is not defined


>>> v = 'sss' + 10
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/tl/<ipython-input-2-8fa7044e551f> in <module>()
----> 1 v = 'sss' + 10

TypeError: cannot concatenate 'str' and 'int' objects


>>> 1/0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/Users/tl/<ipython-input-3-05c9758a9c21> in <module>()
----> 1 1/0

ZeroDivisionError: integer division or modulo by zero


>>> a = [1, 2, 3]

>>> a[3]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/Users/tl/<ipython-input-4-94e7916e7615> in <module>()
----> 1 a[3]

IndexError: list index out of range
```

As seen above we are told what kind of error we made and what caused the exception. A number of standard identifiers are included in Python, and it is also possible to make user-defined exceptions.

Fortunately, Python has some excellent possibilities for exception handling. Basically we can execute a statement, which goes wrong, catch the exception and request some kind of action. The following presents a simple example:

```
>>> try:
>>>     import nonsense
>>> except ImportError:
>>>     print("Nonsense can't be imported but let's ignore the exception")

Nonsense can't be imported but let's ignore the exception
```

The way this works is the following. First the statements between `try` and `except` are executed. If this works without raising an exception, the `except` part is just skipped. If an exception happens and it matches the `except` statement then the code following the `except` statement is executed. It is possible to catch more exceptions by grouping such as:

```
>>> try:
>>>     import numpy
>>> except (ImportError , ValueError , NameError):
>>>     print("numpy can't be imported")
>>> else:
```

```
>>>     print("try succeeded and no exceptions raised")

try succeeded and no exceptions raised
```

We can also raise an exception like:

```
>>> raise NameError('Nonsense')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/Users/tl/<ipython-input-20-04e232945c24> in <module>()
----> 1 raise NameError('Nonsense')

NameError: Nonsense
```

Further, we can use the `finally` keyword to construct a `try...finally` statement, where the 'finally' part is *always* executed before leaving the exception handling. An example for a small function to divide two numbers is:

```
>>> def divide(a, b):
>>>     try:
>>>         res = a / b
>>>     except (ZeroDivisionError, TypeError):
>>>         print("Divide zero error or TypeError!")
>>>     else:
>>>         print("Result is: {:10.4f}".format(res))
>>>     finally:
>>>         print("Always give 'finally' statement!")

>>> divide(3,1)
Result is:     3.0000
Always give 'finally' statement!

>>> divide(3,0)
Divide zero error or TypeError!
Always give 'finally' statement!

>>> divide(3,"0")
Divide zero error or TypeError!
Always give 'finally' statement!
```

This clearly explains the difference between the `else` which is used in case no exceptions are caught, and `finally` which is always executed.

Example 2.6: When implementing scientific computing algorithms in Python, errors are quite often of the kind where the input to the function is not correct. It may be an input of incorrect type, perhaps the values are not as required, or perhaps a matrix may be too large or small. A simple, easy to read and efficient way to handle this kind of problems is to raise errors when some formal condition on the input parameters is not met. By experience, it is very useful to start the coding by writing the docstring to describe the functionality, input/output, raises and example(s). Once that part is in place, it is good programming behaviour to handle the input – raise errors of the proper kind with useful information on the problem.

One simple example could be the Python code to compute $f(x) = x + \sqrt{x}$ for $x \in \mathbb{R}_+$, which can be done as:

```
1 def f(x):
2     """Computes x+sqrt(x).
3
4     Args:
5         x (positive float): input value to the function.
6     Returns:
7         y (positive float): x+sqrt(x).
```

```
    Raises:
        TypeError: if x is not a float.
        ValueError: if x is negative.
    Example:
        >>> f(10.0)
        f(10.0) = 13.1622776602

    """
    if not type(x) == float:
        raise TypeError("Input '{0}' must be a float.".format(x))
    if x < 0:
        raise ValueError("Input '{0}' must be zero or positive.".format(x))
    y = x + x**0.5
    return y
```

We save this Python code in the file `raise_error.py`. If we use the function we could for example get:

```
>>> import raise_error
>>> raise_error.f(10.0)
13.1622776602
>>> raise_error.f(10)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-$input$-3-88b91ae8aaca> $in$ <module>()
----> 1 raise_error.f(10)

.../ch02_python/raise_error/raise_error.py $in$ f(x)
     27     """
     28     if not type(x) == float:
---> 29         raise TypeError("Input '{}' must be a float.".format(x))
     30     if x < 0:
     31         raise ValueError("Input '{}' must be zero or positive.".format
          (x))

TypeError: Input '10' must be a float.
```

We can thus see the line causing the raised error and also in the last line see the type of error and what was actually wrong (being that '10' is not a float).

Code: **raise_error**

2.4.6 Assertions

Assertions are used to crash code early when a state occurs that should in principle not be possible and is implemented in Python via the keyword `assert`. Assertions should only be used for self-tests in the code and not to catch for instance bad or incorrect user input. The latter type of errors should be handled by raising exceptions, print error messages etc. An example of use of an assertion is:

```
>>> x = 1E-19
>>> assert abs(x) > 1E-10, "Required: abs(x) > 1E-10, x is {}".format(x)
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-11-4aca6f6e1a4e> in <module>()
----> 1 assert abs(x) > 1E-10, "Required: abs(x) > 1E-10, x is {}".format(x)

AssertionError: Required: abs(x) > 1E-10, x is 1e-19
```

An important reason only to use assertions for self-testing is that if Python is started with '`-O`' option, then assertions are ignored from the code. One could therefore say that assertions are used when debugging code. The Python organisation recommends to use assertions in the following cases:

- Checking of parameter types, classes, and variables.
- Checking data structure invariants.
- Checking of 'cannot happen' cases – conflicts where two cases says opposite or directly conflicting situations.
- Checking reasonability of return parameters from functions.

Assertions cannot replace unit testing – unit testing is a much more elaborate testing scheme across many input variables and validates the computed results. However, assertions are a supplement to catch some mistakes/errors at an often early stage of code development.

2.4.7 Documentation

Python has a simple yet powerful system for documentation — the philosophy is that if it is not simple, the system is most likely not used anyway. Python code can be "documented" via comments in the code (using '#' followed by the comment) or *docstrings*. Docstrings can, unlike comments, be access runtime and used by tools for generating documentation. Docstrings are text embraced in three beginning double quotes followed by some text and then concluded by three double quotes. An example could be:

```
def square_minus_1(n):
    """
    Function to compute n**2-1 where n is the input to the function.

    """
    return n**2-1
```

The content of the docstring usually contains a description of the function, a specification of input and output to the function as well as examples of its use. Via the standard Python tool `pydoc` it is then possible to automatically generate manual pages in UNIX man style or html pages of a complete Python package. The programmer just needs to make the docstrings and then the documentation can made automatically.

For larger projects where a Python package has been developed, the `Sphinx` tool for creating nice documentation is the de-facto standard in Python. `Sphinx` is a bit time-consuming to set up but once it has been done it is very easy to maintain a documentation of very high quality as the project develops. It is therefore the easiest to start using `Sphinx` from the beginning of a package development project which eases the work when new modules are included. `Sphinx` can automatically extract information from docstrings, insert source code if desired and it is also possible to manually insert more detailed information on algorithms or the mathematical background for a given module. LaTeX equations can e.g. be included which is a significant strength of this tool. `Sphinx` can for instance generate the documentation in HTML and LaTeX format just to provide two examples.

2.5 Scientific computing packages

A few packages are absolute necessities when using Python for scientific computing. These are: 1) `numpy`, which provides core data types (arrays) as well as core computational functions for scientific computing; 2) `scipy`, which builds on top of `numpy` providing e.g. linear algebra functionality, signal processing and optimisation; and 3) `matplotlib`, which is an excellent

package for visualization of 2D data [33, 34]. A number of other packages are also useful scientific computing packages are `sympy` for symbolic calculations and `mpmath` for multi-precision calculations. The packages `pickle`, `pytables`, `h5py` provides file and data access. These packages are covered in the following. A number of other packages are part of the Python standard library – the book by Hellman [31] covers these in detail.

2.5.1 *`numpy` – Numerical Python*

`numpy` (numerical Python – see `http://numpy.scipy.org/`) is a completely essential package when using Python for numerical computing purposes. For MATLAB users it may be beneficial to have a look at `http://www.scipy.org/NumPy_for_Matlab_Users` which describes some similarities and differences between MATLAB and Python. The core data type in `numpy` is a multi-dimensional array as described earlier. Many things in `numpy` are build around this specific datatype named an `ndarray`. This type is not the same as the `array` type in the standard Python library. A few examples are illustrated by the following:

```
>>> import numpy
>>> a = numpy.random.normal(size=(6, 1))
>>> a
array([[ 1.96613579],
       [-0.81300638],
       [-1.05736308],
       [-1.31714556],
       [-0.47589936],
       [ 0.8443035 ]])

>>> b = a.reshape((3, 2))
>>> b
array([[ 1.96613579, -0.81300638],
       [-1.05736308, -1.31714556],
       [-0.47589936,  0.8443035 ]])

>>> type(b)
numpy.ndarray

>>> b.dtype
dtype('float64')
```

As seen we first import the `numpy` package and afterwards a random normal distributed `numpy` array is formed. The `reshape` function allows us to change the way the array is organised. Performing indexing and slicing in a `numpy` array can be done like the following where we continue using the `b`-array from above:

```
>>> b[0:2, :]
array([[ 1.96613579, -0.81300638],
       [-1.05736308, -1.31714556]])

>>> b[:, 1]
array([-0.81300638, -1.31714556,  0.8443035 ])

>>> numpy.size(b, 0)
3
>>> numpy.size(b, 1)
2
```

As seen from the examples above indexing is done from element 0 as in the programming language C. If we want to have access to the function definition `zeros` or `ones` from the `numpy` package, we could do the following:

```
>>> import numpy
>>> a = numpy.zeros((10, 1))
>>> b = numpy.ones((10, 1))
```

When importing `numpy` like this we import all `numpy` functions but we need to access the function by a call of type `numpy.<function>` with `<function>` being the specific function definition we want to use. If we do not want to write '`numpy.`' each time we can do the following:

```
>>> from numpy import *
>>> a = zeros((10, 1))
>>> b = ones((10, 1))
```

Use of the wildcard '`*`' is not generally advised since it normally imports many more function definitions than normally needed. This uses memory and slows things down. Further, different packages may exist with the same function names, which may confuse the programmer. We should not pollute the namespace with functions we do not intend to use. However, in an interactive session where we may wish to have easy access to many functions during a development phase it is not considered to be problematic. But in for written scripts/modules/package it is considered good programming style to only import the specific function definitions which are actually used.

Another thing we should remember is the difference in the arrays defined as:

```
>>> import numpy
>>> a = numpy.random.normal(size=(10))
>>> b = numpy.random.normal(size=(10, 1))
>>> c = numpy.random.normal(size=(1, 10))
```

Here the array `a` is just a one-dimensional array and we access an element as `a[3]`. But `b` is a column vector where we access the value as `b[3, 0]`, and `c` is a row vector where a value is acquired as `c[0, 3]`. It is crucial to remember that `b` and `c` as column and row vectors, respectively, are in fact just arrays of the special kind where there is one column or one row, respectively.

2.5.2 `scipy` – *Scientific Python*

`scipy` is a library to support scientific computations in Python, which is built on top of `numpy`. The `scipy` package includes many areas such as optimisation, polynomial fitting, integration, linear algebra, some signal processing, filters, file i/o, statistical functions etc. The `scipy` functions are made with generality in mind and in cases where both `numpy` and `scipy` provide a given functionality it is normally `scipy` that has the most general approach.

Example 2.7: As an example we use the `scipy.integrate` to numerically integrate the function $f(x) = x^2$ in the interval $x \in [0; 1]$, which can be done as:

```
>>> import numpy
>>> import scipy.integrate
>>> XMIN, XMAX, NOPOINTS = 0.0, 1.0, 9
>>> x = numpy.linspace(XMIN, XMAX, NOPOINTS, endpoint=True)
>>> def g(x): return x**2

>>> scipy.integrate.simps(g(x), x)
0.33333333333333331

>>> scipy.integrate.trapz(g(x), x)
0.3359375
```

```
>>> scipy.integrate.romberg(g, 0, 1)
0.33333333333333331

>>> scipy.integrate.quadrature(g, 0, 1)[0]
0.33333333333333315

>>> scipy.integrate.quad(g, 0, 1)[0]
0.33333333333333337

>>> scipy.integrate.quad(lambda x: x**2, 0, 1)[0]
0.33333333333333337
```

There are different options not used above with respect to e.g. error estimate. The keyword `lambda` defines an anonymous function with argument `x` which returns the result of the expression `x**2`.

Code: **scipy_integrate**

2.5.3 `matplotlib` – *Plotting*

Two-dimensional plotting in Python is normally handled by `matplotlib` [33, 34, 69].[23] Simple 3D-plotting can also be handled by the `matplotlib` toolkit named `mplot3d`. For advanced 3D-plotting it is recommended to use `Mayavi` [64]. `matplotlib` is very flexible and many different plots can be made. It is also possible to include several sub-plots into one plot. Labels, colors, line attributes etc. can be set as desired by the user. Furthermore, a very powerful LaTeX mode can be used allowing use of LaTeX fonts and equations in the plot. Plots can be saved in many formats such as colour encapsulated postscript, JPEG, PNG, PDF etc.

2.5.4 `numexpr` – *Numerical expressions*

The `numexpr` package allows fast computation of expressions, which is very easy to use and for the right applications it can be very efficient. We feed a string with the expression to compute to `numexpr`. This string is analysed, rewritten if necessary for higher efficiency, and compiled on the fly with a JIT compiler. Further, it directly supports multi-threading and is bypassing Pythons Global Interpreter Lock (GIL). The number of threads to use can be set by the command `numexpr.set_num_threads(`M`)` where M is the number of workers/threads to use.

Example 2.8: As a few simple examples of the use and efficiency of `numexpr` let us consider the computations of $\mathbf{y} = \mathbf{a}^{\circ 2} + \mathbf{b}^{\circ 2} + 3\,\mathbf{a} \circ \mathbf{b}$ and $\mathbf{y} = \mathbf{a}^{\circ 10} + \mathbf{a}^{\circ 7} + \mathbf{a}^{\circ 2} \circ \mathbf{b}^{\circ 3}$ where $\circ$ indicates Hadamard (element-wise) multiplication and $\mathbf{a}^{\circ \ell}$ is the Hadamard power of order ℓ. We have:

```
>>> import numpy as np
>>> import numexpr as ne
>>> a = np.random.random_integers(1, 10000, 7E6).astype(np.float64)
>>> b = np.random.random_integers(1, 10000, 7E6).astype(np.float64)

>>> computation = 'a**2 + b**2 + 3*a*b'
>>> %timeit eval(computation)
10 loops, best of 3: 144 ms per loop
```

[23] See `http://matplotlib.sourceforge.net`, which is an excellent source of information on `matplotlib`. For example, a huge catalogue of example plots with full source code included.