1 Welcome to Python

In this introductory chapter, we welcome the reader to Python and make some arguments that we hope will serve to motivate Python programming in finance.

1.1 WHY PYTHON?

We contend that the Python programming language is particularly suited to quantitative analysts/programmers working in the field of financial engineering. This assertion centres on two axes: the first, Python's expressiveness and high-level nature; the second, Python's extensibility and interoperability with other programming languages. Other (arguably not so,) minor arguments to be made for Python programming in general, are the benefits to be had from the use of Python's wealth of standard libraries ('Python comes with batteries included') and Python's support for functional programming idioms.

We certainly do not wish to assert that Python is 'better' in any way than other programming languages (we rejoice in the diversity of programming languages!), but instead wish to emphasise how Python can interoperate with and complement other languages to be found in financial institutions.

1.1.1 Python is a General-Purpose High-Level Programming Language

Python's high-level nature and its rich collection of built-in data types serve to allow the analyst/programmer to focus more on the problems they are solving and less on low-level mechanical constructs relating to such things as memory management in contrast to other programming languages in common use in this domain. Taken together with the simplicity and renowned expressiveness of the Python programming language syntax, this goes some way to explaining the often reported large productivity pickups that result from choosing Python over other languages. As another consequence of these features, programs in Python can be expected to be much shorter and more concise than their representations in other programming languages.

For quantitative analysts, and indeed computational scientists in general, very useful Python packages exist to make the task of numerical analysis programs much easier (SciPy).[1] In addition, quantitative analysts 'in the field' well know that writing programs for finance will often typically involve much more than numerical code alone, as many of these programs are concerned with acquiring and organising data on which the numerical aspects of the program are applied. We have often found that these tasks can be achieved in less lines of code and with significantly less effort in Python than other programming languages.

[1] SciPy is open-source (Python) software for mathematics, science and engineering. See http://www.scipy.org for details for example.

1.1.2 Python Integrates Well with Data Analysis, Visualisation and GUI Toolkits

Another compelling argument for the use of Python by quantitative analysts is the ease with which Python integrates with visualisation software such as GNUPlot[2] making it possible for the analyst to construct personalised 'Matlab-like'[3] enivronments. Furthermore, quantitative analysts generally have neither the interest or time to invest in producing graphical user interfaces (GUIs). They can be nonetheless important. Python provides Tk-based[4] GUI tools making it straightforward to wrap programs into GUIs. Readers interested in learning more about how Python can be integrated with GUI building, data analysis and visualisation software are particularly recommended to consult Hans Peter Langtangen's *Python Scripting for Computational Science* [14].

1.1.3 Python 'Plays Well with Others'

A variety of techniques exist to extend Python from the C and C++ programming languages. Conversely, a Python interpreter is easily embedded in C and C++ programs. In the world of financial engineering, C/C++ prevails and large bodies of this code exist in most financial institutions. The ability for new programs to be written in Python that can interoperate with these code investments is a huge victory for the analyst and the institutions considering its use.

1.2 COMMON MISCONCEPTIONS ABOUT PYTHON

There are a number of ill-informed arguments oft encountered that, when made, impede the propogation or acceptance of Python programming in finance. The most common include 'it is not fast enough', 'it does not engender a clear structure to your code' and (the most incorrect proposition) 'it has no type checking'. In fact, for most applications Python is 'fast enough' and those parts of the application that are computationally intensive can be implemented in fast 'traditional' programming languages like C or C++, bringing the best of both worlds. As for the argument that Python does not engender a clear structure to code, this is hard to understand. Python supports encapsulation at the function, class and namespace levels as well as any of the modern object-oriented or multiparadigm programming languages. Now, what about Python having no type checking? This is simply wrong. Python is dynamically typed, that is to say, type checking is performed at run-time but type checking does happen! Furthermore, the absence of explicit type declarations in the code is one of the keys to why a Python program can be so much more succinct and faster to produce than languages with static type checking. Staying with the topic of Python's type system, it is interesting to note that Python's dynamic type system implicitly supports generic programming. Consider an example taken from the `ppf.math`[5] module

```
def solve_tridiagonal_system(N, a, b, c, r):
  ...
  return result
```

[2] GNUPlot is a cross platform function plotting utility. See http://www.gnuplot.info for details.

[3] Matlab is a numerical computing environment and programming language popular in both industry and academia. See http://www.mathworks.com/ for details.

[4] Tk is an open-source, cross-platform graphical user interface toolkit. See http://www.tcl.tk for details.

[5] Look ahead to the section 'Roadmap for this book' for an explanation of PPF.

Here `N` is the dimension of an $N \times N$ linear system, `a, b, c` are the subdiagonal, diagonal, and superdiagonal of the system respectively, and `r` the right hand side. The point to be made is that the function will work with any types that are consistent with being *Indexable* (i.e. satisfy an *Indexable* concept in the C++0x[6] sense of the word). This admits the use of the function with Python lists, NumPy[7] arrays or some other user-defined array type ... generic programming!

1.3 ROADMAP FOR THIS BOOK

Chapter-by-chapter this book gradually presents a practical body of working code referred to as PPF or the `ppf` package, that implements a minimal but extensible Python-based financial engineering system.

Chapter 2 looks at the overall topology of the `ppf` package, its dependencies and how to build, install and test it (newcomers to Python may be served by looking ahead to Appendix A where a quick tutorial on Python basics is offered).

Chapter 3 considers the topic of implementing Python extension modules in C++ with an emphasis on fostering interoperability with existing C++ financial engineering systems and, in particular, how certain functionality present in `ppf` in fact is underlied by C++ in this fashion.

Chapter 4 lays the groundwork for later chapters (concerned with pricing using techniques from numerical analysis) in that it presents those mathematical algorithms and tools that arise over and over again in computational quantitative analysis, including:

(1) (pseudo) random number generation;
(2) estimation of the standard normal cumulative distribution function;
(3) a variety of interpolation schemes;
(4) root-finding algorithms;
(5) various operations for linear algebra;
(6) generalised linear least-squares data fitting;
(7) stable calculation techniques for computing quadratic and cubic roots; and
(8) calculation of the expectation of a function of a random variable.

Chapter 5 looks at how the `ppf` represents common market information such as discount-factor functions and volatility surfaces.

Chapter 6 is entirely concerned with looking at the data structures used in the `ppf` for representing financial structures: 'flows', 'legs', 'exercise opportunities', 'trades' and the like.

Chapter 7 details the concepts and classes that govern the interactions between the trade representations and pricing models in the `ppf` package.

Chapter 8 offers an implementation of a fully functional Hull–White model in Python, where the characteristic features of the model are assembled from (in as much as is possible) functionally orthogonal components.

Chapter 9 present two general numerical pricing frameworks invariant over pricing models: one lattice based, the other Monte-Carlo based.

[6] The next version of the C++ standard, expected to be completed in 2009.

[7] The fundamental package for scientific computing with Python. SciPy (as indeed PPF) depends on NumPy. See http://numpy.scipy.org for details.

Chapter 10 applies the pricing frameworks and the Hull–White model developed in the preceding chapters to pricing financial structures, specifically, Bermudan swaptions and target redemption notes.

Chapter 11, while keeping things tractable, introduces the idea of and practical techniques for C++/Python 'Hybrid Systems' against the backdrop of existing derivative security pricing and risk management systems in C++.

Chapter 12 gives concrete examples of implementing COM servers in Python and utilising the functionality so exposed in the context of Microsoft Excel.

In the appendices section, Appendix A offers newcomers to Python a brief tutorial. Appendix B provides a primer for the use of the C++ Boost.Python library for fostering interoperability between C++ and Python. Appendix C covers the mathematics of the Hull–White model and Appendix D the mathematics of a simple regression scheme for determining the early exercise premium of a callable structure when pricing using Monte-Carlo techniques.