
Financial Modelling in Python

S. Fletcher & C. Gardner



A John Wiley and Sons, Ltd., Publication

Disclaimer: This eBook does not include ancillary media that was packaged with the printed version of the book.

This edition first published 2009
© 2009 John Wiley & Sons Ltd

Registered office

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book. This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Library of Congress Cataloging-in-Publication Data

Fletcher, Shayne.

Financial modeling in Python / Shayne Fletcher and Christopher Gardner.

p. cm. — (Wiley finance series)

Includes bibliographical references and index.

ISBN 978-0-470-98784-1 (cloth : alk. paper) 1. Finance—Mathematical models—Computer programs.

2. Python (Computer program language) I. Gardner, Christopher. II. Title.

HG106.F59 2009

332.0285'5133—dc22

2009019336

ISBN 978-0-470-98784-1

A catalogue record for this book is available from the British Library.

Typeset in 10/12pt Times by Aptara Inc., New Delhi, India

Printed in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire

Contents

1	Welcome to Python	1
1.1	Why Python?	1
1.1.1	Python is a general-purpose high-level programming language	1
1.1.2	Python integrates well with data analysis, visualisation and GUI toolkits	2
1.1.3	Python ‘plays well with others’	2
1.2	Common misconceptions about Python	2
1.3	Roadmap for this book	3
2	The PPF Package	5
2.1	PPF topology	5
2.2	Unit testing	6
2.2.1	doctest	6
2.2.2	PyUnit	7
2.3	Building and installing PPF	7
2.3.1	Prerequisites and dependencies	7
2.3.2	Building the C++ extension modules	8
2.3.3	Installing the PPF package	9
2.3.4	Testing a PPF installation	9
3	Extending Python from C++	11
3.1	Boost.Date_Time types	11
3.1.1	Examples	12
3.2	Boost.MultiArray and special functions	17
3.3	NumPy arrays	19
3.3.1	Accessing array data in C++	19
3.3.2	Examples	23
4	Basic Mathematical Tools	27
4.1	Random number generation	27
4.2	$N(\cdot)$	28
4.3	Interpolation	29
4.3.1	Linear interpolation	31

4.3.2	Loglinear interpolation	32
4.3.3	Linear on zero interpolation	32
4.3.4	Cubic spline interpolation	33
4.4	Root finding	35
4.4.1	Bisection method	35
4.4.2	Newton–Raphson method	36
4.5	Linear algebra	38
4.5.1	Matrix multiplication	38
4.5.2	Matrix inversion	38
4.5.3	Matrix pseudo-inverse	39
4.5.4	Solving linear systems	39
4.5.5	Solving tridiagonal systems	39
4.5.6	Solving upper diagonal systems	40
4.5.7	Singular value decomposition	42
4.6	Generalised linear least squares	44
4.7	Quadratic and cubic roots	46
4.8	Integration	49
4.8.1	Piecewise constant polynomial fitting	49
4.8.2	Piecewise polynomial integration	51
4.8.3	Semi-analytic conditional expectations	57
5	Market: Curves and Surfaces	63
5.1	Curves	63
5.2	Surfaces	64
5.3	Environment	65
6	Data Model	69
6.1	Observables	69
6.1.1	LIBOR	70
6.1.2	Swap rate	74
6.2	Flows	79
6.3	Adjuvants	82
6.4	Legs	84
6.5	Exercises	85
6.6	Trades	87
6.7	Trade utilities	88
7	Timeline: Events and Controller	93
7.1	Events	93
7.2	Timeline	94
7.3	Controller	97
8	The Hull–White Model	99
8.1	A component-based design	99
8.1.1	Requestor	100
8.1.2	State	101
8.1.3	Filler	104

8.1.4	Rollback	108
8.1.5	Evolve	112
8.1.6	Exercise	115
8.2	The model and model factories	118
8.3	Concluding remarks	121
9	Pricing using Numerical Methods	123
9.1	A lattice pricing framework	123
9.2	A Monte-Carlo pricing framework	128
9.2.1	Pricing non-callable trades	129
9.2.2	Pricing callable trades	131
9.3	Concluding remarks	142
10	Pricing Financial Structures in Hull–White	145
10.1	Pricing a Bermudan	145
10.2	Pricing a TARN	152
10.3	Concluding remarks	157
11	Hybrid Python/C++ Pricing Systems	159
11.1	nth_imm_of_year revisited	159
11.2	Exercising nth_imm_of_year from C++	161
12	Python Excel Integration	165
12.1	Black–scholes COM server	165
12.1.1	VBS client	167
12.1.2	VBA client	167
12.2	Numerical pricing with PPF in Excel	168
12.2.1	Common utilities	168
12.2.2	Market server	169
12.2.3	Trade server	176
12.2.4	Pricer server	187
Appendices		191
A	Python	193
A.1	Python interpreter modes	193
A.1.1	Interactive mode	193
A.1.2	Batch mode	193
A.2	Basic Python	194
A.2.1	Simple expressions	194
A.2.2	Built-in data types	195
A.2.3	Control flow statements	197
A.2.4	Functions	200
A.2.5	Classes	201
A.2.6	Modules and packages	203
A.3	Conclusion	205

B Boost.Python	207
B.1 Hello world	207
B.2 Classes, constructors and methods	207
B.3 Inheritance	209
B.4 Python operators	212
B.5 Functions	212
B.6 Enums	214
B.7 Embedding	214
B.8 Conclusion	216
C Hull–White Model Mathematics	217
D Pickup Value Regression	219
Bibliography	221
Index	223

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 benefit 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 significantl 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 GNUPlot² making it possible for the analyst to construct personalised ‘Matlab-like’³ environments. 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-based⁴ 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 propagation 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++0x⁶ sense of the word). This admits the use of the function with Python lists, NumPy⁷ arrays or some other user-define 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.

The PPF Package

The source code accompanying this book implements a minimal library, `ppf`, for exploring financial modelling in Python. The sections ahead outline the structure and ideas of the package.

The following is a first example of a financial program expressed in Python – the ‘Hello World’ of Quantitative Analysis programs, that is, the Black–Scholes formula for a European option on a single asset:

```
from math import log, sqrt, exp
from ppf.math import N

def black_scholes(S, K, r, sig, T, CP, *arguments, **keywords):
    """The classic Black and Scholes formula.

    >>> print black_scholes(S=42., K=40., r=0.1, sig= 0.2, T=0.5,
    CP=CALL) 4.75942193531

    >>> print black_scholes(S=42., K=40., r=0.1, sig= 0.2, T=0.5,
    CP=PUT) 0.808598915338

    """
    d1 = (log(S/K) + (r + 0.5*(sig*sig))*T)/(sig*sqrt(T))
    d2 = d1 - sig*sqrt(T)

    return CP*S*N(CP*d1) - CP*K*exp(-r*T)*N(CP*d2)

CALL, PUT = (1, -1)

def _test():
    import doctest
    doctest.testmod()

if __name__ == '__main__': _test()
```

2.1 PPF TOPOLOGY

The `ppf` library is a Python package containing a family of sub-packages. The `black_scholes` function listed above is housed in the `ppf.core` subpackage. The topology of `ppf` is as follows:

```
ppf/
  com/
  core/
  date_time/
```

```
market/  
math/  
model/  
  hull.white/  
    lattice/  
    monte_carlo/  
pricer/  
  payoffs/  
test/  
utility/
```

Here is a brief summary of the nature and main roles of each of the `ppf` sub-packages:

com	COM servers wrapping <code>ppf</code> market, trade and pricing functionality (see Chapter 12).
core	Types and functions relating to the representation of financial quantities such as flows and LIBOR rates.
date_time	Date and time manipulation and computations.
market	Types and functions for the representation of common curves and surfaces that arise in financial programming such as discount factor curves and volatility surfaces.
math	General mathematical algorithms.
model	Code specific to implementing numerical pricing models.
pricer	Types and functions for the purpose of valuing financial structures.
test	The <code>ppf</code> unit test suite.
utility	Utilities of a less numerical, general nature such as algorithms for searching and sorting.

2.2 UNIT TESTING

Code in the `ppf` library employs two approaches to testing: interactive Python session testing using the `doctest` module and formalised unit testing using the `PyUnit` module. Both of these testing frameworks are part of the Python standard libraries.

2.2.1 doctest

The way that the `doctest` module works is to search a module for pieces of text that look like interactive Python sessions, and then to execute those sessions to verify that they work as expected. In this way `ppf` modules come with a form of tutorial-like executable documentation:

```
C:\Python25\lib\site-packages\ppf\core>python black_scholes.py -v  
python black_scholes.py -v  
Trying:  
    print black_scholes(S=42., K=40., r=0.1, sig= 0.2, T=0.5, CP=CALL)  
Expecting:  
    4.75942193531  
ok
```

```

Trying:
  print black_scholes(S=42., K=40., r=0.1, sig= 0.2, T=0.5, CP=PUT)
Expecting:
  0.808598915338
ok
2 items had no tests:
  __main__
  __main__..test
1 items passed all tests:
  2 tests in __main__.black_scholes
2 tests in 3 items.
2 passed and 0 failed.
Test passed.

```

2.2.2 PyUnit

A full suite of unit tests for all modules in the `ppf` package is provided in the `ppf.test` sub-package. The tests can be run module-by-module or, to execute all tests in one go, a driver ‘`test.all.py`’ is provided:

```

C:\Python25\Lib\site-packages\ppf\test>python test.all.py --verbose
python test.all.py --verbose
test_call (test_core.black_scholes.tests) ... ok
test_put (test_core.black_scholes.tests) ... ok
test (test_core.libor_rate.tests) ... ok
.
.
.
test_upper_bound (test_utility.bound.tests) ... ok
test_equal_range (test_utility.bound.tests) ... ok
test_bound (test_utility.bound.tests) ... ok
test_bound_ci (test_utility.bound.tests) ... ok

-----
Ran 51 tests in 25.375s

OK

```

2.3 BUILDING AND INSTALLING PPF

In this section we look at what it takes to build and install the `ppf` package.

2.3.1 Prerequisites and Dependencies

`ppf` is composed of a mixture of pure Python modules underlied by some supporting extension modules implemented in standard C++. Accordingly, to build and install `ppf` requires a modern C++ compiler. The C++ extension modules have some library dependencies of their own, notably the Boost C++ libraries and the Blitz++ C++ library. Instructions for downloading

and installing the Boost C++ libraries can be found at <http://www.boost.org> and instructions for Blitz++ can be found at <http://www.oonumerics.org>. Naturally, an installation of Python is also required. On Windows, the authors favour the freely available ActiveState Python distribution, see <http://www.activestate.com> for download and installation details. Also required on the Python side for ppf is an installation of the NumPy package, see <http://www.scipy.org> for download and installation details.

2.3.2 Building the C++ Extension Modules

The ppf C++ extension modules are most conveniently built using the Boost.Build system¹ a copy of which is included with the ppf sources. Also provided with the ppf sources for the convenience of Windows users is a pre-built executable 'bjam.exe'. Although these notes will become a little Windows-centric at this point, the basic principles will hold for *NIX users also. On Windows, the ppf package has been successfully built and tested with the Microsoft Visual Studio C++ compiler versions 7.1, 8.0 (express edition), 9.0 (express edition), mingw/gcc-3.4.5,² mingw/gcc-4.3.0 with Python versions 2.4 and 2.5, Boost versions 1.33.1, 1.34.0, 1.35, 1.36, 1.37 and Blitz++ version 0.9. The ppf package has also been built and tested on the popular Linux-based operating system, Ubuntu-8.04.1 with Boost version 1.36.0, Blitz++ version 0.9 and gcc-4.2.3.

In the remainder of this section, without loss of generality, we will assume a Windows operating system, Blitz++ version 0.9, the ActiveState distribution of Python version 2.5 and Boost version 1.36.

Build Instructions

- Prerequisites

- Copy `c:/path/to/ppf/ext/bjam.exe` to somewhere in your `%PATH%`
- Install
 - o Blitz++-0.9
 - o Boost-1.36
 - o ActiveState Python 2.5
 - o NumPy for Python 2.5 (version 1.0.4 or 1.1.0)
- Edit as appropriate for your site
 - o `c:/path/to/ppf/ext/build/user-config.jam`
 - o `c:/path/to/ppf/ext/build/site-config.jam`

- Build

- `c:/path/to/ppf>cd ext&&bjam [debug|release]`
This will create:
 - o `c:/path/to/ppf/ppf/math/ppf_math.pyd` and
 - o `c:/path/to/ppf/ppf/date.time/ppf_date.time.pyd`

¹ See <http://www.boost.org/doc/tools/build/index.html>.

² Minimalist GNU for Windows – see <http://www.mingw.org>.

2.3.3 Installing the PPF Package

Assuming the steps of the previous section have been performed, installation of the `ppf` package which relies on the standard Python Distutils package is very simple.

- Install

```
- c:/path/to/ppf>python setup.py install
```

which will copy the `ppf` package to the standard Python installation location (`c:/python25/lib/site-packages/ppf`).

2.3.4 Testing a PPF Installation

The easiest way to verify a `ppf` installation is to run the `ppf` unit test suite.

- Test

```
- c:/python25/lib/site-packages/ppf/test>python test_all.py --  
verbose
```

It is usual in financial institutions that make use of quantitative analysis programs to have a considerable investment in C++. Thus it can be important to foster interoperability between C++ and Python. This chapter studies how Python modules can be implemented in C++ by means of the Boost.Python¹ library (see also Appendix B for a primer on the Boost.Python library).

3.1 BOOST.DATE_TIME TYPES

It is common in quantitative analysis programming to require manipulation of and computations involving dates. The ‘Python Library’ contains excellent functionality for such activities. Pricing systems written in C++, however, will be implemented using C++ datatypes for the representation of dates and times. For pricing frameworks implemented in a hybrid of Python and C++, it would be convenient to settle on a common representation of these fundamental types. Accordingly, in this section we demonstrate the ‘reflection’ of functionality from the C++ Boost.Date_Time library to Python.

Our reflection of the C++ date types into Python will be housed in the Python module ‘ppf_date_time.pyd’, implemented in C++. We declare this intention in the entry point to our Python module in the file ‘module.cpp’:

```
#include <boost/python/module.hpp>

namespace ppf
{
    namespace date_time
    {
        void register_date();
        void register_date_more();

    } // namespace date_time

} // namespace ppf

BOOST_PYTHON_MODULE(ppf_date_time)
{
    using namespace ppf::date_time;

    register_date();
    register_date_more();
}
```

¹ Boost provides free peer-reviewed portable C++ source libraries. See <http://www.boost.org> for details.

In 'register_date.cpp' we instantiate Boost.Python class_ objects describing the C++ types and functions we intend to use from Python:

```
void register_date()
{
    using namespace boost::python;
    namespace bg = boost::gregorian;
    namespace bd = boost::date_time;

    // types and functions ...

    class_<bg::date>(
        "date"
        , "A date type based on the gregorian calendar"
        , init<>("Default construct not a date-time")
        .def(init<bg::date const&>())
        .def(init
            <
                bg::greg_year
            , bg::greg_month
            , bg::greg_day
            >((arg("y"), arg("m"), arg("d")))
            , "Main constructor with year, month, day ")
        .def("year", &bg::date::year)
        .def("month", &bg::date::month)
        .def("day", &bg::date::day)

        // ...

        ;

    class_<std::vector<bg::date> >(
        "date_vec"
        , "vector (C++ std::vector<date> ) of date")
        .def(vector_indexing_suite<std::vector<bg::date> >())
        ;

    // more types and functions ...
}
```

Once exposed in this fashion, the types so define in the ppf_date_time module are imported into the ppf subpackage ppf.date_time by means of import statements in the module's '.__init__.py':

```
from ppf.date_time import *
```

3.1.1 Examples

IMM Dates

As an example of what we have achieved, let's see how, in Python, we can compute so-called IMM (international money market) dates for a given year, i.e. the 3rd Wednesday of March, June, September, and December in the year. The ppf.date_time package provides the

module `nth_imm_of_year` in which is define class `nth_imm_of_year`. The workhorse of the class implementation is the `Boost.Date_Time` function `nth_kday_of_month`:

```
from ppf.date_time import \
    weekdays \
    , months_of_year \
    , nth_kday_of_month \
    , year_based_generator

class nth_imm_of_year(year_based_generator):
    '''Calculate the nth IMM date for a given year

    '''
    first = months_of_year.Mar
    second = months_of_year.Jun
    third = months_of_year.Sep
    fourth = months_of_year.Dec

    def __init__(self, which):
        year_based_generator.__init__(self)
        self._month = which

    def get_date(self, year):
        return nth_kday_of_month(
            nth_kday_of_month.third
            , weekdays.Wednesday
            , self._month).get_date(year)

    def to_string(self):
        pass
```

Exercising the class `nth_imm_of_year` functionality in an interactive Python session goes like this:

```
>>> from ppf.date_time import *
>>> imm = nth_imm_of_year
>>> imm_dates = []
>>> imm_dates.append(imm(imm.first).get_date(2005))
>>> imm_dates.append(imm(imm.second).get_date(2005))
>>> imm_dates.append(imm(imm.third).get_date(2005))
>>> imm_dates.append(imm(imm.fourth).get_date(2005))
>>> for t in imm_dates:
...     print t
2005-Mar-16
2005-Jun-15
2005-Sep-21
2005-Dec-21
```

With class `nth_imm_of_year` some useful questions regarding IMM dates can now be answered elegantly and easily. For example, what is the IMM date immediately preceding a given date? This is answered in the `ppf.date_time.first_imm_before` module:

```
from ppf.date_time import \
    weekdays \
```

```
, months_of_year      \  
, nth_kday_of_month   \  
, year_based_generator  
from nth_imm_of_year import *  
  
def first_imm_before(start):  
    '''Find the IMM date immediately preceding the given date.  
    '''  
    imm = nth_imm_of_year  
    first_imm_of_year = imm(imm.first).get_date(start.year())  
    imm_date = None  
    if start <= first_imm_of_year:  
        imm_date = imm(imm.fourth).get_date(start.year() - 1)  
    else:  
        for imm_no in reversed([imm.first, imm.second, imm.third,  
                                imm.fourth]):  
            imm_date = imm(imm_no).get_date(start.year())  
            if imm_date < start:  
                break  
  
    return imm_date
```

In an interactive Python session:

```
>>> from ppf.date_time import *  
>>> print first_imm_before(date(2007, Jun, 27))  
2007-Jun-20
```

The `ppf.date_time` package also contains the symmetric `first_imm_after` function.

Holidays, Rolls and Year Fractions

Other common activities in financial modelling include determining if a date is a business day, 'rolling' a date to a business day and the computation of elapsed time between two dates according to common market conventions.

The `ppf.date_time.shift_convention` module shows an easy way to emulate C++ enum types:

```
class shift_convention:  
    none          \  
, following     \  
, modified_following \  
, preceding     \  
, modified_preceding = range(5)
```

This idiom is employed again in the `ppf.date_time.day_count_basis` module:

```
class day_count_basis:  
    basis_30360   \  
, basis_act_360 \  
, basis_act_365 \  
, basis_act_act = range(4)
```

The `ppf.date_time.is_business_day` module provides the means to answer the question of whether or not a given date is a business day:

```
from ppf.date_time import weekdays

def is_business_day(t, financial_centres=None):
    ''' Test whether the given date is a business day.
        In this version, only weekends are considered
        holidays.
    '''
    Saturday, Sunday = weekdays.Saturday, weekdays.Sunday
    return t.day_of_week().as_number() != Saturday \
        and t.day_of_week().as_number() != Sunday
```

The `ppf.date_time.shift` module provides functionality to 'shift' a date according to the common market shift conventions:

```
from ppf.date_time import *
from is_business_day import *
from shift_convention import *

def shift(t, method, holiday_centres=None):
    d = date(t)
    if not is_business_day(d):
        if method == shift_convention.following:
            while not is_business_day(d, holiday_centres):
                d = d + days(1)
        elif method == shift_convention.modified_following:
            while not is_business_day(d, holiday_centres):
                d = d + days(1)
            if d.month().as_number() != t.month().as_number():
                d = date(t)
                while not is_business_day(d, holiday_centres):
                    d = d - days(1)
        elif method == shift_convention.preceding:
            while not is_business_day(d, holiday_centres):
                d = d - days(1)
        elif method == shift_convention.modified_preceding:
            while not is_business_day(d, holiday_centres):
                d = d - days(1)
            if d.month().as_number() != t.month().as_number():
                while not is_business_day(d, holiday_centres):
                    d = d + days(1)
        else: raise RuntimeError, "Unsupported method"

    return d
```

The `ppf.date_time.year_fraction` module provides functionality to compute year fractions:

```
from ppf.date_time \
    import date, gregorian_calendar_base
from day_count_basis import *
```

```
is.leap_year = gregorian.calendar_base.is_leap_year

def year_fraction(start, until, basis):
    '''Compute accruals
    '''
    result = 0
    if basis == day_count_basis.basis_act_360:
        result = (until - start).days()/360.0
    elif basis == day_count_basis.basis_act_365:
        result = (until - start).days()/365.0
    elif basis == day_count_basis.basis_act_act:
        if start.year() != until.year():
            start_of_to_year = date(until.year(), 1, 1)
            end_of_start_year = date(start.year(), 12, 31)
            result = (end_of_start_year - start).days()/ \
                (365.0, 366.0)[is_leap_year(start.year())] \
                + (int(until.year()) - int(start.year()) - 1) + \
                (until - start_of_to_year).days()/ \
                (365.0, 366.0)[is_leap_year(until.year())]
        else:
            result = (until - start).days()/ \
                (365.0, 366.0)[is_leap_year(until.year())]
    elif basis == day_count_basis.basis_30360:
        d1, d2 = start.day(), until.day()
        if d1 == 31:
            d1 -= 1
        if d2 == 31:
            d2 -= 1
        result = (int(d2) - int(d1)) + \
            30.0*(int(until.month()) - int(start.month())) + \
            360.0*(int(until.year()) - int(start.year()))
        result = result / 360.0
    else:
        raise RuntimeError, "Unsupported basis"

    return result
```

In the following interactive session, the year fraction between two dates is computed under a variety of different day count basis conventions:

```
>>> from ppf.date_time import *
>>> add_months = month_functor
>>> Nov = months_of_year.Nov
>>> begin = date(2004, Nov, 21)
>>> until = begin + add_months(6).get_offset(begin)
>>> year_fraction(begin, until, day_count_basis.basis_30360)
0.5
>>> year_fraction(begin, until, day_count_basis.basis_act_365)
0.49589041095890413
>>> year_fraction(begin, until, day_count_basis.basis_act_act)
0.49285126132195523
```

3.2 BOOST.MULTIARRAY AND SPECIAL FUNCTIONS

The use of multidimensional arrays in quantitative analysis programs is ubiquitous. Python, or rather the Python libraries provide a variety of types that serve for their representation. Like the date types of the previous section, however, we prefer to emphasise interoperability with C++ and so, to this end, might favour reflection of C++ array types into Python. The `ppf` package exposes the Boost.MultiArray multidimensional array types `boost::multi_array<double,N>` for $N=1,2,3$ to Python. To achieve this, advantage was taken of a C++ template meta-program that facilitates reflection of the arrays, the code for which is present in the source code accompanying this book (see ‘`ext/boost/multi_array/multi_array.hpp`’).

The array types are housed in the `ppf.math` module implemented in the C++ Python extension ‘`ppf.math.pyd`’ and imported into the namespace of the `ppf.math` subpackage. Usage of the array types is natural and intuitive. Here is an example taken from the `ppf.math` unit tests:

```
class solve_upper_diagonal_system_tests(unittest.TestCase):
    def test(self):

        # Solve upper diagonal system of linear equations ax = b
        # where
        #
        # a = 3x3
        #   [ 1.75   1.5   -2.5
        #     0     -0.5   0.65
        #     0      0    0.25 ]
        #
        # and b = [0.5, -1.0, 3.5].

        a = ppf.math.array2d([3,3])
        a[0, 0], a[0, 1], a[0, 2] = (1.75, 1.5, -2.5)
        a[1, 0], a[1, 1], a[1, 2] = (0.0, -0.5, 0.65)
        a[2, 0], a[2, 1], a[2, 2] = (0.0, 0.0, 0.25)
        b = ppf.math.array1d([3])
        b[0] = 0.5
        b[1] = -1.0
        b[2] = 3.5

        # Expected solution vector is x = [2.97142857  20.2  14.0].

        x = ppf.math.solve_upper_diagonal_system(a, b)
        assert len(x) == 3 and math.fabs(x[0] - 2.971428571) < 1.0e-6 \
            and math.fabs(x[1] - 20.2) < 1.0e-6 and math.fabs(x[2] -
                14.0) < 1.0e-6
```

In addition to the multi-array types, the module `ppf.math` also exposes some useful utility functions implemented in C++. In the file ‘`ppf/math/limits.hpp`’ are the following template function definitions

```
#if !defined(LIMITS_5DDE828B_9989_44F5_9728_47AA72323D96_INCLUDED)
# define LIMITS_5DDE828B_9989_44F5_9728_47AA72323D96_INCLUDED
```

```
# if defined(_MSC_VER) && (_MSC_VER >= 1020)
#   pragma once
# endif // defined(_MSC_VER) && (_MSC_VER >= 1020)

# include <boost/config.hpp>

# include <limits>

namespace ppf { namespace math {

template <class T>
T epsilon()
{
    return std::numeric_limits<T>::epsilon();
}

template <class T>
T min BOOST_PREVENT_MACRO_SUBSTITUTION ()
{
    return (std::numeric_limits<T>::min)();
}

template <class T>
T max BOOST_PREVENT_MACRO_SUBSTITUTION ()
{
    return (std::numeric_limits<T>::max)();
}

}} // namespace ppf::math

#endif//!defined(LIMITS_5DDE828B_9989_44F5_9728_47AA72323D96_INCLUDED)
```

In 'ext/lib/math/src/register_special_functions.cpp', instantiations of these templates are exposed to Python:

```
#include <boost/python/def.hpp>

#include <ppf/math/limits.hpp>

namespace ppf { namespace math {

void register_special_functions()
{
    using namespace boost::python;

    def("epsilon", epsilon<double>);
    def("min.flt", min BOOST_PREVENT_MACRO_SUBSTITUTION <double>);
    def("max.flt", max BOOST_PREVENT_MACRO_SUBSTITUTION <double>);
}

}} // namespace ppf::math
```

An example of the use of the `epsilon` function is again provided by a `ppf.math` unit test:

```
class bisect_tests(unittest.TestCase):
    def test1(self):
        tol = 5*ppf.math.epsilon()
        left, right, num_its = \
            ppf.math.bisect(lambda x: x*x + 2.0*x - 1.0
                           , -3, -2
                           , lambda x, y: math.fabs(x-y) < tol, 100)
```

Further examples of the use of these special functions can be found in the next chapter.

3.3 NUMPY ARRAYS

Despite the efforts of the preceding section regarding reflection of C++ `Boost.MultiArray` types into Python, in practice, when working in Python, the authors have found the facilities of NumPy arrays to be far more convenient (NumPy was mentioned briefly in section 1.2). Specifically, their notational conveniences and the large body of functionality provided by the NumPy library motivates their use in Python beyond the argument of C++ interoperability. Indeed, when working in C++, a library dedicated to scientific manipulation of arrays such as Blitz++² wins the authors' favour for such work over 'lower-level' container types like native C arrays or `Boost.MultiArray` types. But now to the crux of the matter. If we haven't made this point earlier then we'll make it for the first time now. One of the great strengths of Python is the ability to drop into C or C++ code 'when performance counts'. That is, the ability to factor out that characteristic operation that must be done as efficiently as possible and pull it down into a compiled component is essential. Now, in the field of numerical programming, doesn't that characteristic operation almost always involve operating on arrays of data?

So, can we have it all? Can we have the convenience of NumPy in Python combined with the convenience and efficiency of Blitz++ in C++ where the data is shared between these array types? The short answer is 'yes we can', as we will demonstrate in the next subsection.

3.3.1 Accessing Array Data in C++

This subsection is concerned with the topic of accessing a NumPy array's data in C++. To do this, we need to work with the Python C API and we'll also take advantage of `Boost.Python` where we can. The approach is fairly idiomatic and can be more or less wrapped up in a set of reasonably small utility functions. Let's begin with this most simple of functions from 'ppf/util/python/detail/decreef.hpp':

```
#if !defined(DECREF_4A1F1D9D_CE18_4CA1_AF52_DA1C51847FB4_INCLUDED)
# define DECREF_4A1F1D9D_CE18_4CA1_AF52_DA1C51847FB4_INCLUDED

# if defined(_MSC_VER) && (_MSC_VER >= 1020)
# pragma once
# endif // defined(_MSC_VER) && (_MSC_VER >= 1020)
```

² Blitz++ is a C++ class library for scientific computing which provides performance on par with Fortran 77/90. See <http://www.oonumerics.org/blitz> for details.

- [*download History and Physical Examination \(Current Clinical Strategies\)*](#)
- [**click Elite Panzer Strike Force: Germany's Panzer Lehr Division in World War II pdf, azw \(kindle\), epub**](#)
- [read City on a Grid: How New York Became New York](#)
- [click It Came From Beneath The Sink \(Goosebumps, Book 30\) pdf, azw \(kindle\)](#)

- <http://www.experienceolvera.co.uk/library/Vegan-Cupcakes-Take-Over-the-World--75-Dairy-Free-Recipes-for-Cupcakes-that-Rule.pdf>
- <http://aseasonedman.com/ebooks/Elite-Panzer-Strike-Force--Germany-s-Panzer-Lehr-Division-in-World-War-II.pdf>
- <http://cavalldecartro.highlandagency.es/library/City-on-a-Grid--How-New-York-Became-New-York.pdf>
- <http://pittiger.com/lib/It-Came-From-Beneath-The-Sink--Goosebumps--Book-30-.pdf>