Python for Finance

Risk In Finance

ACADEMIC YEAR 2021-2022

Lecturer: Dr. D. O'Kane

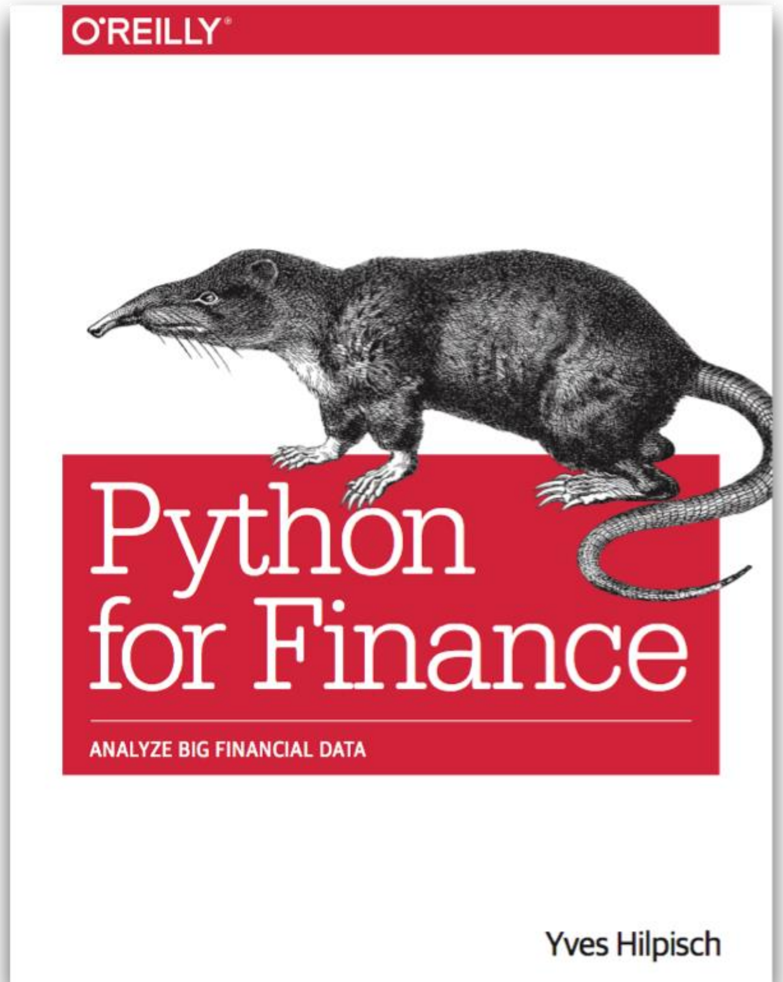
Why Python ?

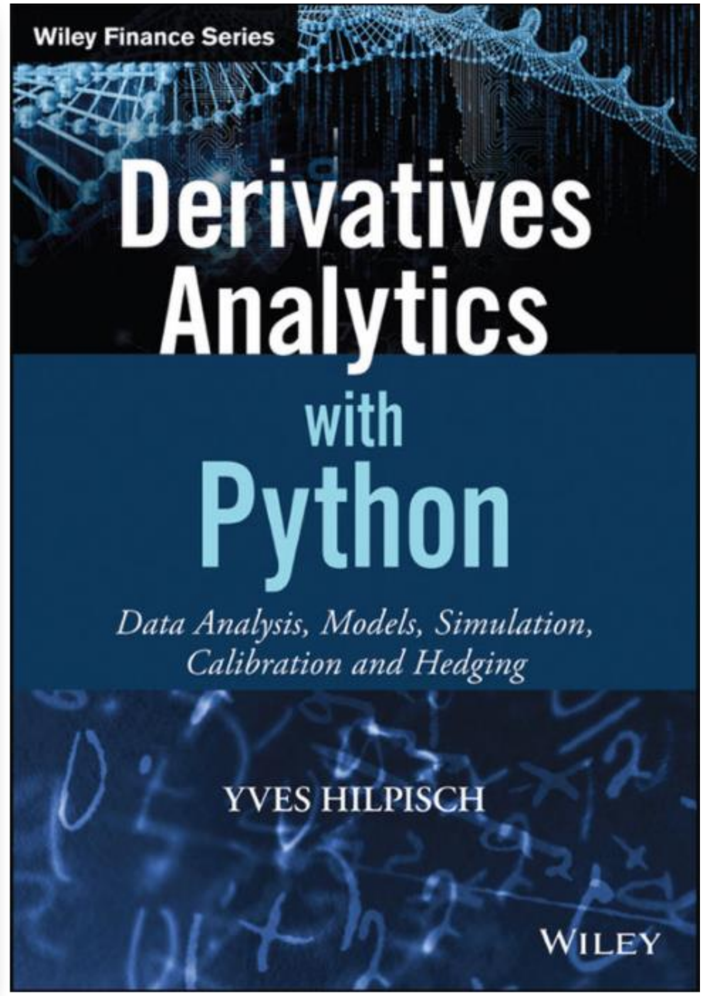
- Most commonly used programming language today
- It is easy to learn as it has a simple syntax
- Python is an open source general purpose language
- Python has a huge community of users especially in finance world
- Pure Python is not as fast as C++ or Fortran or even Matlab
- **But** this has been solved by libraries developed in C++ for Python

What Libraries ?

- Numpy = Library of very fast, basic numerical methods
- Pandas = Library for loading, manipulating and analysing data
- Matplotlib and Seaborn = Libraries for data visualization
- Scipy = Library of scientific mathematical functions
- Numba = Fast JIT compiler makes code run as fast as C++
- Scikit-Learn = Comprehensive machine learning library
- FinancePy = Derivative valuation in Python
- You get the best of both worlds
  - A simple interpreted language
  - Loads of pre-written functions
  - High speed execution

References: Two Books by Yves Hilpisch



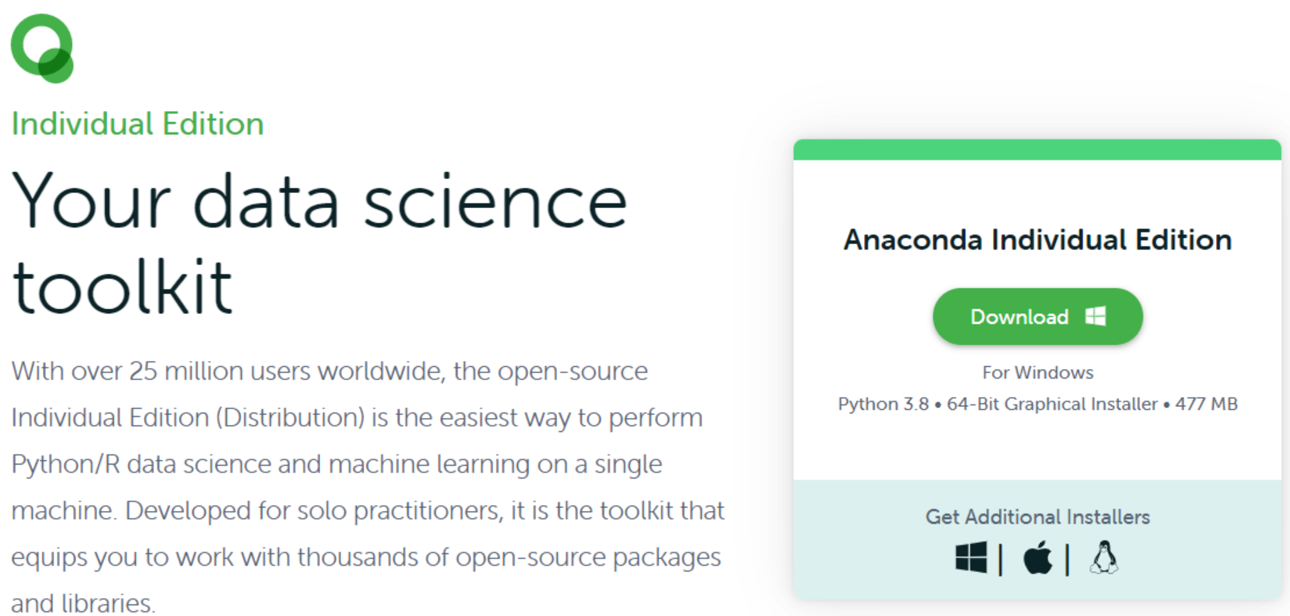


- See https://github.com/yhilpisch/

Python Installation

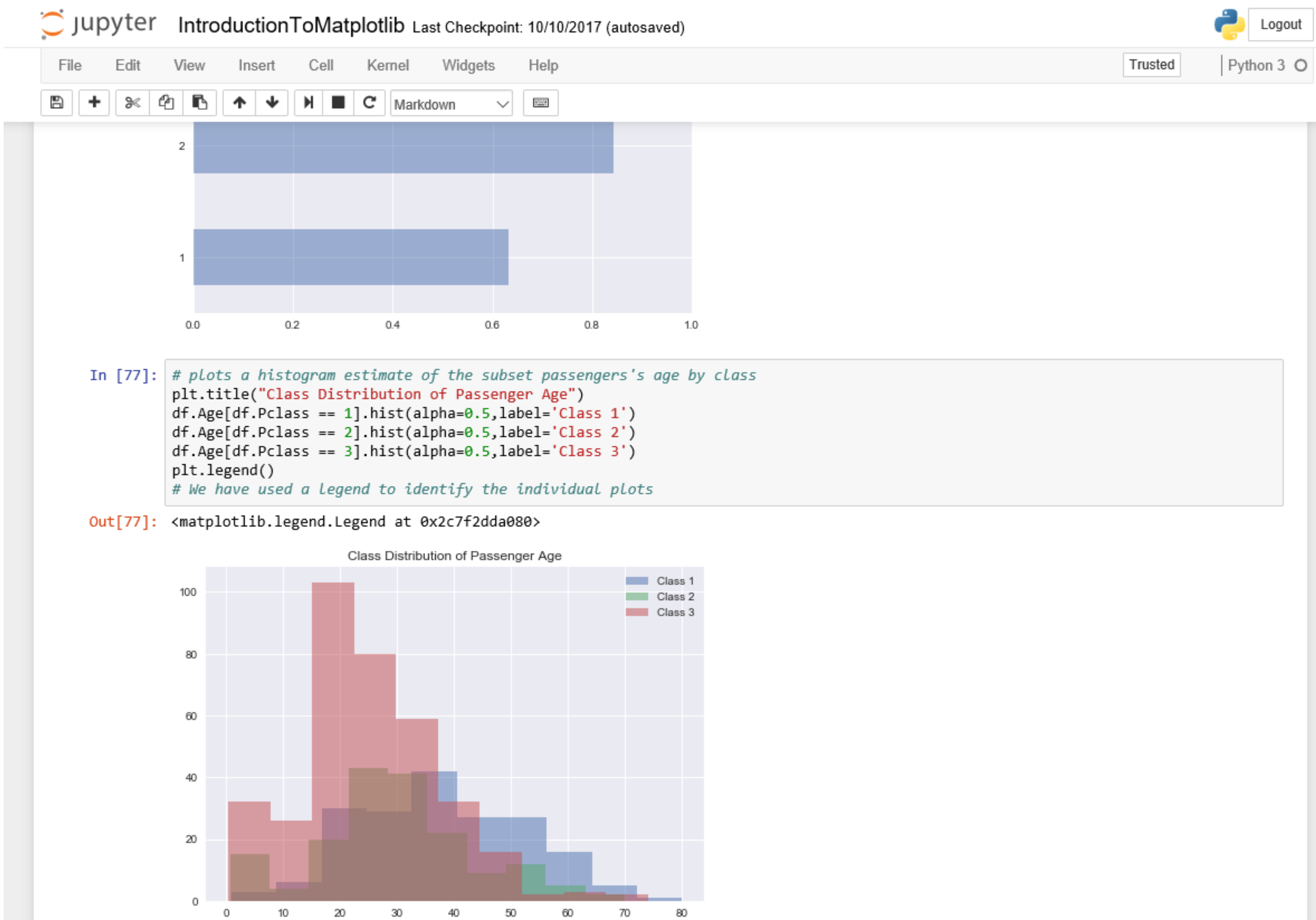
Where to get Python 3

- We are using Python 3 as Python 2 is no longer supported
- Continuum is a company that provides a simple install package
- Get it at https://www.anaconda.com/products/individual
- Then install it – it installs Python + All Important Packages



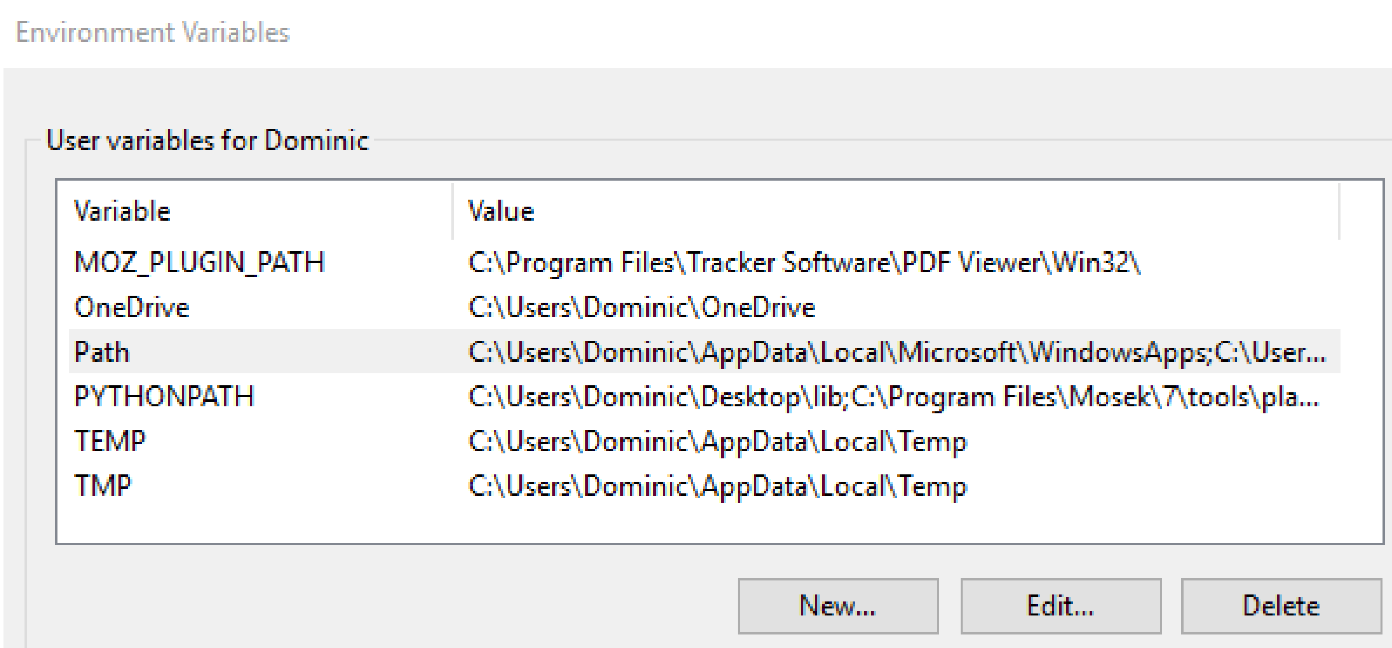
We will use the Jupyter Notebook

- Very powerful environment for analysis e.g. Machine Learning



Starting and Using the Jupyter Notebook

- Open Anaconda cmd and type "**jupyter notebook**"
- If this does not work, add the anaconda directory to your path
- To do this in windows, type 'path' into Cortana
- It should prompt you to edit environment files for user
- Click on this and an Environment variables gui will open



Starting the Jupyter Notebook

- You should edit the variable called 'path'
- On windows add the following directory path to the path

C:\Users\YOURNAME\Anaconda3\Scripts

- Close the path setting window and open a new command line tool using 'cmd'
- Create a folder for your work

Using the Jupyter Notebook

- You type in a cell and press SHIFT + RETURN to calculate it
- Each calculation is numbered in the order of execution
- [*] means that it is still calculating
- Use ESC M to make a cell into a comment
- # is a header 1, ## is a header 2, ### is a header 3 *Transfer Code model into Markdown*
- "%matplotlib inline" displays plots in notebook
- %time allows you to calculate the wall time (does not take into account other processes – like the clock on your wall)
- Allows you to use Latex formulas using $ symbol

Python Coding

Python Datatypes

- Integers

>>> z = 5/2 # This assigns 2.5 to z

- Floats

>>> x = 3.452

- Strings

>>> name = "EDHEC"

- In Python you can do multiple assignments

>>> x, y = 10,20

- This is useful as many functions can return more than one value

Basic Printing in Python

- Can use the print command – **it needs brackets**

```
print('Hello!')
Hello!
```

- Can combine a set of outputs and spaces are automatically added

```
x = 5.4233
print('Hello!' , x)
Hello! 5.4233
```

- You can add strings and then print them – for numerical values you can use str(x) but no space is added *str() transfer the numerical values into string format*

```
print("Hello" + str(x) + "EDHEC")
Hello5.4233EDHEC
```

- Python is first adding the strings and then printing them

Mathematics in Python

- Assignment uses = and comparison uses ==
- Variables are given a type as soon as they are assigned a value
- Python works out the type itself – you may need to help it !
- Standard operators + - * / work as expected
- Remainder function is %
- The logical operators are **and**, **or** and **not**
- Python works out if something is an integer or a float from context – be careful

```
type(15.0)
float
type(1)
int
```

We need Maths Functions

- Raw python does not know mathematical functions apart from +, -, * and / and maybe a few more
- To have access to more functionality we need to import the Python Math library

Instead:
x = math.exp(5.0), which means x = e^5

- To do this we can add the following at the top of our code

```
import math
```

- Now we need to call

```
math.exp(x)
```

- But this is ugly, so instead I write

```
from math import exp
```

- Now I can call it without any prefix – but be careful – if you have another library with an exp function there may be confusion

Fancy Printing in Python

- Can use fancy printing format to control precision. Suppose

```
x = 3.34239239203898239 8; string = 'Hello'
```

- No formatting gives

```
print('x = `,x)
x =  3.342392392038982 4
```

- Formatting code – 'f' is floating and 9.5 = 9 chars with 5 decimal places and **you have to drop the comma and use a % symbol**

```
print('x = %7.5f' % x)
x = 3.34239
```

- Combining strings – formatting code 's' and floats in a tuple

```
print('%s = %6.5f' % (string,x))
Hello = 3.34239
```

Times and Dates in Python

Dates in Python

- Python has its own built-in date and times library

```
import datetime as dt
print('Current date and time: ', dt.datetime.now())
print(dt.datetime.now().strftime('%y/%m/%d %H:%M'))
print('Current year:' , dt.datetime.now().strftime('%Y'))
print('Month of year: ', dt.datetime.now().strftime('%B'))
print('Day of the month : ', dt.datetime.now().strftime('%d'))
```

- The output is

```
Current date and time: 2017-10-19 03:22:31.417640
17/10/19 03:22
Current year: 2017
Month of year: October
Day of the month : 19
```

Differencing Dates

- First, we create two dates

```
import datetime as dt
tradeDate = dt.date(2017,3, 13)  # 13 March 2017
expiryDate = dt.date(2020,3,20)  # 20 March 2020
```

- We can use simple subtraction to get a **timedelta** object

```
dateDiff = expiryDate - tradeDate
print(dateDiff)
1103 days, 0:00:00
```

- We want the number of days as an integer

```
# That is a time delta object - to get the number of days
dateDiff.days
1103
```

Lists, Tuples and Dictionaries

Python Lists

- If we have a list of data, we can store it in a list! A list can be empty initially

```
list1 = []
```

- Or we can populate it – data types can be mixed

```
list1 = ['Edhec', 'Business School', 493, 'Nice']
```

- We access members using square brackets

```
print (list1[0])
>>>Edhec
```

- We can add new members using append

```
list1.append('France')
```

- The command delete can be used to delete an entry
- Using colons [start:end] we can access sublists `list1[1:3]`

Python Matrices

- We can also define a matrix using Python lists
- A matrix is simply a list of lists

```
m = [[1,4,5],[3,2,7]]
```

- We can access the elements as

```
m[0][0]
1
```

```
m[1][2]
7
```

前面是行，后面是列

- Note that the first element has indices m[0][0]
- **However, we do not use Python lists for any computationally heavy matrix calculations**
- We use **Numpy** arrays and matrices - these are 10-30 times faster to use – we will introduce Numpy later

Python Tuples

- A Tuple is like a list, but it is an immutable (unchanging) sequence
- It is defined using round brackets

```
tup1 = ('Edhec', 'Business School', 493, 'Nice')
```

- As with lists we access members using square brackets
- It is faster to iterate a tuple than a list
- The data in a tuple is write-protected

Python Dictionary

- A Dictionary is a list of key and value pairs
- A key is what you use to look up the value
- Keys and values are separated by a colon

```
dict = {'Name' : 'Pierre' , 'Age': 23, 'Nationality' : 'French'}
```

- We access the members as follows

Type 'Tab' to get the list of function key words

```
dict['Name']
Pierre
dict['Age']
23
```

- Elements can be amended directly using their key

```
dict['Age'] = 24
```

List Comprehensions

List Comprehensions using Text

- Powerful tool in Python to do complex operations in a single line

```
list1 = []
for char in 'EDHEC':
   list1.append(char)


print(list1)
['E', 'D', 'H', 'E', 'C']
```

- The list comprehension looks like this

```
list2 = [ char for char in 'EDHEC' ]
print(list2)
['E', 'D', 'H', 'E', 'C']
```

List Comprehensions in Maths

- Powerful tool in Python to do complex operations in a single line

```
list1 = []
for x in range(0,11):
   list1.append(x**2)

print(list1)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- The list comprehension looks like this

```
list2 = [ x**2 for x in range(0,11)]
print(list2)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- It seems sort of backwards

Program Control Flow in Python

Loops using for

- We can loop over any list using the **for command**

```
words = ['cat', 'monkey', 'hippopotamus’]
for w in words:
    print(w, len(w))
```

```
cat 3
monkey 6
hippopotamus 12
```

- If we need to loop over a series then we use **range** which returns series from the first input to last number **exclusive**

```
for i in range(0,4):
    print(i,i**2)
```

```
0 0
1 1
2 4
3 9
```

- If you only give range one number, range will always start at zero

Control Flow using if, elif and else

- Python requires **indentation** instead of brackets to control program flow and a **colon** after each condition

```
x=5

if x < 0:
    print("X is negative")
elif x < 10:
    print("X is positive but less than 10")
else:
    print("X is positive and greater than 10")
```

- Indentation must be 4 spaces or a tab or the program will not run
- You will quickly get use to this style format as you use Python

Control Flow using break

- Sometimes you want to jump out of an inner loop
- You can use the break statement to do this
- The code is on the left and the output on the right

```
for n in range(2, 10):
    prime = True
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n/x)
            prime = False
            break

    if prime == True:
            print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2.0
5 is a prime number
6 equals 2 * 3.0
7 is a prime number
8 equals 2 * 4.0
9 equals 3 * 3.0
```

Functions in Python

Functional Programming

- In Python we can write functions
- Split your code into logical blocks that can be reused
- Python functions can
  - Manage files
  - Draw graphs
  - Calculate and Return values
  - Lots more ..
- We only need to give a name and list of arguments (inputs)
- No need to specify datatypes

Functions: Using def

- We use the **def** keyword to define the function
- Don't give the function the same name as a built-in function

```
from math import exp

def sigmoid(x):
    ''' This function returns the value of the sigmoid function '''
    e = exp(-x)
    s = 1.0/(1.0+e)
    return s
```

- Type help(sigmoid) and you will see the docstring

```
help(sigmoid)
Help on function sigmoid in module __main__: sigmoid(x)
        This function returns the value of the sigmoid function
```

Functions: A function can return multiple arguments

- Suppose we want to return the value of the exponential

```
def sigmoid2(x):
    e = exp(-x); s = 1.0/(1.0+e)
    return s, e
```

- We then call it as follows to get back **a tuple** (the return type)

```
ret = sigmoid2(1.0)
print(ret)
(0.7310585786300049, 0.36787944117144233)
```

- Or to get the individual values directly we use the syntax

```
x, y = sigmoid2(1.0)
print(x,y)
0.7310585786300049, 0.36787944117144233
```

Function Default Arguments

- We use default arguments to set some which almost never change which saves the user from having to input a value

```
import datetime as dt
def yearFraction(d1,d2,daysInYear = 365.252):
    dateDiff = d2 – d1
    yearFraction = dateDiff.days / daysInYear
    return yearFraction
```

- Here is an example

```
startDt = dt.datetime(2011,1,1)
endDt = dt.datetime(2011,6,1)
yearFraction(startDt, endDt)
0.41341320512960916
```

Function Keyword Arguments

- We can call the function using keywords to specify which argument is which

```
startDt = dt.datetime(2011,1,1)
endDt = dt.datetime(2011,6,1)
yearFraction(startDt, endDt)
0.41341320512960916
yearFraction(d1=startDt, d2=endDt)
0.41341320512960916
yearFraction(d2=startDt, d1=endDt)
-0.41341320512960916
```

- This can be very useful when there are lots of arguments and many of them are default arguments

Lambda Functions in Python

- Sometimes you only call a function from one place
- Maybe it needs to know the value of some local variables
- A lambda function can do this better than a function

```
def f(x): return x*x
```

```
f = lambda x: x*x
```

- Think of lambda as the word function and x is the argument
- After the : we have the function itself
- Some simple examples (lambda functions can get complex!)

```
f = lambda x: x**2 + 2*x + 5
```

```
f = lambda x, y, z: x+y+z
```

List Comprehensions

List Comprehensions using Text

- Powerful tool in Python to do complex operations in a single line
- You will see lots of Python people using it
- Consider a standard Python loop

```
list1 = []
for char in 'EDHEC':
   list1.append(char)
print(list1)
['E', 'D', 'H', 'E', 'C']
```

- The list comprehension that does the same looks like this

```
list2 = [ char for char in 'EDHEC' ]
print(list2)
['E', 'D', 'H', 'E', 'C']
```

List Comprehensions in Maths

- Powerful tool in Python to do complex operations in a single line

```
list1 = []
for x in range(0,11):
   list1.append(x**2)

print(list1)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- The list comprehension looks like this

```
list2 = [ x**2 for x in range(0,11)]
print(list2)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- It seems sort of backwards as the function is before the loop

Introduction to Python Notebook 1

Add your notes here

Python
String Manipulation

Python String Manipulation

- Here are some useful commands for manipulating string

s = "EDHEC is a Business School"

- len(s) returns the length of the string which is 26
- s[4] returns 'C'
- s[10:15] returns 'Busi'
- s.count('s') returns 4
- s.find('B') returns 11
- s.find('X') returns -1
- s.replace('a','the') returns 'EDHEC is the Business School'
- s.lower() returns 'edhec is a business school'
- s.upper() returns 'EDHEC IS A BUSINESS SCHOOL'

Python File I/O

Getting a List of Files

- We often need to find the file before we open it (or many)

```
mypath = "./data/“
import os
files = os.listdir(mypath)
print(files)
['.ipynb_checkpoints', 'bondPrice.py', 'giltBondPrices.txt',
'optionPortfolio.csv', 'stocks', 'timeSeriesData.pkl']
```

- To use a pattern matching filter, the **glob** library can be used

```
import glob
files = glob.glob("./data/*.txt")
print(files)
['./data\\giltBondPrices.txt']
```

Python File I/O: Reading from a File

- We open a file and prepare to read from it using open() with 'r'
- This creates a **file object** that we use to extract the file contents
- We read the whole file into a list object using readlines()
- It detects EOL characters in the file and uses these

```
filename = "./data/giltBondPrices.txt"
f = open(filename,'r')
lines = f.readlines()
f.close()
```

- Once you have used the file, close it to release any resources
- The **with** command can do this for you automatically

```
with open(filename,'r') as f:
    lines = f.readlines()
```

Contents of File

- When we read a line from a file, it is usually a string of data fields
- We can simply print the contents

```
lines
```

```
['epic\tdescription\tcoupon\tmaturity\tbid\task\tchange\tincome yield\tgross redemption yield\n',
 'TR13\tUk Gilt Treasury Stk\t4.5\t07-Mar-13\t101.92\t102.07\t-0.01\t4.41\t0.22\n',
 'T813\tUk Gilt Treasury Stk\t8\t27-Sep-13\t107.86\t107.98\t-0.03\t7.41\t0.23\n',
 'TR14\tUk Gilt Treasury Stk\t2.25\t07-Mar-14\t102.9\t103.05\t0.01\t2.18\t0.22\n',
 'T514\tUk Gilt Treasury Stk\t5\t07-Sep-14\t109.28\t109.43\t0.02\t4.57\t0.23\n',
 'TR15\tUk Gilt Treasury Stk\t2.75\t22-Jan-15\t105.57\t105.68\t0.05\t2.6\t0.33\n',
 'T4T\tUk Gilt Treasury Stk\t4.75\t07-Sep-15\t112.92\t113.04\t0.04\t4.2\t0.35\n',
 'TY8\tUk Gilt Treasury Stk\t8\t07-Dec-15\t124.39\t124.55\t0.04\t6.43\t0.34\n',
 'TS16\tUk Gilt Treasury Stk\t2\t22-Jan-16\t104.92\t105.04\t0.07\t1.91\t0.49\n',
 'T16\tUk Gilt Treasury Stk\t4\t07-Sep-16\t113.44\t113.55\t0.08\t3.52\t0.56\n'
```

- The first column is some bond ID
- Then we have string and numeric values
- Looking at the file we see that it is tab separated – "\t"
- There is an end of line character – "\n"

Parsing Data In a File

- We can use replace to remove the end of line character
- We can split the data into columns using the split command

```
dataTable = []
for line in lines:
    line2 = line.replace("\n", "")
    dataFields = line2.split("\t")
    dataTable.append(dataFields)
```

- dataFields becomes a list of strings
- dataTable becomes a list of lists
- We may need to do more work e.g. converting date strings
- In practice we will use Pandas for this sort of work

Python File I/O: Writing to a File

- You open a new file using open() with 'w'
- Or you can open an existing one with 'a' for append
- We can write to the file using the write(string) command
- At the end you shouldn't forget to close the file
- Or better, use the command **with**

```
with open(filename,'w') as f:
    f.write("Hello World")
    ……
```

Introduction to File IO and Strings Notebook 2

Add your notes here

Using the Libraries

The Python Stack

- The power of Python is partly due to its simple language
- **It is MOSTLY due to its set of excellent libraries listed below**

Library	Description
Numpy	Fast mathematics library with vectorization
Scipy	Math functions, statistics and optimizers
Pandas	Advanced manipulation of data tables
Matplotlib	Advanced plotting and visualisation
Statsmodels	Time series analysis / econometrics
Numba	High performance calculations

- These libraries will be our initial focus and then I will show how to combine them to do useful things

Importing Modules

- Most of the power of Python comes from using its very extensive libraries – or in Python we call them **modules**
- These have been written by various individuals and groups and made available for free
- All of the main ones we need came with the Anaconda package
- To access a library from your code you need to import it
- You should import it with a name to prevent names clashing

```
import numpy as np
```

- The np means that to access any Numpy function called **function** we call it with np.function(....)

Using Pip

- **Pip** is the name of the program that you use to install new or update existing libraries in your Python package
- Use it from the Mac or Windows **command line** (not in Python)
- The path to pip should be in your user-defined path
- You can install a project called SomeProject as follows

```
C:\Users>pip install 'SomeProject''
```

- If you want a specific version then use

```
C:\Users> pip install 'SomeProject==1.4'
```

- To upgrade a specific project

```
C:\Users> pip install –upgrade SomeProject
```

Deprecations

- As you work with Python from time to time you will receive warnings such as

```
DeprecationWarning: Implicitly casting between incompatible kinds...
```

- The purpose of Deprecation warnings is to tell you that the form of the library you are using has changed and that this function you called is to be removed
- What happens is that developers fix and improve libraries and may change the way a function is named or called
- They leave the old code so your library does not break
- But they warn you to change your code as the old way will no longer be supported and may not work in future

Libraries: NumPy and Numba

Numpy is the Core Python Scientific Library

- NumPy = Numerical Python
- It's a dedicated library for numerical work
- While we can use Python's built-in lists, they are not efficient
- NumPy uses less memory as it uses numpy arrays
- These assume that the data members are all the same
- For this reason, the calculations are faster
- They have been optimised
- We always import NumPy as follows

```
import numpy as np
```

- Calling it "np" is a widely-used convention

NumPy Functions

- NumPy includes the following features
  - A fast and efficient multidimensional array type **ndarray**
  - This includes powerful shape manipulation functions
  - Indexing and slicing of multidimensional arrays
  - Linear algebra – transpose, dot product, eigenvalues
  - A comprehensive library of mathematical functions
  - Vectorized fast calculations
  - Deep and shallow copies for memory management
- I am not going to do every function here, just the basics
- I will introduce extra functions later as we encounter them

The Foundation of NumPy is its ndarray data type

- **ndarray** is a homogeneous multidimensional array data type

- It has a number of methods which we can call including
  - ndarray.ndim gives the number of dimensions
  - ndarray.shape gives shape tuple of the size in each dimension

- There are several ways to create an **ndarray** which include:
  - It can be created by passing in a Python list (or list of lists)
  - It can be initialised using np.zeros(shape tuple)
  - Or it can be initialised using np.ones (shape tuple)

Creating a NumPy Array

- Creating a NumPy array by converting a Python list

```
a = np.array([3,5,4,2])
```

- We can use zeros to create an 10-element array of **zeros**

```
b = np.zeros( 0)
```

Here is 10 not 0

- Use the **shape tuple** (d1,d2,...) to create multi-dimensional arrays
- To create a 5x9x2 = 3-dimensional array of zeros

```
b = np.zeros( (5,9,2) )
```

- We can create a 2x5 = 2-dimensional array of **ones**

```
C = np.ones( (2,5) )
```

Indexing Arrays

```
v = np.array([12,22,13,44,22,43,35,36])
```

Create Array

```
v[0]
```

```
12
```

Starts at zero

```
v[2:5]
```

```
array([13, 44, 22])
```

2:5 has elements 2,3,4

```
v[5:]
```

```
array([43, 35, 36])
```

5: gives all elements after 5

```
v[5:-1]
```

```
array([43, 35])
```

-1 is from the end

```
v[1:4] = 100
```

Broadcasting

```
v
```

```
array([ 12, 100, 100, 100,  22,  43,  35,  36])
```

Array has changed

Useful Functions: linspace and arange

- Want to get a list of values evenly spaced between two values
- Values between 0 and 10 with 5 evenly spaced values returned

```
np.linspace(0,10,5)
```

```
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

- Note you don't pass 4 (even though it may seem more intuitive)
- If you want a list of integers, then use arange

```
l = np.arange(10)
print(l)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

Random Numbers in NumPy

- Numpy has a lot of random number distributions built-in
- You need to look under numpy.random
- Everything you need is there
- For example, a random int between 1 and 100 is found by

```
np.random.randint(1, 101)
53
```

- For reproducibility set the seed

```
np.random.seed(1828)
```

- This will give you the same sequence of random numbers each time you run your code

Uniform and Gaussian Random Numbers in NumPy

- Uniform random numbers

```
x = np.random.rand(10)
print(x)
array([0.26406512, 0.4857891 , 0.8550659 , 0.54195836, 0.48441921,
       0.54879056, 0.40970468, 0.98964574, 0.05902159, 0.83340854])
```

- Gaussian random numbers now returned in a 2D array

```
# Gaussian random numbers
r=np.random.normal(size=(5,5))
print(r)
[[-1.33171433 -0.74678445  1.13349986 -1.52567032  0.09068109]
 [ 0.77295278  1.15462762  0.14064668 -0.20941348 -0.18446004]
 [-0.25627145  0.59579659 -1.80004184 -0.96032238  0.29522222]
 [ 0.43145469  0.84674349  0.10065509 -0.69472277  0.03555611]
 [ 0.05831341  0.64811795 -0.05200189  1.12076513 -0.71721171]]
```

NumPy Vectorisation

- NumPy's functions allow vectorisation- Python's functions do not
- You pass in a vector of values; you get back a vector of values
- Here is an example using the exponential function

```
x = np.random.rand(10)
y = np.exp(x)
print(y)
[1.2179072  2.66579699 2.50019139 1.63080461 2.20330939 1.16205315
 2.24043983 1.47829626 1.30233741 1.15762022]
```

- This works for all NumPy's mathematical functions
- Not only does this save us from writing a loop, but it is also much faster too than doing this in Python
- Let us see …

Speed Test: Compare Python vs NumPy

- First we generate 10,000 random numbers

```
from math import exp
numElements = 10000
rarray=np.random.rand(numElements)
```

- Function f1 stores 10,000 values of the exponential in array x

```
def f1(rv):
    x=[]
    for r in rarray: x.append(exp(r))
    return x
```

- Timing

```
x=[]
%timeit x=f1(rarray)
100 loops, best of 3: 1.99ms per loop
```

NumPy Vectorized Calculations are fast

- First we generate 10,000 random numbers
- We then store 10,000 values of the exponential in array x

```
def f2(rv):
    x=np.exp(rv)
    return x
```

- We also store 10,000 of the exponential in ndarray x
- The calculation is about 50 times faster

```
%timeit x = f2(rarray)
10000 loops, best of 3: 42 µs per loop
```

- The looping is done in C code inside Numpy – not in Python

Numba provides high-performance but does not win

- It uses a just-in-time (JIT) compiler to convert code to native machine code using – Numba decides what to optimise

```
from numba import njit
@njit
def f3(rv):
    x = []
    for r in rv:
        x.append(exp(r))
    return x
```

```
Basic Python 1990.00000 ms
Numpy Python 42.00000 ms
Numba Python 254.00000 ms
```

- Numba works best when NumPy vectorization is not possible

Introduction to Numpy and Numba Notebook 3

Add your notes here

Libraries: Matplotlib

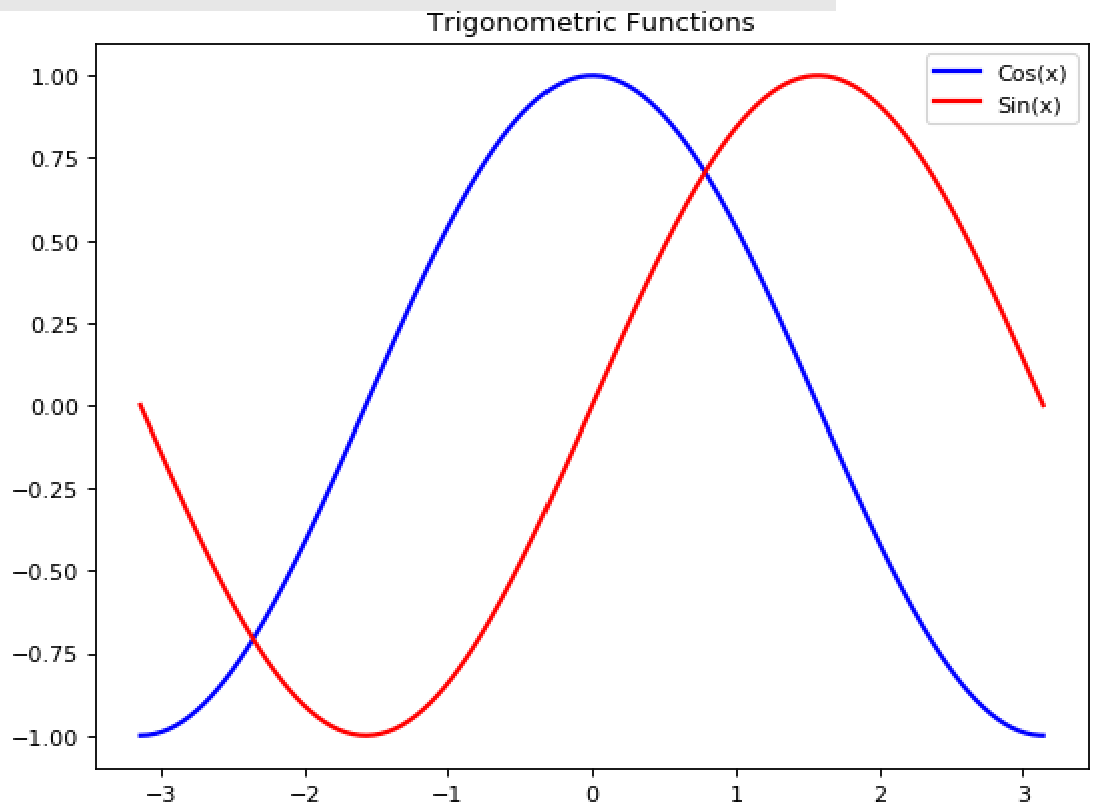
Matplotlib

- Matplotlib is a Python 2D ands 3D plotting library which produces publication quality figures in a variety of hardcopy formats
- It is the most widely used 2D plotting package in Python
- Matplotlib has a subsection called **pyplot**
- This provides a plotting interface that is similar to Matlab
- We load it as follows

```
import matplotlib.pyplot as plt
```

Simple Plots

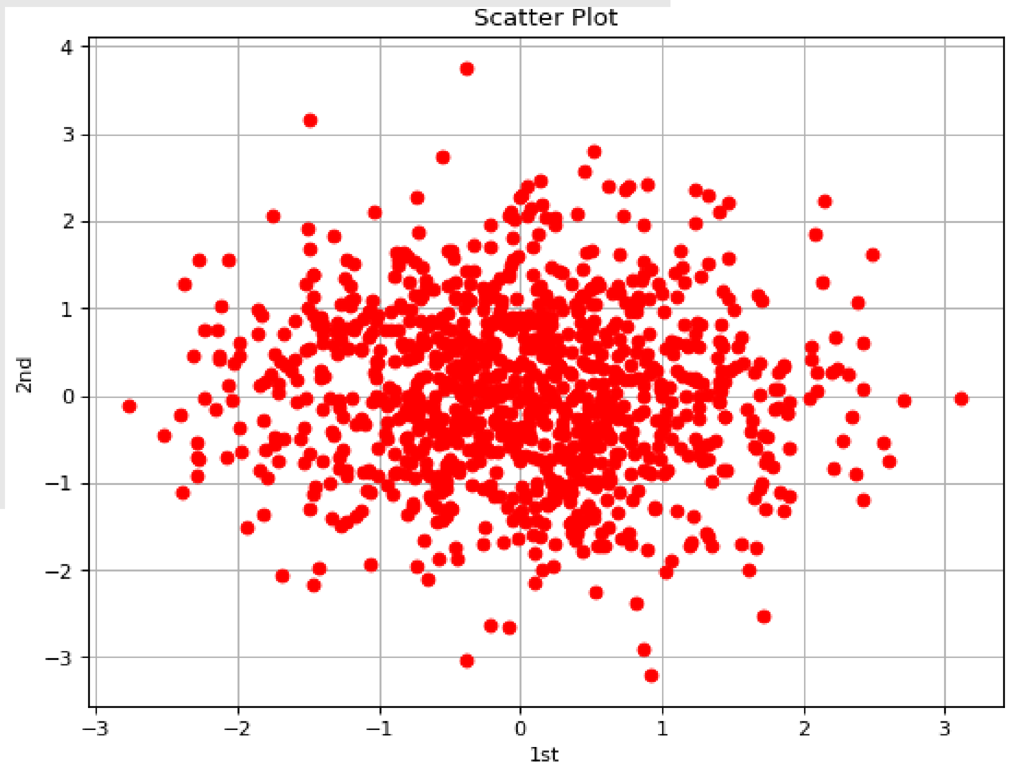
```
x = np.linspace(-np.pi,np.pi,256,endpoint=True)
cx, sx = np.cos(x), np.sin(x)
plt.figure(figsize=(8,6), dpi=80)
plt.plot(x, cx, color='blue', label="Cos(x)", linewidth=2)
plt.plot(x, sx, color='red', label="Sin(x)", linewidth=2)
plt.title("Trigonometric Functions")
plt.legend()
```



- Make sure you understand each line above
- You should be able to do this without looking at the documentation

Scatterplots

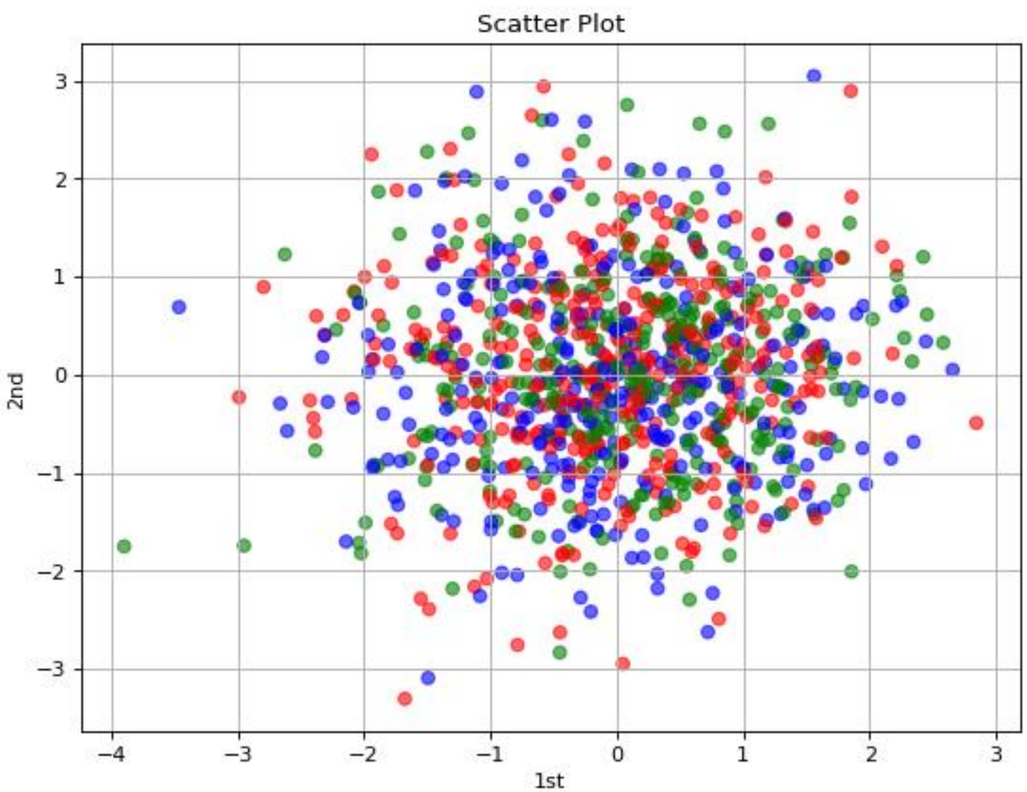
```
y = np.random.standard_normal((1000, 2))
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(y[:, 0], y[:, 1], 'ro')
plt.grid(True)
plt.xlabel('1st')
plt.ylabel('2nd')
plt.title('Scatter Plot')
```



- 'ro' means red dot
- **Colours** can include b=blue, g=green, y=yellow, w=white, m=magenta, k=black
- **Markers** can take many values including .=dot, o=circle,*=star, -=lines, v=triangle down, s=square, h=hexagon, +=plus, ...

Scatterplots with coloured points – use a ColorMap

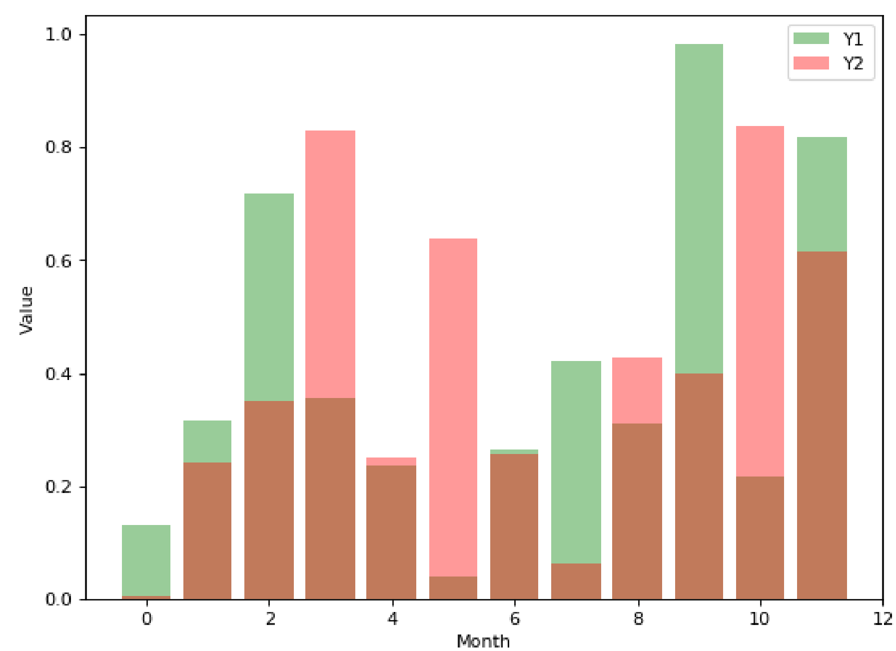
```
from matplotlib import colors
colours = ['red','green','blue']
c = np.random.randint(0, len(colours), len(y))
cmap = colors.ListedColormap(colours)
plt.scatter(y[:, 0], y[:, 1], c=c, marker='o', cmap=cmap, alpha=0.6)
```



Bar Charts

```
n = 12
x = np.arange(n)
y1 = np.random.uniform(0.0,1.0,n)
y2 = np.random.uniform(0.0,1.0,n)
plt.figure(figsize=(8,6), dpi=80)
plt.bar(x, y1, facecolor='green', alpha=0.4, label="Y1")
plt.bar(x, y2, facecolor='red', alpha=0.4, label = "Y2")
plt.xlabel("Month")
plt.ylabel("Value")
plt.legend()
```

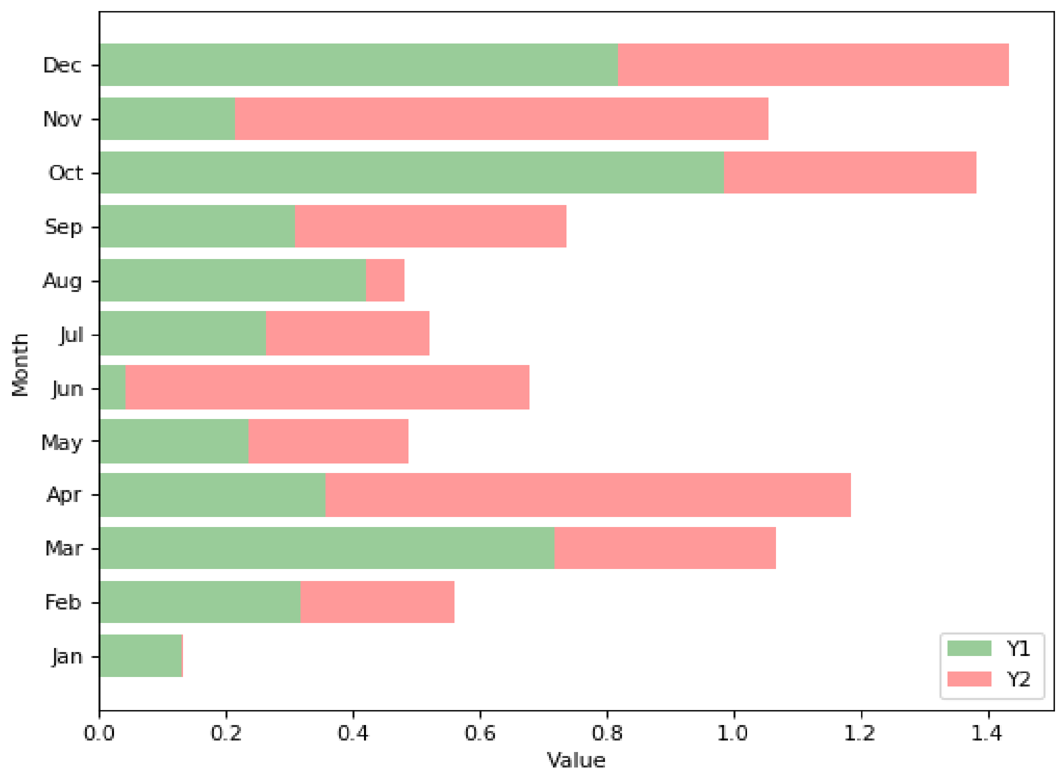
- Alpha is the transparency so that we can see both columns



EDHEC BUSINESS SCHOOL

Horizonal Stacked Bar Charts and Ticks

```
plt.barh(x, y1, facecolor='green', alpha=0.4, label="Y1")
plt.barh(x, y2, left = y1, facecolor='red', alpha=0.4, label = "Y2")
plt.xlabel("Value")
plt.ylabel("Month")
plt.yticks(x, ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))
plt.legend()
```

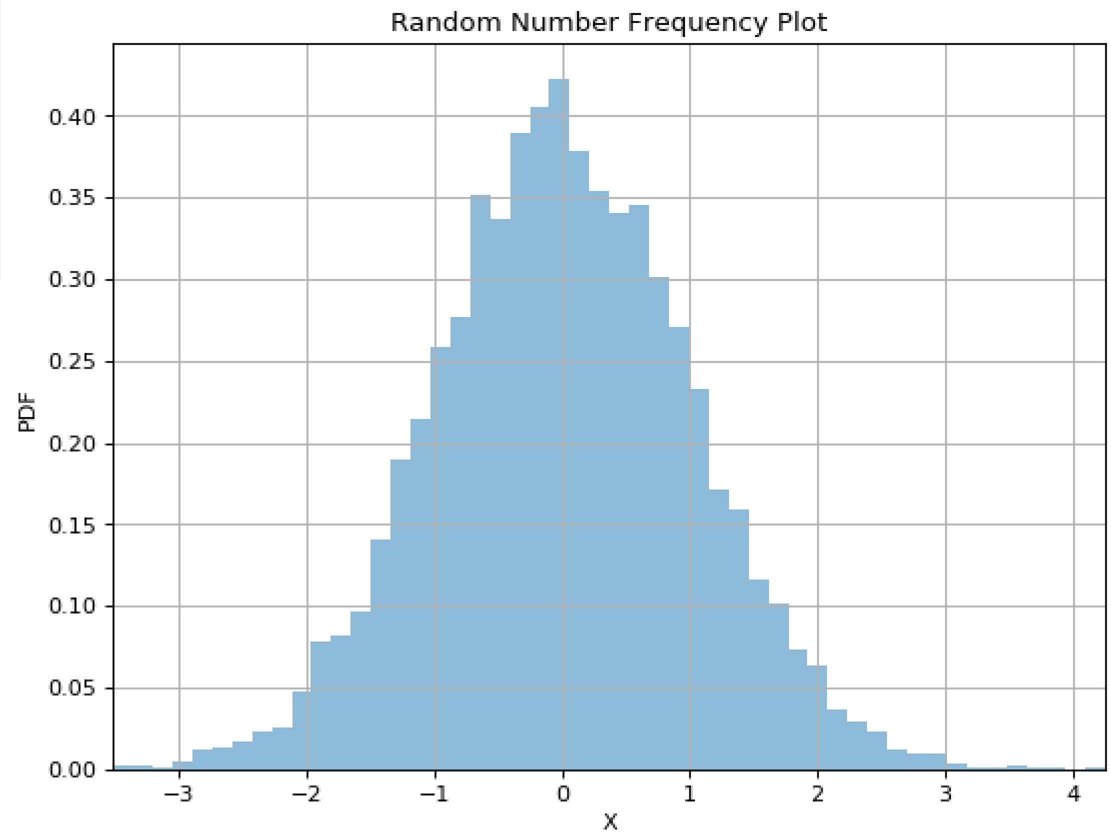


***bin:* 横坐标的分隔数量**

Histograms

```
n = np.random.randn(10000)
plt.figure(figsize=(8,6), dpi=80)
plt.hist(n, bins=20, alpha=0.5, normed=1)
plt.title("Random Number Frequency Plot")
plt.xlim((min(n), max(n)))
plt.xlabel("X")
plt.ylabel("PDF")
plt.grid(True)
```

- **normed=1** converts frequencies to probabilities



Use Styles to make good-looking figures easily

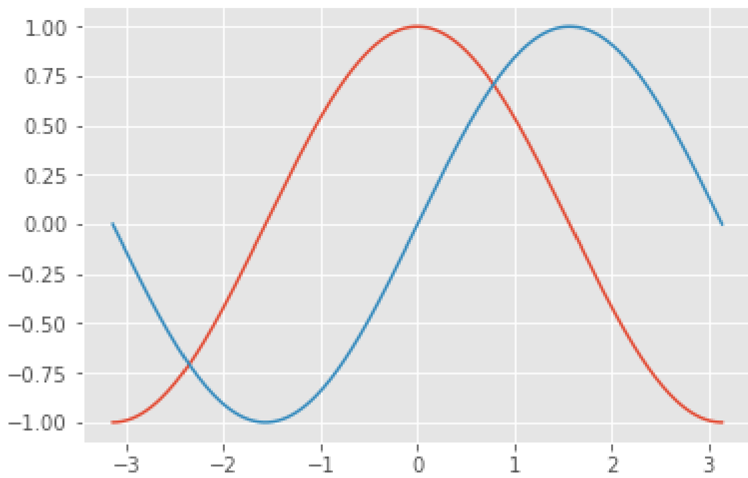
- The whole look of the graphs can be changed using the style

```
plt.style.use('ggplot')
```

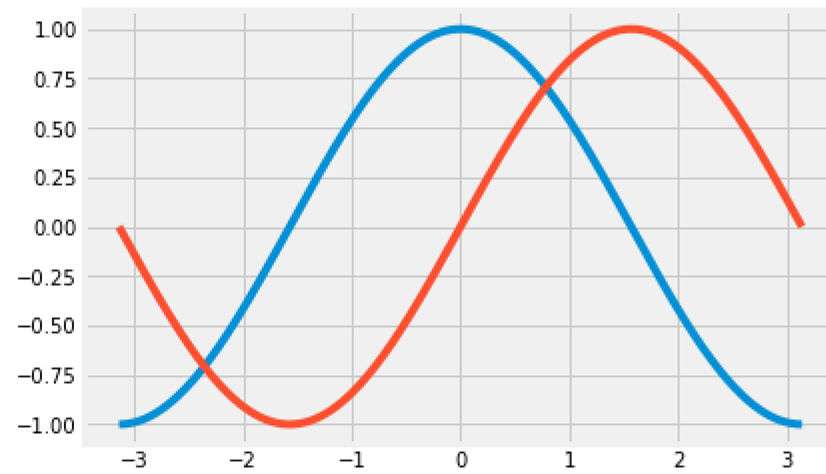
- The style 'ggplot' is similar to the style used in ggplot in R
- This is widely liked
- There are lots of styles – look at documentation of matplotlib
- To recover the default settings you need to use

```
import matplotlib as mpl
mpl.rcParams.update(mpl.rcParamsDefault)
```

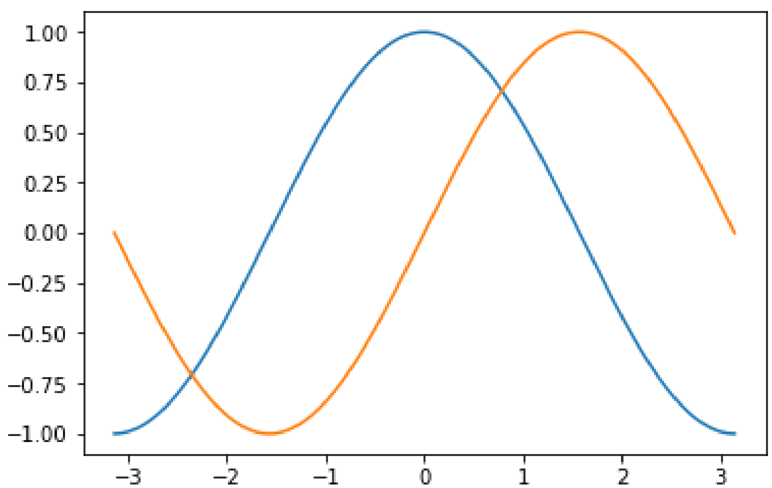
Different Styles for a Graph with no formatting



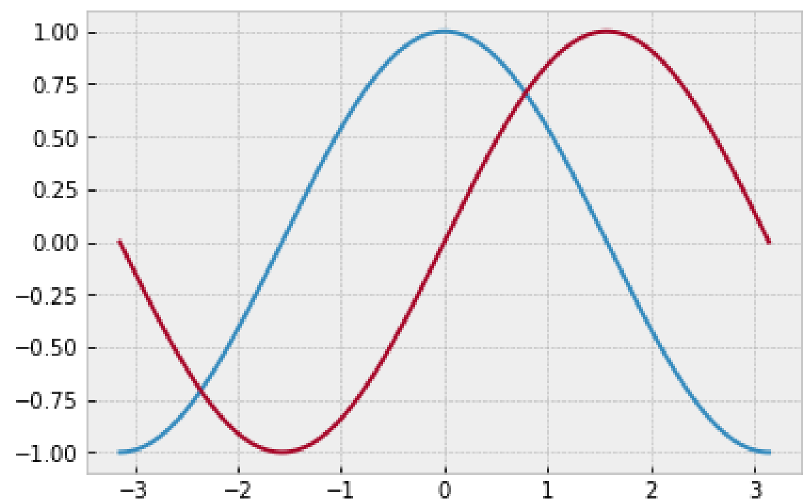
ggplot



fivethirtyeight



default



bmh

Saving Plots to File

- We often need to store the plots in a file
- Matplotlib can handle a range of formats

```
plt.savefig('surfaceplot.png')
plt.savefig('surfaceplot.jpeg')
plt.savefig('surfaceplot.pdf')
```

- If you want better looking statistical plots, try Seaborn
- Seaborn is a separate library that sits on top of matplotlib
- Its purpose is to produce attractive and informative statistical plots in Python and is closely integrated with Numpy and Pandas

Introduction to Matplotlib Notebook 4

Add your notes here

Libraries: Pandas

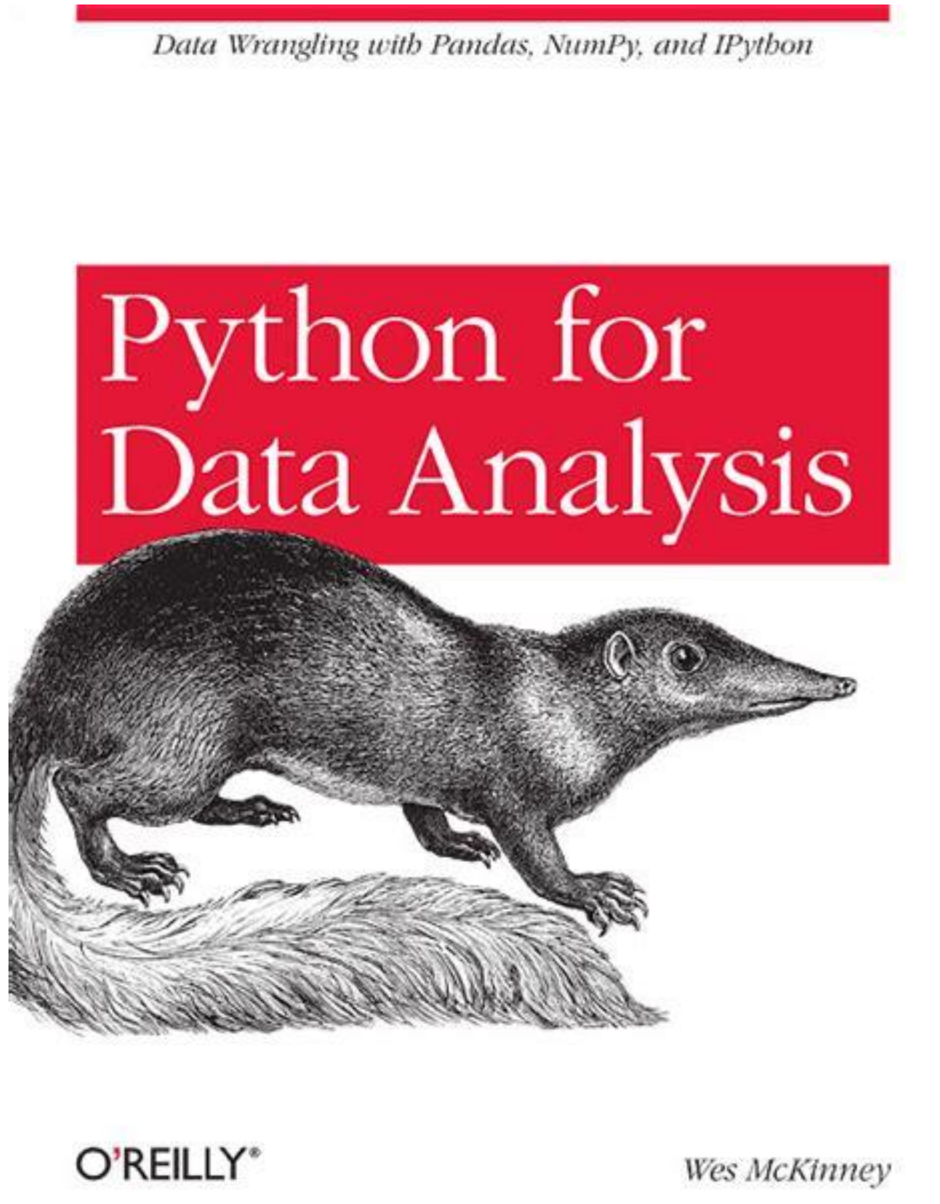
Pandas

- **Pandas = Panel Data** enables you to do efficient data analysis
- Has two key data types:
  - **Series** – A data array with a named index
  - **DataFrame** – A data matrix with labelled index and columns
- It can handle huge data sets with high efficiency
- It has been highly optimised to deal with time series data
- Many standard calculations are built in and so are fast
- **You should use it instead of Excel for any data work**
- It provides SQL like ways to join data and create useful reports
- You should import it as follows:

```
import pandas as pd
```

Pandas Reference

- The book Python for Data Analysis is a good reference
- The author Wes McKinney wrote the Pandas Library



Pandas DataFrame

- The **DataFrame** is the core Pandas object and borrows from R
- It is similar in format to an Excel spreadsheet
- Has tools for easy reading in of data from different sources including csv files, excel sheets, SQL databases and others
- Can do pivot table style operations on the data
- Time series functionality so understands date ranges, lagging, statistical functions
- Can easily slice the data into smaller subsets
- Partly coded in Cython (Python compiled to C) to make it fast

Pandas Series

- A Pandas Series is like a numpy array with a named index
- It has a few inputs – the main ones are **Data** and **Index**
- Here we define, access and plot a series

```
sales = pd.Series(data=[303,391,374, 401], index=['Q1','Q2','Q3', 'Q4'])
```

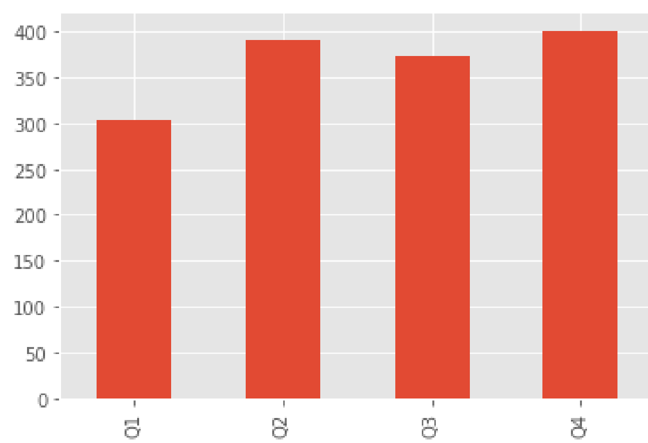
```
sales
```

```
Q1     303
Q2     391
Q3     374
Q4     401
dtype: int64
```

```
sales['Q1']
```

```
303
```

```
plt.style.use('ggplot')
sales.plot.bar();
```



The Pandas DataFrame

- A DataFrame can be created in many ways including this way

```
df = pd.DataFrame([1,3,5,7], columns = ['odd'], index = ['a','b','c','d'])
df
```

	odd
a	1
b	3
c	5
d	7

Create DataFrame with one column

```
df['even'] = [2,4,6,8]
```

```
df
```

	odd	even
a	1	2
b	3	4
c	5	6
d	7	8

Add a new column

```
type(df['odd'])
```

```
pandas.core.series.Series
```

Each column is a Pandas Series

Accessing the Pandas DataFrame

- The first column, the **index,** can be accessed using **df.index**
- The column names can be accessed using **df.columns**
- Access rows by index using **df.loc['a']** or row number **df.iloc[0]**

```
df.loc['a']
```

```
odd     1
even    2
Name: a, dtype: int64
```

```
df.iloc[2]
```

```
odd     5
even    6
Name: c, dtype: int64
```

- We don't do this much but it's worth knowing

Removing Columns

- To remove a column, we call drop

```
df = df.drop(['Currency'], axis=1)
```

- This returns a new data frame without the dropped column
- It does not change df, so we must assign the result to change df

Examining Datasets

- Loading from a CSV file is quite straightforward

```
df = pd.read_csv(“./data/optionPortfolio.csv")
```

- Examining the DataFrame

  df.shape 没有括号

  - **df.shape()** prints the number of rows and columns
  - **df.head(n)** prints the top n rows of the DataFrame
  - **df.tail(n)** prints the bottom n rows of the DataFrame
  - **df.info()** prints all the columns and the non-null values
  - **df.describe()** gets a statistical description of the data

- The last of these is very useful – I use it a lot in ML

Examining the DataFrame using head() and info()

	Trade_ID	TradeDate	Currency	OptionType	Ticker	TradedStockPrice	NumOptions	Strike	ExpiryDate
0	OPT_201313_0	2013-01-03	USD	PUT	CHK	15.865658	490	15.0	2013-12-29
1	OPT_201313_1	2013-01-03	USD	PUT	AAPL	77.442856	280	76.0	2013-12-29
2	OPT_201313_2	2013-01-03	USD	PUT	FB	27.770000	0	26.0	2013-12-29
3	OPT_201313_3	2013-01-03	USD	PUT	AAPL	77.442856	620	74.0	2013-07-02
4	OPT_201318_1	2013-01-08	USD	CALL	AAPL	75.044289	750	75.0	2013-04-08

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4253 entries, 0 to 4252
Data columns (total 9 columns):
Trade_ID            4253 non-null object
TradeDate           4253 non-null object
Currency            4253 non-null object
OptionType          4253 non-null object
Ticker              4253 non-null object
TradedStockPrice    4253 non-null float64
NumOptions          4253 non-null int64
Strike              4253 non-null float64
ExpiryDate          4253 non-null object
dtypes: float64(2), int64(1), object(6)
memory usage: 299.1+ KB
```

Examining an Option Portfolio

- We can get a unique list of tickers

```
df['Ticker'].unique()
array(['CHK', 'AAPL', 'FB', 'MSFT', 'XRX', 'AMZN', 'BA', 'BLCM'],
dtype=object)
```

- We can see how many option trades there are by ticker

```
df['Ticker'].value_counts()
CHK      589
AMZN     574
XRX      566
FB       566
AAPL     562
MSFT     556
BA       556
BLCM     284
Name: Ticker, dtype: int64
```

Apply is used to amend a column

- The trade and expiry dates are non specific objects
- Want to convert them to datetimes
- Need to create a function which takes as input an element of the column and then use the **apply** method

```
def dateConverter(dt):
    dt = dt.replace('-',' ')
    return pd.to_datetime(dt, format='%Y %m %d',dayfirst=True)

df['TradeDate'] = df['TradeDate'].apply(dateConverter)
df['ExpiryDate'] = df['ExpiryDate'].apply(dateConverter)
```

- This is a very powerful way to do complex operations simply

Conditionals on the Rows

- We can select subsets of the data using Boolean filters
- We select all Put options using "==" comparison not assignment

```
df['OptionType'] == "PUT"
```

```
0        True
1        True
2       False
3        True
4        True
        ...
4193    False
4194     True
4195     True
4196     True
4197     True
Name: OptionType, Length: 4198, dtype: bool
```

- This returns an array of Boolean Trues and False values
- We can then use this to select the True valued rows

Filters are very powerful

- We can select subsets of the data using Boolean filters
- Here we select all of the Put options

```
df[df['OptionType'] == "PUT"].head()
```

- Here we select PUTS on Apple – use the & operator for AND

```
df[(df['Ticker'] == "AAPL") & (df['OptionType'] == "PUT")].head()
```

- Here we use a condition requiring Puts where the expiry date has to be before the 2 October 2013

```
import datetime as dt
df[(df['ExpiryDate'] < dt.datetime(2013,10,2)) & (df['OptionType'] ==
"PUT")]
```

Group By

- This enables us to do a breakdown by a specific field
- We want to count the number of options by Ticker

```
df[['NumOptions','Ticker']].groupby(['Ticker']).count()
```

- We want to calculate the average strike by Ticker

```
df[['Strike','Ticker']].groupby(['Ticker']).mean()
```

- See the notebook for examples

Introduction to Pandas Notebook 5

Add your notes here

Libraries: StatsModels

Time Series Data

- I created a file of time series data
- I have a set of individual stock files from the NASDAQ
- I have downloaded them from Kaggle
- Some are in the course project
- I then read them in using Python and created a data frame
- I then save them as a Pickle file - Pickle is a way to convert some Python object to a format where it can be stored in a file
- It is Python specific

Reading in Data

- How to construct the large Pandas timeseries DataFrame
- I use the pandas **read_csv** function to load the stock files

```
df_all = pd.DataFrame()

for ticker in ['ba','dis','ge','hpq','ibm','intc','jnj','jpm','ko',
               'mcd','mo','mrk', 'pg', 'utx', 'xom']:

    filename = ticker + ".us.txt"
    full_filename = ".\\data\\stocks\\" + filename
    df = pd.read_csv(full_filename)
    df["Ticker"] = ticker
    df['Date'] = df['Date'].apply(dateConverter)

    df_all = pd.concat([ df_all, df], axis=0)
```

- I then concatenate each file to the large time series data and save it to a pickle file

```
df_all.to_pickle(".//data//timeSeriesData.pkl")
```

Constructing Time Series Data File Notebook 6

Add your notes here

Loading Financial Time Series Data

- There are several ways of loading data from public sources
- Problem is they break every few years – so I use my own data which was created in Notebook 06

```
df_all = pd.read_pickle('.\\data\\timeSeriesData.pkl')
```

```
df_all.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 184562 entries, 0 to 12073
Data columns (total 8 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   Date     184562 non-null  datetime64[ns]
 1   Open     184562 non-null  float64
 2   High     184562 non-null  float64
 3   Low      184562 non-null  float64
 4   Close    184562 non-null  float64
 5   Volume   184562 non-null  int64
 6   OpenInt  184562 non-null  int64
 7   Ticker   184562 non-null  object
dtypes: datetime64[ns](1), float64(4), int64(2), object(1)
memory usage: 12.7+ MB
```

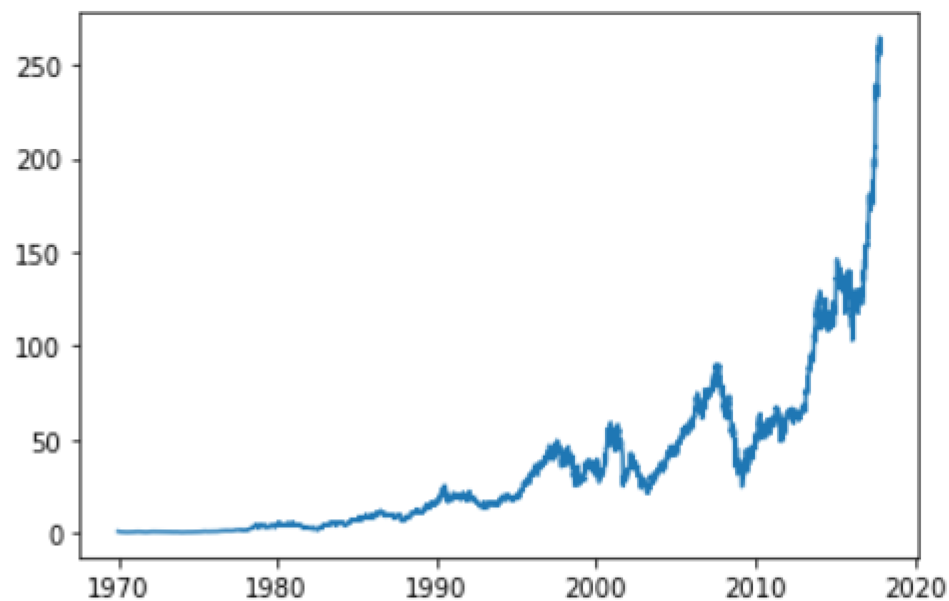
Plotting the Close

- We extract the dates and the close prices

```
df = df_all[df_all.Ticker=="ba"]
dates = df['Date']
timeSeries = df['Close']
```

- Plotting is simple

```
plt.plot(dates, timeSeries);
```



Statsmodels

- Statsmodels is a library that sits on top of NumPy and SciPy
- It contains statistical models that have a similar interface to R
- Includes linear (regression) models of many forms
- Descriptive statistics
- Statistical tests
- Time Series analysis including
  - VAR and SVAR models
  - AR/ARMA Kalman Filter, Macro filters
  - ARCH and GARCH

Linear Regression in Statsmodels

- We have done this in Scikit but sometimes we want to use a dedicated statistics library to do all of our time series analysis
- We generate a noisy linear relationship and fit using OLS

```
numPoints = 20
x = np.linspace(-5, 5, numPoints)
np.random.seed(1)

# normal distributed noise
y = -5 + 3*x + 4 * np.random.normal(size=x.shape)

# Create a data frame containing all the relevant variables
data = pd.DataFrame({'x': x, 'y': y})

from statsmodels.formula.api import ols
model = ols("y ~ x", data).fit()
```

Linear Regression Results

- We get the results of the model fit using

```
print(model.summary())
```

- These are as follows

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.804
Model:                            OLS   Adj. R-squared:                  0.794
Method:                 Least Squares   F-statistic:                     74.03
Date:                Fri, 23 Feb 2018   Prob (F-statistic):           8.56e-08
Time:                        14:48:29   Log-Likelihood:                -57.988
No. Observations:                  20   AIC:                             120.0
Df Residuals:                      18   BIC:                             122.0
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -5.5335      1.036     -5.342      0.000      -7.710      -3.357
x              2.9369      0.341      8.604      0.000       2.220       3.654
==============================================================================
Omnibus:                        0.100   Durbin-Watson:                   2.956
Prob(Omnibus):                  0.951   Jarque-Bera (JB):                0.322
Skew:                          -0.058   Prob(JB):                        0.851
Kurtosis:                       2.390   Cond. No.                         3.03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Time Series

- Load up the stock prices

```
# Function loads historical stock prices
df_all = pd.read_pickle('.\\data\\timeSeriesData.pkl')
```

- We have a time series with 185k row – lots of tickers

```
df_all.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 184562 entries, 0 to 12073
Data columns (total 8 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   Date     184562 non-null  datetime64[ns]
 1   Open     184562 non-null  float64
 2   High     184562 non-null  float64
 3   Low      184562 non-null  float64
 4   Close    184562 non-null  float64
 5   Volume   184562 non-null  int64
 6   OpenInt  184562 non-null  int64
 7   Ticker   184562 non-null  object
dtypes: datetime64[ns](1), float64(4), int64(2), object(1)
memory usage: 12.7+ MB
```

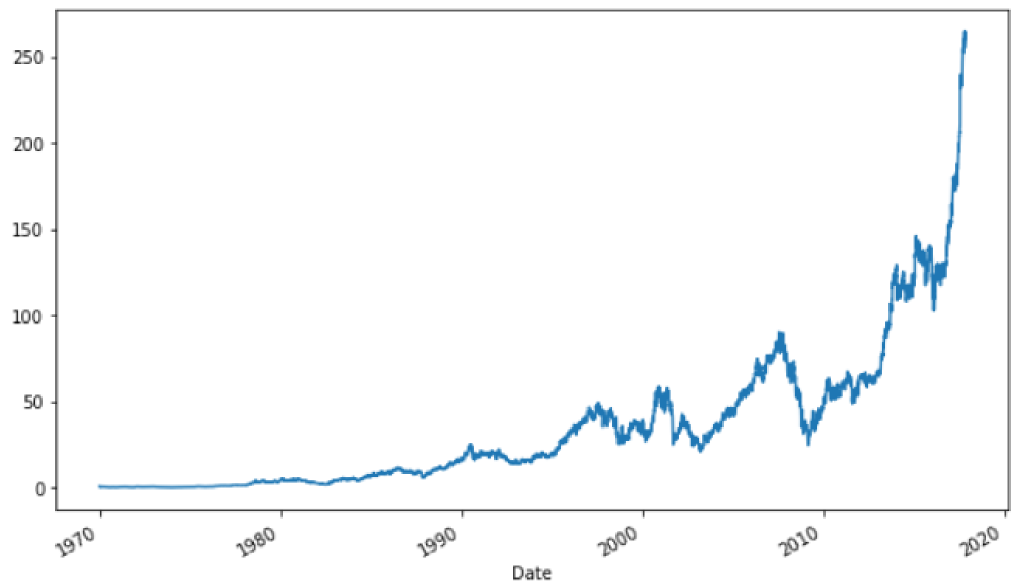
	Date	Open	High	Low	Close	Volume	OpenInt	Ticker
0	1970-01-02	0.7587	0.8092	0.7587	0.8092	753088	0	ba
1	1970-01-05	0.8263	0.8429	0.8263	0.8345	879203	0	ba
2	1970-01-06	0.8429	0.8598	0.8429	0.8429	1607067	0	ba
3	1970-01-07	0.8429	0.8598	0.8429	0.8512	767501	0	ba
4	1970-01-08	0.8512	0.8512	0.8263	0.8429	958476	0	ba

Focus on the Close Price for Boeing (BA)

- Grab all the rows for Boeing and drop all but Close

```
df_ba = df_all[df_all.Ticker=="ba"]
df_ba = df_ba.drop(['Open', 'High', 'Low', 'Volume', 'OpenInt', 'Ticker'], axis=1)
# Make the date the index
df_ba.index = df_ba['Date’]
df_ba['Close'].plot(figsize=(10,6));
```

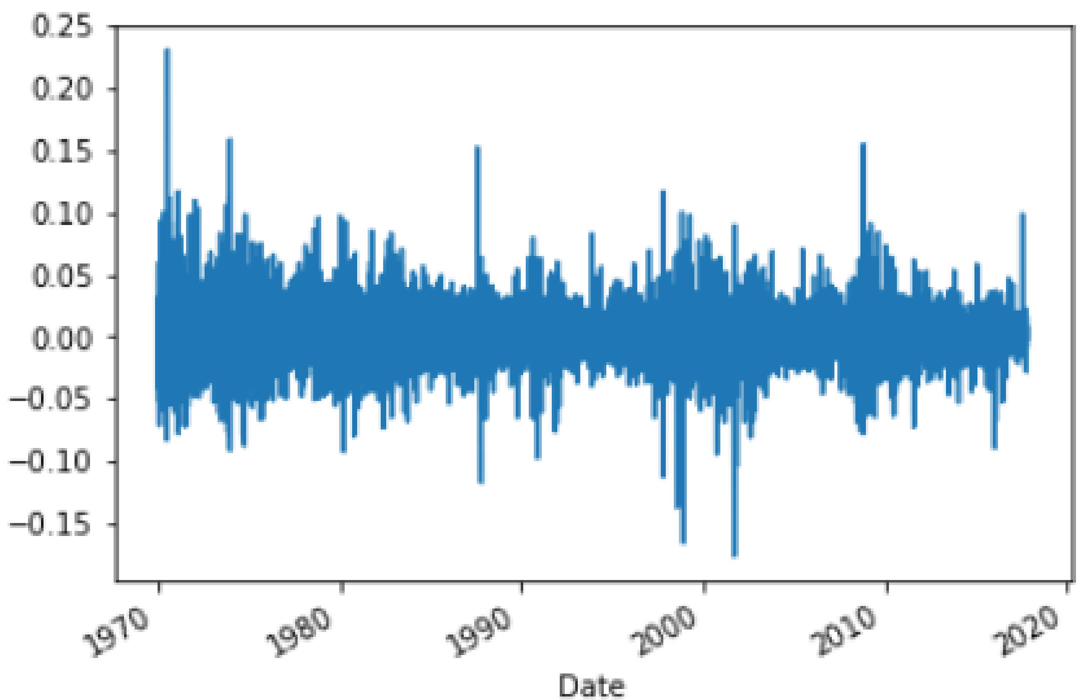
- These are as follows



Percentage Changes

- Want a stationary time series – we take percentage differences using a one-day lag

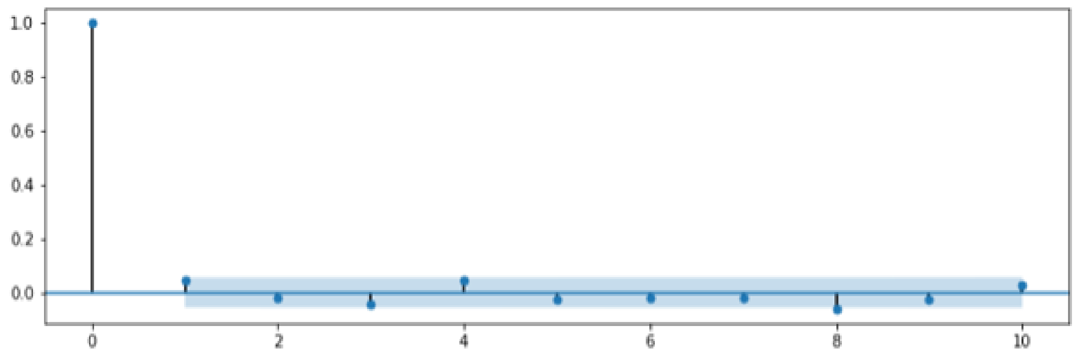
```
df_ba_close = df_ba['Close']
diffs = df_ba_close.pct_change(1)
diffs.plot();
```

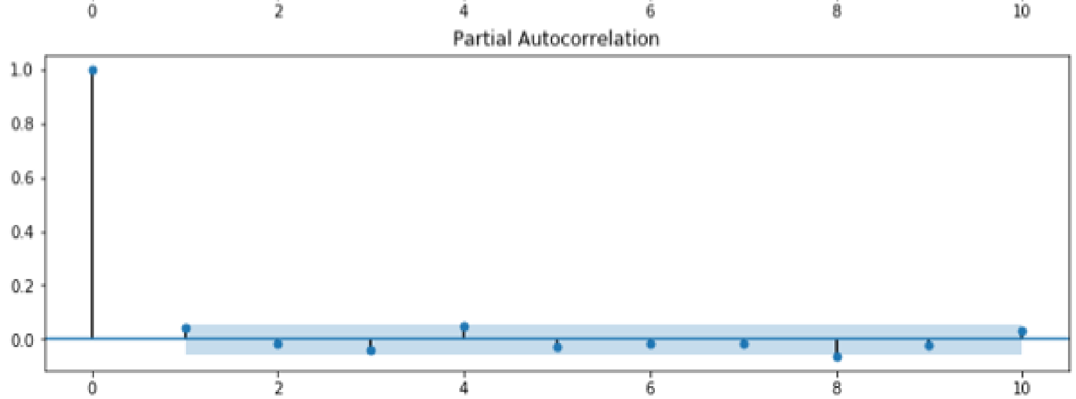


Time Series: Autocorrelation Tests

- We can easily calculate and plot the Autocorrelation and PACF
- Squeeze converts the dataframe to a Pandas Series

```
import statsmodels as sm
s = diffs.values.squeeze()
sm.graphics.tsaplots.plot_acf(s[1:], lags = 5);
sm.graphics.tsaplots.plot_pacf(s[1:], lags = 10);
```





Testing Stationarity using ADF

- The standard test is known as Augmented Dickey-Fuller

```
from statsmodels.tsa.stattools import adfuller
dfTest = adfuller(s[1:])
print('Test Statistic %9.5f' % dfTest[0])
print('p-value  %9.5f' % dfTest[1])
print('Number of Lags Used %9.5f' % dfTest[2])
print('Number of Observations Used',dfTest[3])
for conf in dfTest[4]:
    print('Critical Value at %s: %9.5f' % (conf, dfTest[4][conf]))
```

Test Statistic -79.29879

p-value 0.00000

Number of Lags Used 1.00000

Number of Observations Used 12071

Critical Value at 1%: -3.43089

Critical Value at 5%: -2.86178

Critical Value at 10%: -2.56690

Introduction to StatsModels for Time Series Notebook 7

Add your notes here

Libraries: Scipy

Scipy

- SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering
- It includes a number of modules that may be of interest to us
  - Special functions – scipy.special
  - Integration – scipy.integrate
  - Optimization – scipy.optimize
  - Interpolation – scipy.interpolate
  - Linear Algebra – scipy.linalg
  - Statistics – scipy.stats
- We will discuss just a few of these in this course
- For more information check out https://docs.scipy.org/doc/scipy/reference/tutorial/index.html

Scipy: A Simple One-Dimensional Optimiser

- We define a one-dimensional function with a minimum

```
def f(x):
    return -np.exp(-(x-0.7)**2)
```

- We call the function **minimize_scalar** as follows

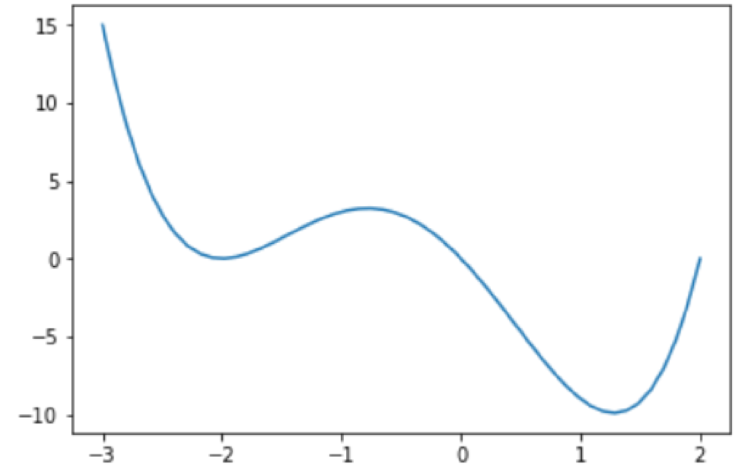
```
from scipy import optimize
result = optimize.minimize_scalar(f)
print(result)
fun: -1.0
nfev: 10
nit: 9 success:
True x: 0.6999999997839409
```

- It allows bounds on the range of solutions, but no other constraints, a choice of algorithms and tolerances

Scipy: Minimising a Non-Convex Function

- We define a function as follows

```
def f(x):
    return (x - 2.0) * x * (x + 2.0)**2
```



- I want the left solution not the global minimum so need to use the bounded method

```
result = optimize.minimize_scalar(f, bounds=(-3,-1), method='bounded')
     fun: 3.2836517984978577e-13
 message: 'Solution found.'
    nfev: 12
  status: 0
 success: True
       x: -2.000000202597239
```

If it has two minimum, we need to use the bounded method. 不然到第一个最低点就停止继续寻找了

Scipy: Multidimensional Optimisation

- I define a multidimensional objective function

```
def fn(x):
    n = len(x); v = 0.0
    for i in range(0,n):
        v = v + (x[i]-i)*(x[i]-i)
    return v
```

- We set the bounds and starting point and call the optimizer

```
bnds = ((0, 5), (0, 5), (0,5), (0,5)); x0 = (0,0,0,0)
res = optimize.minimize(fn, x0, bounds=bnds)
```

- The main return values are the following

```
fun: 1.1868227899471725e-15
success: True
x: array([ 0. , 0.99999999, 2.00000003, 3. ])
```

Scipy: Constrained Multidimensional Optimisation

- I define constraints as a Python dictionary using lambda functions

```
cons = ({'type': 'ineq', 'fun': lambda x:  x[0] - 2 * x[1] + 2},
        {'type': 'ineq', 'fun': lambda x: -x[1] - 2 * x[3] + 6},
        {'type': 'ineq', 'fun': lambda x: -x[2] + 2 * x[3] + 2})
```

- These inequalities are all greater than zero constraints

```
res = optimize.minimize(fun, x0, bounds=bnds, constraints=cons)
```

- The main return values are the following

```
fun: 0.19999999999998
success: True
x: array([ 0 , 0.8, 2.0, 2.6 ])
```

- The constraints have stopped the optimiser from finding the previous solution which had a minimum function value of 0.0

Using Scipy for Optimisation Notebook 8

Add your notes here

Case Study: Option Valuation with Monte Carlo

Black Scholes Analytical

- The valuation of a call option can be programmed easily

```
from math import log, exp, sqrt
from scipy.stats import norm

def priceCallOptionAnalytical(S0,K,T,r,q,sigma):
    d1 = (log(S0/K) + (r - q + 0.5*sigma*sigma)*T) /sigma*sqrt(T)
    d2 = (log(S0/K) + (r - q - 0.5*sigma*sigma)*T) /sigma*sqrt(T)
    value = S0 * exp(-q*T) * norm.cdf(d1,0.0,1.0) - K * exp(-r*T) * norm.cdf(d2,0.0,1.0)
    return value
```

- This is very fast to execute, as we would expect.

```
%timeit priceCallOptionAnalytical(S0,K,T,r,q,sigma)
133 µs per loop
```

Basic Python

- Monte Carlo evaluation can be done easily too

```
import random
def priceCallOptionMC(S0,K,T,r,q,sigma,numPaths):
    payOff = 0.0
    for i in range(0,numPaths):
        z = random.gauss(0.0,1.0)
        S = S0 * exp((r-q-sigma*sigma/2.0) * T + sigma * sqrt(T) * z)
        payOff += max(0,S-K)
    value = payOff * exp(-r*T) / numPaths
    return value
```

- But it is very slow – about 1,000 times slower than analytical

```
%timeit priceCallOptionMC(S0,K,T,r,sigma,numPaths)
98.4 ms per loop
```

Numpy Vectorization

- Using Numpy vectorization works here as we have a simply one-dimensional loop over paths

```
import numpy as np
def priceCallOptionMC_Numpy(S0,K,T,r,q,sigma,numPaths):
    z = np.random.normal(size=numPaths,loc=0.0,scale=1.0)
    S = S0 * np.exp((r-q-sigma*sigma/2.0) * T + sigma * sqrt(T) * z)
    payoff = np.maximum(S-K,0)
    value = np.sum(payoff)/numPaths * np.exp(-r*T)
    return value
```

- This can be memory intensive due to need to hold randoms
- Execution time is about 30 times faster using Numpy

```
%timeit priceCallOptionMC_Numpy(S0,K,T,r,q,sigma,numPaths)
3.7 ms per loop
```

Numba JIT Wins!

- Using Numba we can return to the basic python version

```
from numba import njit
@njit
def priceCallOptionMC_Numba(S0,K,T,r,q,sigma,numPaths):
    payOff = 0.0
    for i in range(0,numPaths):
        z = random.gauss(0.0,1.0)
        S = S0 * exp((r-q-sigma*sigma/2.0) * T + sigma * sqrt(T) * z)
        payOff += max(0,S-K)
    value = payOff * exp(-r*T) / numPaths
    return value
```

- Execution time is even faster than using Numpy - low memory

```
%timeit priceCallOptionMC_Numba(S0,K,T,r,q,sigma,numPaths)
```

2.84 ms per loop

Conclusions

- Using Numba, Python code becomes very fast
- Yet we can retain readability and flexibility over the code which is not possible with Numpy vectorizations
- Applying Numba to Numpy seems to make it slower !
- With Numba, Python becomes C-like in its speed
- And it is 10-100 times faster than VBA

Option Pricing Using Monte Carlo Notebook 9

Add your notes here

Case Studies

Bond Yield Curves

Fitting and Interpolation

Generate Cashflow Times

- # of payments left is **maturity** x **frequency** rounded down
- The first payment time is then the maturity minus all full periods
- We use Numpy's linspace to generate the times

```
def flowTimes(maturity, frequency):
    small = 1e-10
    numPaymentsMinusOne = int(maturity * frequency-small)
    firstPayment = maturity - numPaymentsMinusOne / frequency
    return np.linspace(firstPayment,maturity,numPaymentsMinusOne+1)
```

- Some examples make it clear

```
print(flowTimes(2.75,2))
[ 0.25 0.75 1.25 1.75 2.25 2.75]
print(flowTimes(10.1,2))
[ 0.1 0.6 1.1 1.6 2.1 2.6 3.1 3.6 4.1 4.6 5.1 5.6 6.1 6.6 7.1 7.6 8.1 8.6 9.1
9.6 10.1]
```

Calculate Full Bond Price from Yield

- It is then straightforward to calculate the full price of a bond

```
def bondFullPriceFromYield(y,maturity,coupon,frequency):
    paymentTimes = flowTimes(maturity,frequency)
    price = 0.0; df = 1.0
    for t in paymentTimes:
        df = 1.0/(1.0 + y/frequency)**(t*frequency)
        price += ( coupon / frequency ) * df
    price += df
    return price
```

- We also need to calculate accrued interest

```
def accruedInterest(maturity,coupon,frequency):
    paymentTimes = flowTimes(maturity,frequency)
    accruedPeriod = 1.0/frequency - paymentTimes[0]
    return accruedPeriod * coupon
```

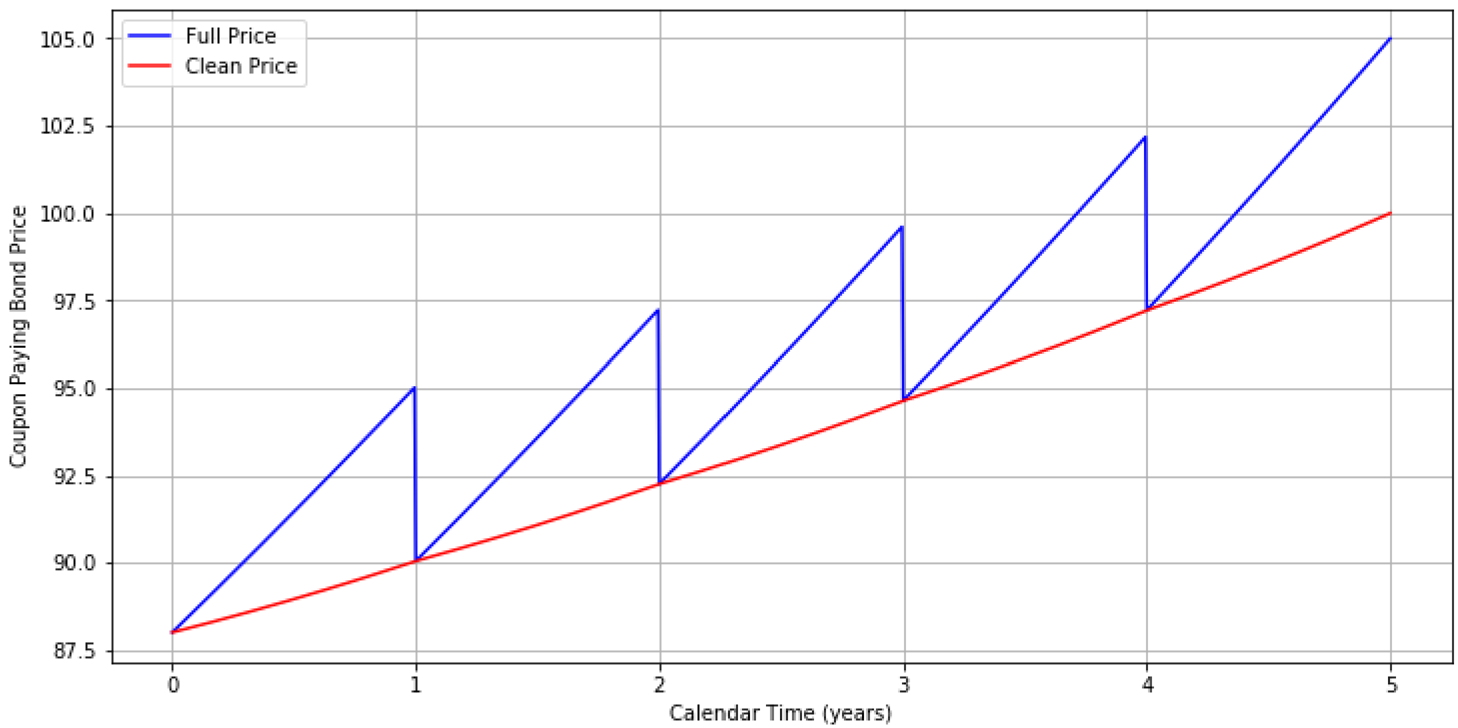
Bond Price Action

- The price action of a bond through time assuming a constant yield is then easy to generate

```
def plotFullPriceAction(y, maturity, coupon, frequency):
    calendarTimes = np.linspace(0.0,maturity,1001)
    fullPrices = []
    cleanPrices = []
    for t in calendarTimes:
        yearsToMaturity = maturity - t
        fullPrice = bondFullPriceFromYield(y, yearsToMaturity, coupon, frequency)
        accrued = accruedInterest(yearsToMaturity, coupon, frequency)
        cleanPrice = fullPrice - accrued

… plotting code …
```

Bond Price Action

- Setting the maturity to 5 years, coupon of 5%, annual frequency and a yield of 8% we have the following price action
- We see the clean price and the full price



- Plotting such graphs is easy in Python Jupyter notebooks

Loading Bond Data

- I have loaded a dataset of US Gilts with prices from 19 Sep 2012
- It is tab separated so use sep = '\t' to load into a dataframe

```
bondDf = pd.read_csv('./data/giltbondprices.txt',sep='\t')
```

	epic	description	coupon	maturity	bid	ask	change	income yield	gross redemption yield
0	TR13	Uk Gilt Treasury Stk	4.50	07-Mar-13	101.92	102.07	-0.01	4.41	0.22
1	T813	Uk Gilt Treasury Stk	8.00	27-Sep-13	107.86	107.98	-0.03	7.41	0.23
2	TR14	Uk Gilt Treasury Stk	2.25	07-Mar-14	102.90	103.05	0.01	2.18	0.22
3	T514	Uk Gilt Treasury Stk	5.00	07-Sep-14	109.28	109.43	0.02	4.57	0.23
4	TR15	Uk Gilt Treasury Stk	2.75	22-Jan-15	105.57	105.68	0.05	2.60	0.33

- I then added a new column with the mid price

```
bondDf['mid'] = 0.5*(bondDf['bid'] + bondDf['ask'])
```

- Note that these are clean prices as this is market convention and so we need to add on accrued to use them in our bond math

Calculating the Yield Curve

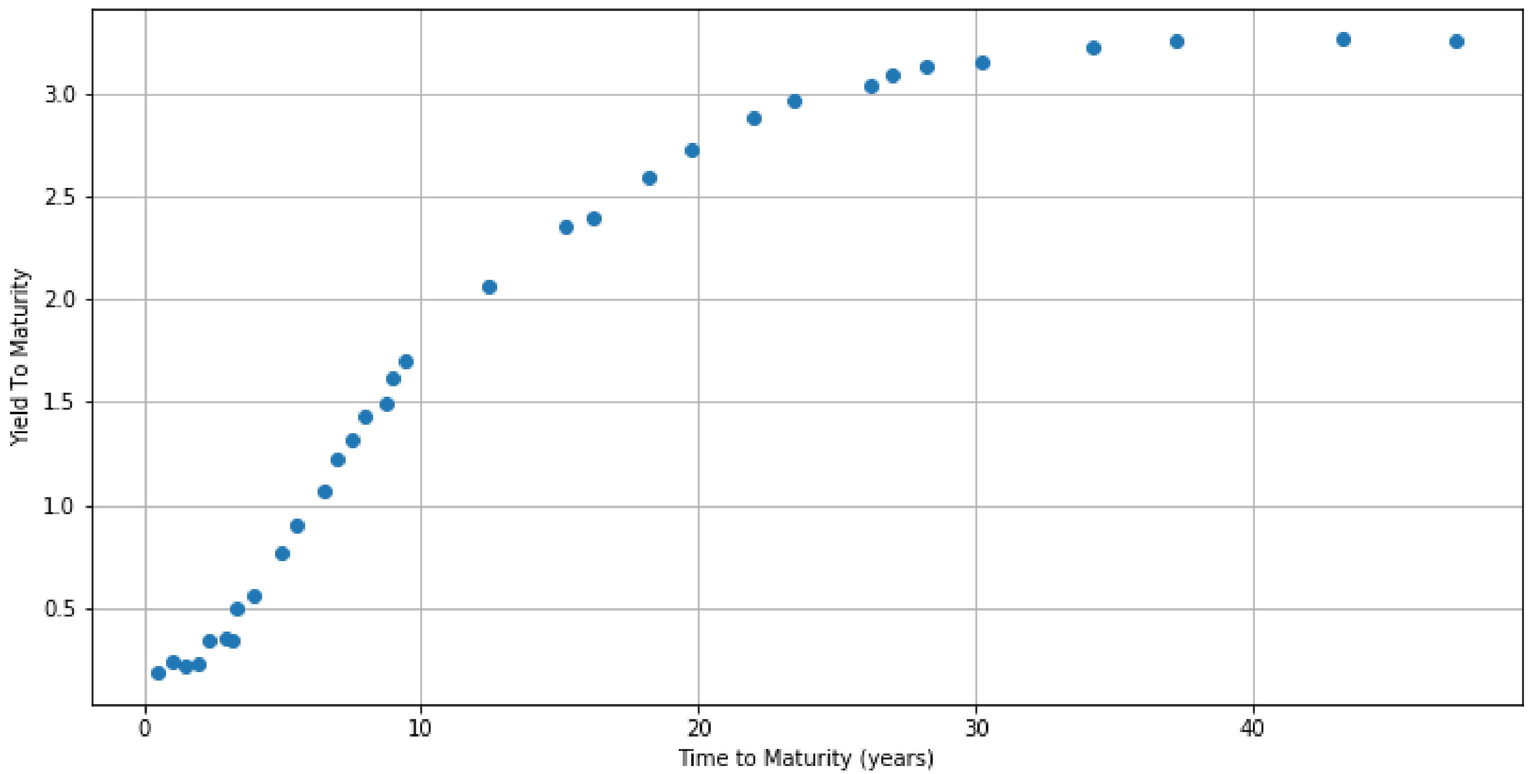
- Want to calculate the yield to maturity using the full price
- I use a lambda function inside the scipy optimize.newton function

```
def bondFullPriceToYield(fullPrice,maturity,coupon,frequency):

    paymentTimes = flowTimes(maturity,frequency)

    ytm_func = lambda y: \
        sum([(coupon/frequency)/(1.0+y/frequency)**(frequency*pmtTime) for pmtTime
in paymentTimes ]) + \
        1.0/(1.0+y/frequency)**(frequency*paymentTimes[-1]) – fullPrice

    initial_guess = 0.05
    return optimize.newton(ytm_func, initial_guess)
```

- This is pushing the lambda function to its limits
- We will see how to handle more complex functions later

The Yield Curve Points

```
plt.figure(figsize=(12, 6))
plt.plot(bondDf['yearsToMaturity'], bondDf['ytm'], 'o')
plt.grid(True)
plt.xlabel('Time to Maturity (years)')
plt.ylabel('Yield To Maturity')
```

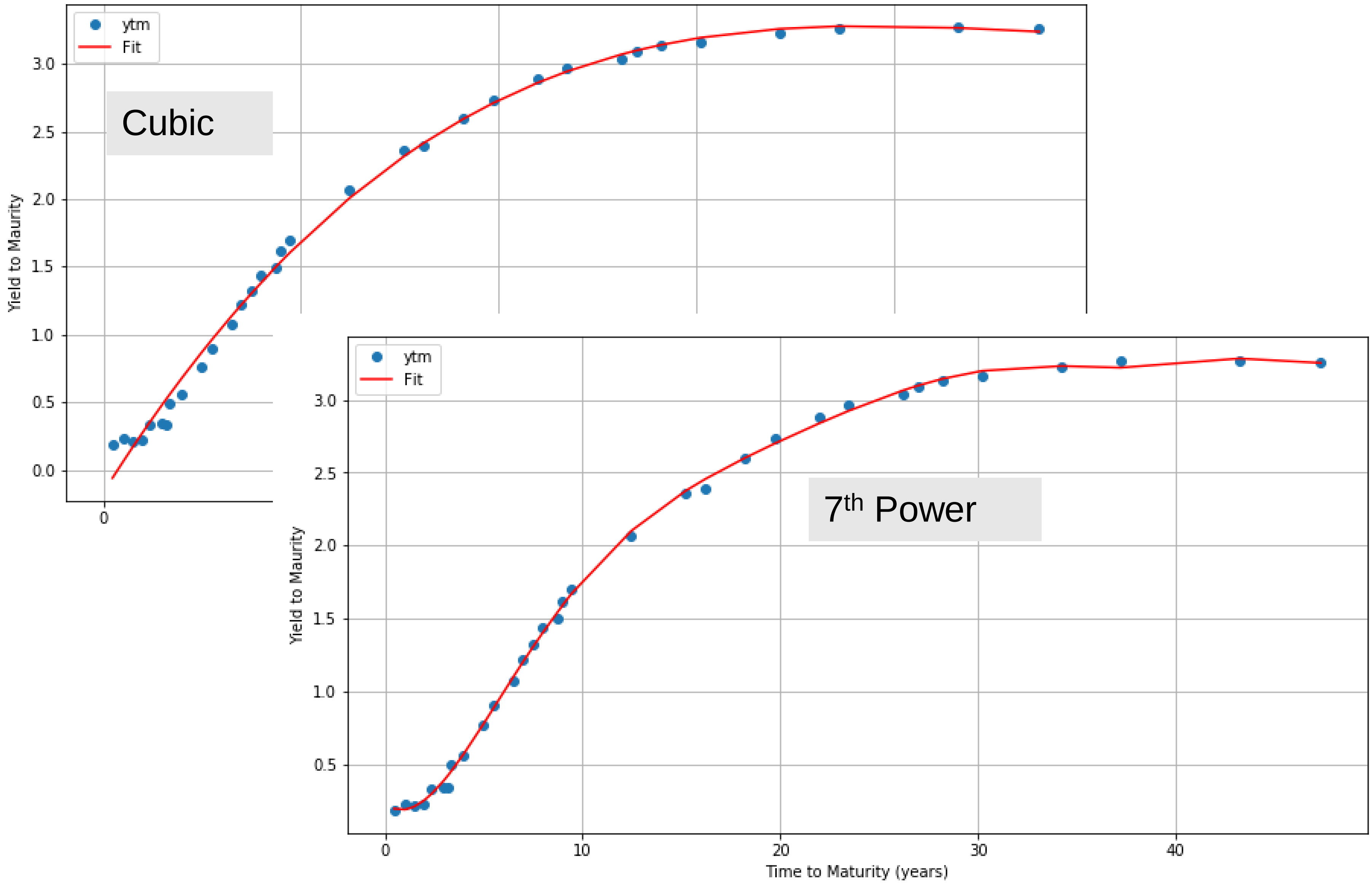


Fit and Plot

- It is not hard to fit a polynomial function to the curves
- I wrote a function to do this which is shown below

```
def fitAndPlot(x,f,d):
#  x is the vector of years and f is the vector of yields
    coeffs = np.polyfit(x, f, deg=d)
    ry = np.polyval(coeffs, x)

    plt.figure(figsize=(12, 8))
    plt.plot(x, f, 'o')
    plt.plot(x, ry, 'r', label='Fit')
    plt.legend(loc=0)
    plt.grid(True)
    plt.xlabel('Time to Maturity (years)')
    plt.ylabel('Yield to Maurity')
```

Fitting Using Polynomials



Fitting Bond Yield Curves Notebook 10

Add your notes here

Case Studies

Mean-Variance Portfolio Optimisation

The Theory

- We have the time series of returns for a universe of N assets
- We wish to determine the optimal allocation on the basis of a mean-variance criteria
- The portfolio return is estimated on T historical daily returns

$$\mu_P = \sum_{i=1}^{N} w_i \mu_i = 252 \times \sum_{i=1}^{N} w_i \sum_{t=1}^{T} r_{it}$$

- The portfolio variance is given by

$$\sigma_P^2 = \sum_{i,j=1}^{N} w_i w_j \sigma_{ij}^2$$

- The σ_{ij}^2 is the covariance of historical returns between i and j

The Stock Price Data

- I loaded equity prices stored in single ticker format

```
timeSeriesData = pd.read_pickle('.\\data\\timeSeriesData.pkl')
```

- The format of this dataframe is

	Date	Open	High	Low	Close	Volume	OpenInt	Ticker
0	1970-01-02	0.7587	0.8092	0.7587	0.8092	753088	0	ba
1	1970-01-05	0.8263	0.8429	0.8263	0.8345	879203	0	ba
2	1970-01-06	0.8429	0.8598	0.8429	0.8429	1607067	0	ba
3	1970-01-07	0.8429	0.8598	0.8429	0.8512	767501	0	ba
4	1970-01-08	0.8512	0.8512	0.8263	0.8429	958476	0	ba

- I want to have the tickers as columns and just examine the close price and have the date as the index

Aligning the Time Series

- The code is a bit complicated at first sight

```
df_all = pd.DataFrame()
for ticker in tickers:
    df_ticker = closePrices[closePrices.Ticker == ticker]
    df_ticker = df_ticker.set_index('Date')
    df_ticker.columns = [['Ticker', ticker]]
    df_ticker = df_ticker.drop(['Ticker'], axis=1)
    df_all = pd.concat([df_ticker, df_all], axis=1, join ="outer")
    df_all = df_all.dropna()
```

- The new dataframe looks like this - which is what we want

	xom	utx	pg	mrk	mo	mcd	ko	jpm	jnj
Date									
1972-01-07	1.8452	0.31680	1.9119	0.7077	0.04379	0.7691	0.9870	2.9672	0.9929
1972-01-14	1.8132	0.32548	1.9523	0.6998	0.04379	0.7448	0.9870	3.0419	0.9685
1972-01-21	1.8452	0.33400	1.9927	0.6840	0.04379	0.7530	0.9747	3.0088	0.9767

Extracting the Returns

- We have closing prices but need returns

```
returns = df_all.pct_change(periods = 1)
returns.dropna(inplace=True)
returns.head()
```

- Using the pct_change function I calculated the daily returns
- I dropped any NA values and filtered out a list of tickers
- Passing this into the dataframe selected just those asset return
- I calculated the average returns, covariance and correlations

```
assetReturns = newReturns.mean()
assetCovariance = newReturns.cov()
assetCorrelations = newReturns.corr()
```

Portfolio Measures of Risk and Return

- In the Jupyter notebook all variables are in memory scope and do not need to be input - we only explicitly pass the weight vector

```
def portfolioVolatility(weights):
    return np.sqrt(np.dot(weights.T, np.dot(assetCovariance * 252, weights)))

def portfolioReturn(weights):
    return np.sum(assetReturns * weights) * 252

def portfolioSharpeRatio(weights):
    return (portfolioReturn(weights) - rfr) / portfolioVolatility(weights)
```

- If we were to write this as standalone python code we would need to pass in the **assetReturns** and **assetCovariance** and **rfr**

Generating the Efficient Frontier

- We iterate over returns from the lowest to the highest
- For each we find the portfolio with lowest variance or volatility

```
minRet = min(assetReturns*252)
maxRet = max(assetReturns*252)
trets = np.linspace(minRet, maxRet, 50)
```

- We have fifty steps in our loop over the different return values
- We store results of the optimiser in an array so we can plot them

```
for tret in trets:
   …
   tvols.append(res['fun'])
```

- Each return value has to become a constraint of the optimiser

Constrained Optimisation

- We now have three constraints – each weight has to be in range 0-100% (no short selling), we are fully invested and a fixed return
- The weights constraint is set by simple bounds on each variable

```
bnds = tuple((0, 1) for x in weights)
```

- The investment constraint states that sum of allocations is 100%
- The return constraint sets the average portfolio return
- These take the form of a tuple of dictionaries

```
cons = ({'type': 'eq', 'fun': lambda x:  portfolioReturn(x) - tret},
         {'type': 'eq', 'fun': lambda x:  np.sum(x) - 1})
```

- We use lambda functions for the constraints - the optimiser looks for the value of *x* that sets the function value to zero

Constrained Optimisation

- We pass these to the optimizer as follows

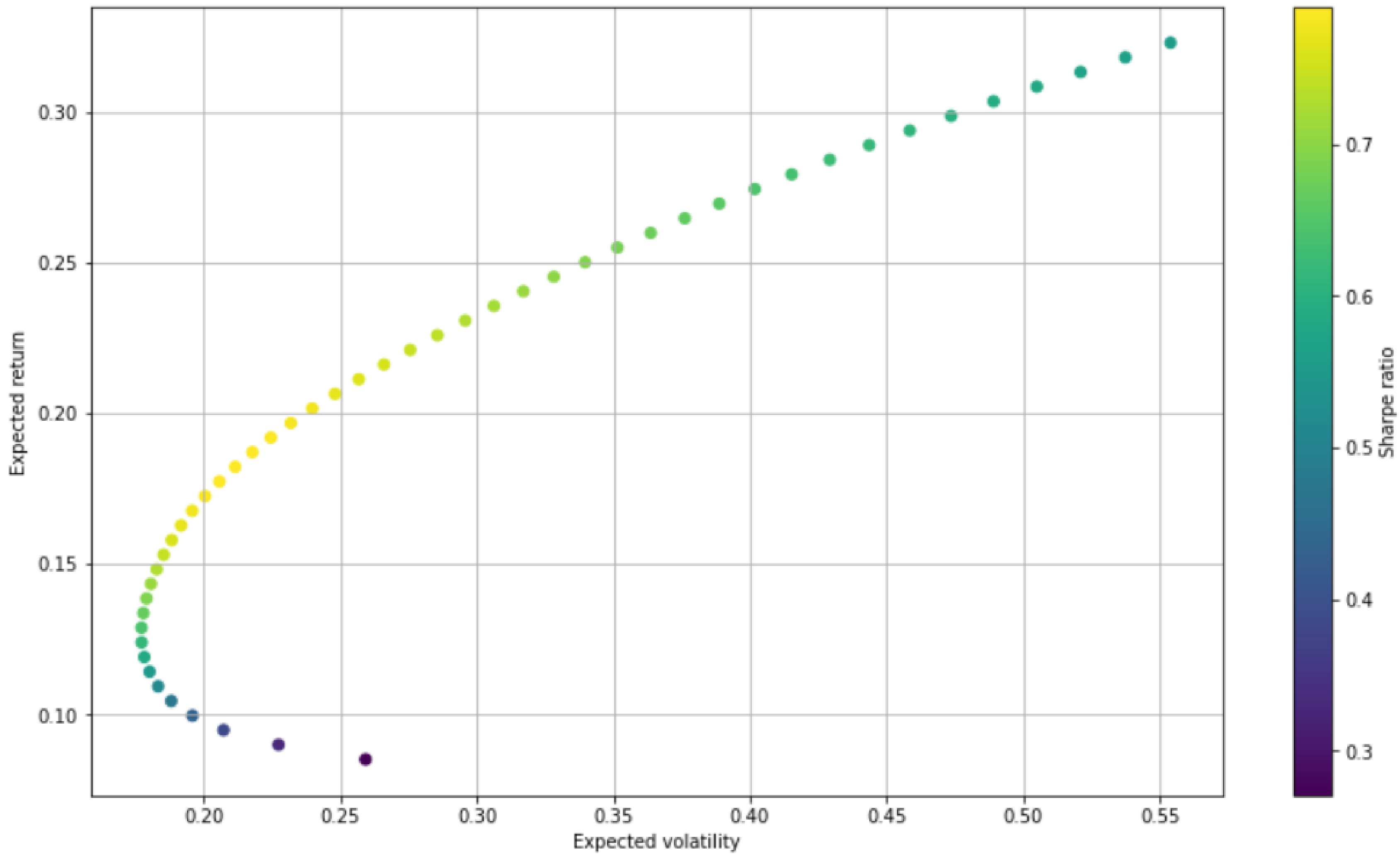
```
res = sco.minimize(portfolioVolatility, initialWeights, method='SLSQP', bounds=bnds, constraints=cons)
```

- I use the Sequential Least Squares Programming (**SLSQP**) method
- I plot the results as a scatterplot with a grid - easier to read
- I set the color of the dots to be the Sharpe Ratio
- A colorbar indicates the value of the Sharpe Ratio

```
plt.figure(figsize=(14, 8))
plt.scatter(tvols, trets, c=(trets-rfr) / tvols, marker='o')
plt.grid(True)
plt.xlabel('Expected volatility')
plt.ylabel('Expected return')
plt.colorbar(label='Sharpe ratio')
```

Efficient Frontier

- And we get the following plot



Mean Variance Portfolio Optimization Notebook 11

Add your notes here

Python Modules

Python Modules

- Jupyter notebooks are great if we are experimenting
- Or if we want to share ideas and teach people how to code
- In practice we want to use Python to automate processes
- We need it to do large complex calculations
- We want it to interface to other stages in some process
- We need it to pull in other code we have already written
- For example it needs to generate automatic daily risk management reports that get emailed to all of the traders
- At this point we need a body of code that works
- We develop in Python modules – a **module** is a .py file that contains Python code

Integrated Development Environment

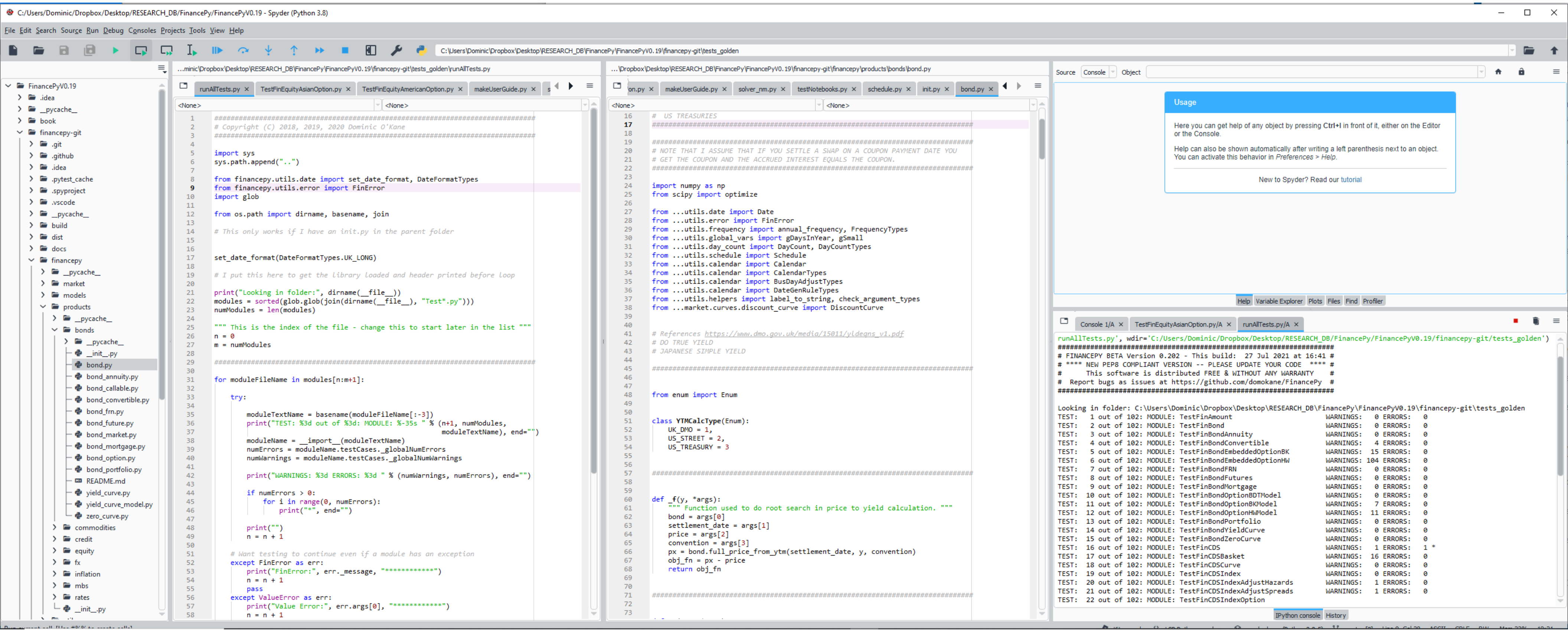
- We develop modules in an **Integrated Development Environment** (IDE)
- This is a graphical user interface that combines an editor with a console for running plus lots of other useful tools
  - Editing multiple files
  - Code syntax checking
  - Debugger
  - Variable watcher
  - Console window
- There are several Python IDEs
- I prefer Spyder – it comes with your Anaconda installation.

To Run Spyder

- In Windows, hit the Windows Key
- Choose the Anaconda menu
- Then you will see Spyder
- Click on it

Here is my FinancePy Project in Spyder

- Multiple editors, file manager, command window, debugger …



Executing a Python File in Spyder

- Python code is usually stored in a module of form **filename.py**
- Load this file so that you can see its contents
- This file can contain classes, functions and commands
- Click on the file so that the text is visible
- Press the play button to execute the file
- Running the file in Spyder will execute any code that is left-aligned – functions will be created but not executed
- The output will be displayed in the Command window

Structure of a Python Module File

- Python code is stored in a module file of form **filename.py**
- This module can contain classes, functions and commands
- Running the file will load functions and classes into memory and will execute any code that is not inside a class or function

Simple Self-Contained MC Option module

- We import dependencies, define a function and call it.
- This module is called BlackScholesMC_1.py

```
import numpy as np

def blackScholesMonteCarlo(numPaths,s0,k,T,r,sigma):
    payoff = 0.0
    for i in range(1,numPaths):
        z = np.random.normal(0.0,1.0)
        sT = s0 * np.exp((r-0.5*sigma**2) * T + sigma * np.sqrt(T) * z)
        payoff += max(sT-k,0)
    value = np.exp(-r*T) * payoff/numPaths
    return value

numPaths = 50000; s0 = 100.0; k = 100.0; T=1.0; r=0.05; sigma = 0.20
price = blackScholesMonteCarlo(numPaths,s0,k,T,r,sigma)
print(price)
10.427788307249811
```

We need Numpy

Define a function

Call the function and print values

You can import your functions from other modules

- You can access the functions you have written in other modules
- You just need to import the other module

```
import filename as fn
```

- You can then access the function using

```
fn.functioname(x)
```

- Or you can write

```
from filename import functionname
```

- And simply call it as

```
functioname(x)
```

We can isolate Functions so they can be reused

- I want to call this function from another module
- I create a new module called callBlackScholesMC_1.py

```
from BlackScholesMC_1 import blackScholesMonteCarlo

numPaths = 50000; s0 = 100.0; k = 100.0
T=1.0; r=0.05; sigma = 0.20
price = blackScholesMonteCarlo(numPaths,s0,k,T,r,sigma)
print(price)
10.520588738511313
10.501508113310235
```

I import my function from the module I just created

Call function and print result

- It runs twice – as it's random I get two slightly different results
- Why did it run twice ?
- Because when BlackScholesMC_1 is imported it runs that module

Importing a Module from another Programme

- Sometimes we want to include functionality, e.g. unit tests, in the same module as the function
- If we run the module then run the tests
- But I do not want these tests to run when I import the module
- To achieve this I use the following trick

```
if __name__ == '__main__':
    INDENTED TEST CODE HERE
```

- If the __name__ variable which is internal to Python equals "__main__" then the file is being run directly, not imported
- This ensures the INDENTED CODE will not be run if the file is imported, it will only run if we run the module directly

We can isolate Functions so they can be reused

- I want to call this function from another module BlackScholesMC_2 with this conditional around the test code

```
from BlackScholesMC_2 import blackScholesMonteCarlo

numPaths = 50000; s0 = 100.0; k = 100.0
T=1.0; r=0.05; sigma = 0.20
price = blackScholesMonteCarlo(numPaths,s0,k,T,r,sigma)
print(price)
10.520588738511313
```

I import my function from the module I just created

Call function and print result

- It runs once !
- This is very powerful – now you can start reusing functions

Object Oriented Python

Object Oriented Programming

- OO was one of the big revolutions in programming languages
- The idea is to unify functionality and data into a single entity
- This entity is called a Class
- The functions are called Class Methods
- The data are called Class Members
- A specific use of a class is called objects
- **Example**: Class Humans
  - Class Methods: Walk, Talk, Eat, Drink, Sleep, …
  - Class members: Sex, Height, Weight, Hair colour, …
  - Object: Me, You

Object Oriented Programming

- OO is different from functional programming
- OO languages usually provide the following:
  - Data Encapsulation
  - Data Abstraction
  - Inheritance
  - Modularity
  - Polymorphism
- I will not explain all of these, but they are all important
- It is a revolutionary new way for designing your code
- Code becomes easier to write, easier to organise and re-usable

Creating a Python Classes

- By convention we ALWAYS start a class name with a capital letter
- Here we define a class called **Circle**

```
class Circle(object):
    pass
```

- We can now create this class

```
b = Circle(object)
```

- And we can check the type to confirm what happened

```
type(x)
__main__.Circle
```

Python Circle Class: Attributes

- Let's add some methods and attributes to the class

```
class Circle ():
    def __init__(self, radius=1):
        print('Creating circle')
        self._radius = r
```

- The function __init__ inside a class is called a **method**
- This special function __init__ is **automatically** called if you create a Circle type – in C++ this is called a **constructor**
- The **self** input is a reference to the object itself
- The radius is an attribute of the Circle
- If it is not passed in, then a default value of 1 is used
- I prefix all class members with _ to signify that it is protected

Python Classes : Naming data members

- Changing the radius can be done explicitly

```
self._radius = 0.25
```

- This may not be desirable
- In other OO languages like C++, some class members can be made **private** so that you cannot change them directly
- The reason is that the coder wants to implement some validation or perhaps there is some sort of order dependency in value setting that needs to be enforced
- This does not exist in Python – it is only enforced by convention
- **The convention is that if you prefix a class member with a _ then you signal to users that it should not be changed like this**

Python Circle Class : Methods

- Methods are functions inside the Class – you must pass **self** as the first argument – this tells the function which object its from

```
PI = 3.14159
class Circle():
…
    def area(self):
        a = PI * self._radius ** 2
        return a
```

- We can call the area method as follows

```
c = Circle(4.0)
c.area()
50.26544
```

Python Circle Class : Using a set method

```
class Circle (object):
def __init__(self,r):
        self._radius = r
def area(self):
        a = PI * self._radius ** 2
        return a
def setRadius(self, r):
         self._radius = r
```

- You shouldn't really do this outside a module

```
circle._radius = 0.25
```

- Do this instead or just create a new Circle object

```
circle.setRadius(0.25)
```

Summary

- The Object-Oriented paradigm is very different to pure functional programming and takes a while to understand
- It makes your code much easier to organize and to understand
- Your classes can be re-used in different projects without any additional effort
- You can control access to data members and ensure that validation is performed
- There is a lot more to object-oriented programming including inheritance that we do not have time to cover here

Case Study: A Vanilla Option Class

A Vanilla Option Class

- What are the attributes of a Vanilla Option
- These are the things you would find on a term sheet
- They are
  - Option Expiry Date
  - Option Strike
  - Option Type – Call or Put
- Should the following be in the class
  - Stock price ?
  - Trade date ?
  - Volatility ?
  - Risk-free rate ?

Initiating the Option Class

- We create and instantiate the class in the usual way

```
class Option(object):

    def __init__ (self, expiry_date, strike_price, option_type ):

        self._expiry_date = expiry_date
        self._strike_price = float(strike_price)
        self._option_type = option_type.upper()

        if self._option_type != "CALL" and self._option_type != "PUT":
            print("Unknown option type")
```

- Once we have checked option_type here we don't need to do it again

Doing some Type Checking

- We may want to do some type checking on inputs
- For example, we may want to check that expiry_date is a date
- We use the command **isinstance** to check and return a message

```
class Option(object):

    def __init__ (self, expiry_date, strike_price, option_type ):

        if isinstance(expiry_date, date) == False:
            print("Expiry date is not a date")

        self._expiry_date = expiry_date
        …
```

- There are better ways to handle errors, but this is a good start

I need to import my dependencies

- I import the functions, classes and set the constants I will need

```
from math import exp, log, sqrt
from scipy import optimize
from scipy.stats import norm
from datetime import date

DAYS_IN_YEAR = 365.242
```

- We need scipy for the optimizer and for the NORMCDF
- Later when I would like vectorised calculations I switch to Numpy

Valuing the Option

- We create and instantiate the class in the usual way

```
def value(self, value_date, stock_price, interest_rate, dividend_yield, volatility ):

    t = abs(self._expiry_date - value_date).days / DAYS_IN_YEAR
    r = interest_rate; q = dividend_yield; s = stock_price;
    k = self._strike_price; v = volatility

    d1 = (log(s/k) + (r - q + v*v / 2.0) * t) / (v * sqrt(t))
    d2 = (log(s/k) + (r - q - v*v / 2.0) * t) / (v * sqrt(t))
    if self._option_type == "CALL":
        v = s * exp(-q * t) * norm.cdf(d1)
        v = v - k * exp(-r * t) * norm.cdf(d2)
    elif self._option_type == "PUT":
        v = k * exp(-r * t) * norm.cdf(-d2)
        v = v - s * exp(-q * t) * norm.cdf(-d1)
    return v
```

I often rename the variables to make the formulae shorter and easier to read

Implied Volatility

- The implied volatility calculation involves a root search
- We use Scipy's Newton function to do this

```
def impliedVolatility(self, value_date, option_mkt_value, stock_price,
                      dividend_yield, interest_rate):

    argtuple = (self, value_date, stock_price, dividend_yield,
                interest_rate, option_mkt_value)

    sigma = optimize.newton(f,x0=0.2, args=argtuple, tol=1e-8, maxiter=50)
    return sigma
```

- We use the tuple args to pass in extra information

Implied Volatility – Objective Function

- We need a function that gives zero at the implied volatility
- This is declared outside the class – so we need to pass all the class information

```
def f(volatility, *args):

    self = args[0]
    valueDate = args[1]
    stockPrice = args[2]
    divYield = args[3]
    interestRate = args[4]
    value = args[5]

    objFn = self.value(valueDate,stockPrice,divYield,volatility,interestRate) - value

    return objFn
```

Calling the Option Class

- We can now call the class and value a call option

```
expiry_date = date(2022, 6, 1)
stockPrice = 100
volatility = 0.30
interest_rate = 0.05
dividend_yield = 0.0
stockPrices = 100.0
value_date = date(2022, 1, 1)

callOption = Option(expiry_date, 100.0, "CALL")

value = callOption.value(value_date, stockPrice, interest_rate,
            dividend_yield, volatility)
```

Object-Oriented Code

- We have only just introduced the idea of OO code
- There are a number of books and I encourage you to read them
- As soon as your project gets large, OO becomes a very powerful way to organise and structure code
- You will get a chance to extend the Option class in the coursework

Case Study: A Bond Class

Objective

- We build a class which gives the price & yield of a standard bond
- It's a very simple class that ignores basis conventions and actual cashflow dates
- The class function takes in
  - Remaining Maturity in years
  - Coupon
  - Frequency
- It's a class you could easily extend for your own needs
- We have already defined a number of bond functions so we re-use those

Bond Class

- What should a bond contain as members ?
- Use "Bond **has** a X" to decide …

```
from HelperFunctions import flowTimes
import scipy.optimize as optimize
from datetime import date
DAYS_IN_YEAR = 365.242

class Bond():
    def __init__(self, maturity_date, coupon, frequency):
        self._maturity_date = maturity_date
        self._coupon = coupon
        self._frequency = frequency
```

- We cannot compute the payment dates as we don't know the value date or the bond issue date

Calculate the Full Price from the Yield

- The full price is the discounted sum of all payments

```
def fullPriceFromYield(self, value_date, yld):
    years = abs(self._maturity_date - value_date).days / DAYS_IN_YEAR
    paymentTimes = flowTimes(years, self._frequency)
    price = 0.0
    for t in paymentTimes:
        df = 1.0/(1.0 + yld/self._frequency)**(t*self._frequency)
        price += ( self._coupon / self._frequency ) * df
    price += df # par
    return price
```

- However, bond prices are usually quoted clean
- We need to subtract the accrued interest

Calculate the Clean Price from the Yield

- The full price is the discounted sum of all payments

```
def cleanPriceFromYield(self, value_date, yld):
    full_price = self.fullPriceFromYield(value_date, yld)
    clean_price = full_price - self.accruedInterest(value_date, )
    return clean_price
```

- We re-use the full price function – avoid duplication of code
- We then subtract the accrued interest

Calculate the Accrued Interest

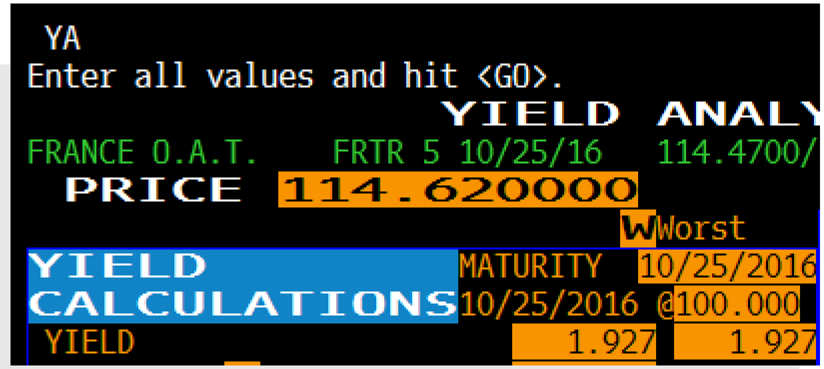
- The accrued is the year fraction from the last coupon to the value / settlement date times the annualised coupon

```
def accruedInterest(self, value_date):
    years = abs(self._maturity_date - value_date).days / DAYS_IN_YEAR
    paymentTimes = flowTimes(years, self._frequency)
    accruedPeriod = 1.0/self._frequency - paymentTimes[0]
    return accruedPeriod * self._coupon
```

- I am not sure if this is optimal – why ?
- How might you improve the code ?

OAT Comparison Revisited

- The full price is the discounted sum of all payments




```
maturity_date = date(2016, 10, 25)
cpn = 0.05
freq = 1
bond = Bond(maturity_date, cpn, freq)
clean_price = 1.1462
yld = bond.yieldFromCleanPrice(value_date, clean_price)
print('Yield:', yld*100)
```

Yield: 1.9280277547132966

- Bloomberg gets 1.927% so we are out by 0.001%
- Why is it not exactly right ? We are not calculating the payment sizes exactly using Actual 360.

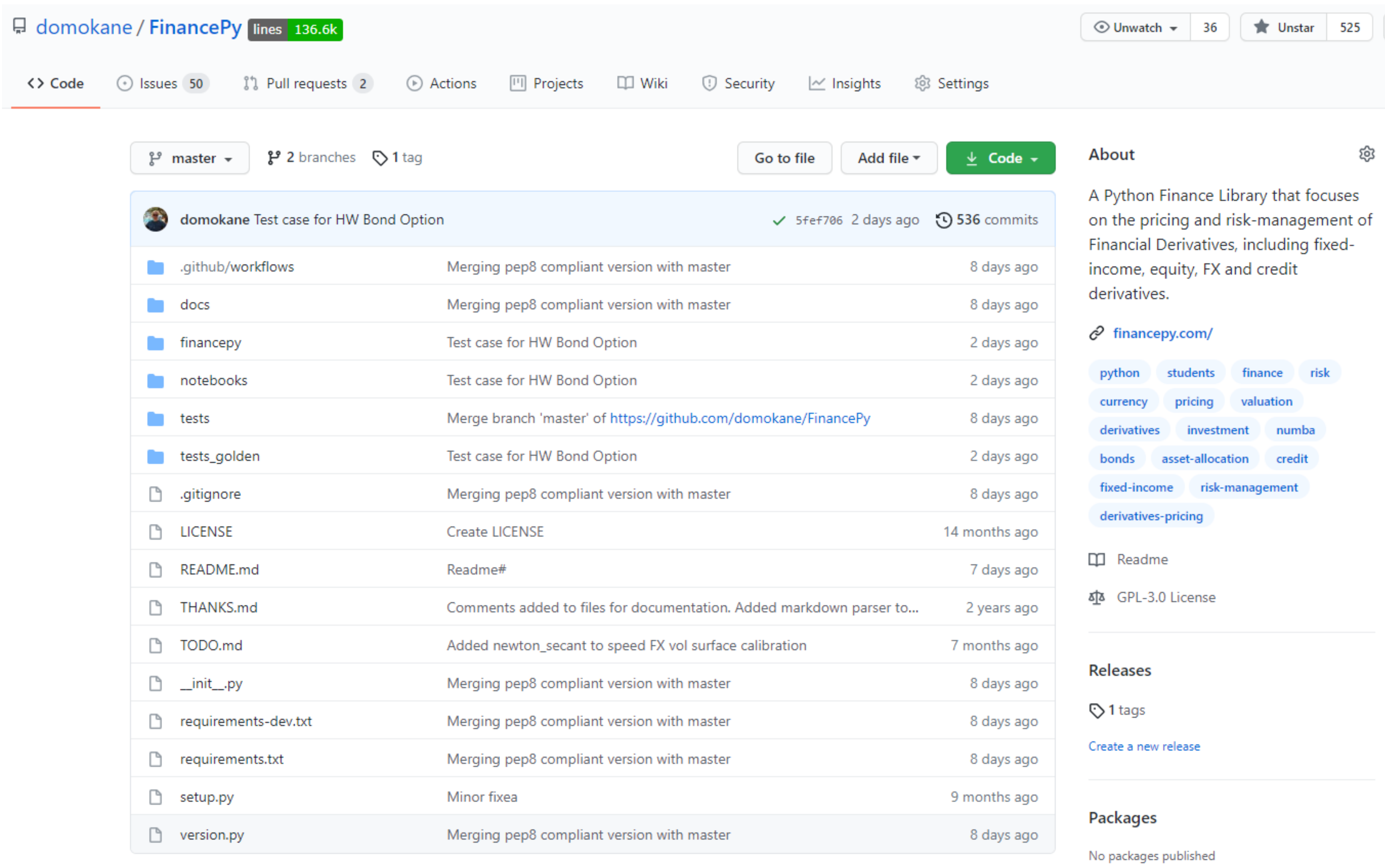
An External Library: FinancePy

What is FinancePy ?

- FinancePy is a Python-based library for the valuation of financial securities, **with a special focus on financial derivatives**
- I have developed this as a teaching tool, and it can also be used by practitioners to do valuation and risk
- It's an example of how you can easily grow the Python ecosystem
- Handles a broad range of asset classes including:
  - bonds
  - equities
  - currencies
  - interest rates
  - inflation
- And derivatives on all of these

FinancePy at Github

- https://github.com/domokane/FinancePy



FinancePy Design

- Utils
  - Basic functionality used across the library
- Market
  - Holders, processors of market data as Python Classes
- Models
  - Quantitative valuation model library as Python Classes
- Products
  - Financial securities including derivatives as Python classes

Utils

- There are a lot of market conventions used in finance
- We ensure these are followed as exactly as possible in FinancePy
- Date
  - In finance, dates are key to determining valuation
  - There are certain key dates (CDS, IMM dates)
- Calendar
  - Need to know all holiday dates in NY, Europe, London, …
- Schedule
  - Need to calculate series of cashflow payment dates in accordance with market conventions
  - Getting the date correct is essential as the timing of payments plus the right discount rate determines the present value

Products

Bonds

- Bond
- BondAnnuity
- BondConvertible
- BondEmbeddedOption
- BondFRN
- BondFuture
- BondMortgage
- BondOption

Credit

- CDS
- CDSBasket
- CDSCurve
- CDSIndexOption
- CDSIndexPortfolio
- CDSOption
- CDSTranche

Funding

- FixedLeg
- FloatLeg
- IborBasisSwap
- IborCallableSwap
- IborDeposit
- IborFuture
- IborFRA
- IborSwap
- IborCapFloor
- IborSwaption
- IborSingleCurve
- IborDualCurve
- IborOIS
- OIS
- OISCurve
- IborBermudanSwaption

Equity

- EquityAmericanOption
- EquityAsianOption
- EquityBarrierOption
- EquityBasketOption
- EquityChooserOption
- EquityCliquetOption
- EquityCompoundOption
- EquityDigitalOption
- EquityFixedLookbackOption
- EquityFloatLookbackOption
- EquityRainbowOption
- EquityOneTouchOption
- EquityVanillaOption
- EquityVarianceSwap

FX

- FXForward
- FXVanillaOption
- FXBarrierOption
- FXBasketOption
- FXRainbowOption
- FXDigitalOption
- FXFixedLookbackOption
- FXFloatLookbackOption
- FXVarianceSwap

Inflation

- InflationBond
- InflationSwap

- Each of these is a Python class under **Products**

Market

- Discounting future cashflows correctly is essential

Discount Curves

- DiscountCurve
- DiscountCurveFlat
- DiscountCurveNS
- DiscountCurveNSS
- DiscountCurvePoly
- DiscountCurvePWF
- DiscountCurvePWL
- DiscountCurveZeros

- Managing the volatility assumptions for options is key

Volatility

- EquityVolCurve
- FXVolSurface
- IborCapVolCurve
- IborCapVolCurveFn

Models

- Models are not product-specific

Lognormal

- GBMProcess
- ModelBlack
- ModelBlackScholes
- ModelBlackScholesAnalytical
- ModelBlackScholesShifted
- ModelCRRTree

Rates

- ModelRatesBDT
- ModelRatesBK
- ModelRatesCIR
- ModelRatesHL
- ModelRatesLMM

Credit

- ModelGaussianCopula
- ModelLossDbnBuilder
- ModelLHPlus
- ModelMertonCredit
- ModelMertonCreditMkt

Normal

- ModelBachelier
- ModelRatesVasicek

Stochastic Vol

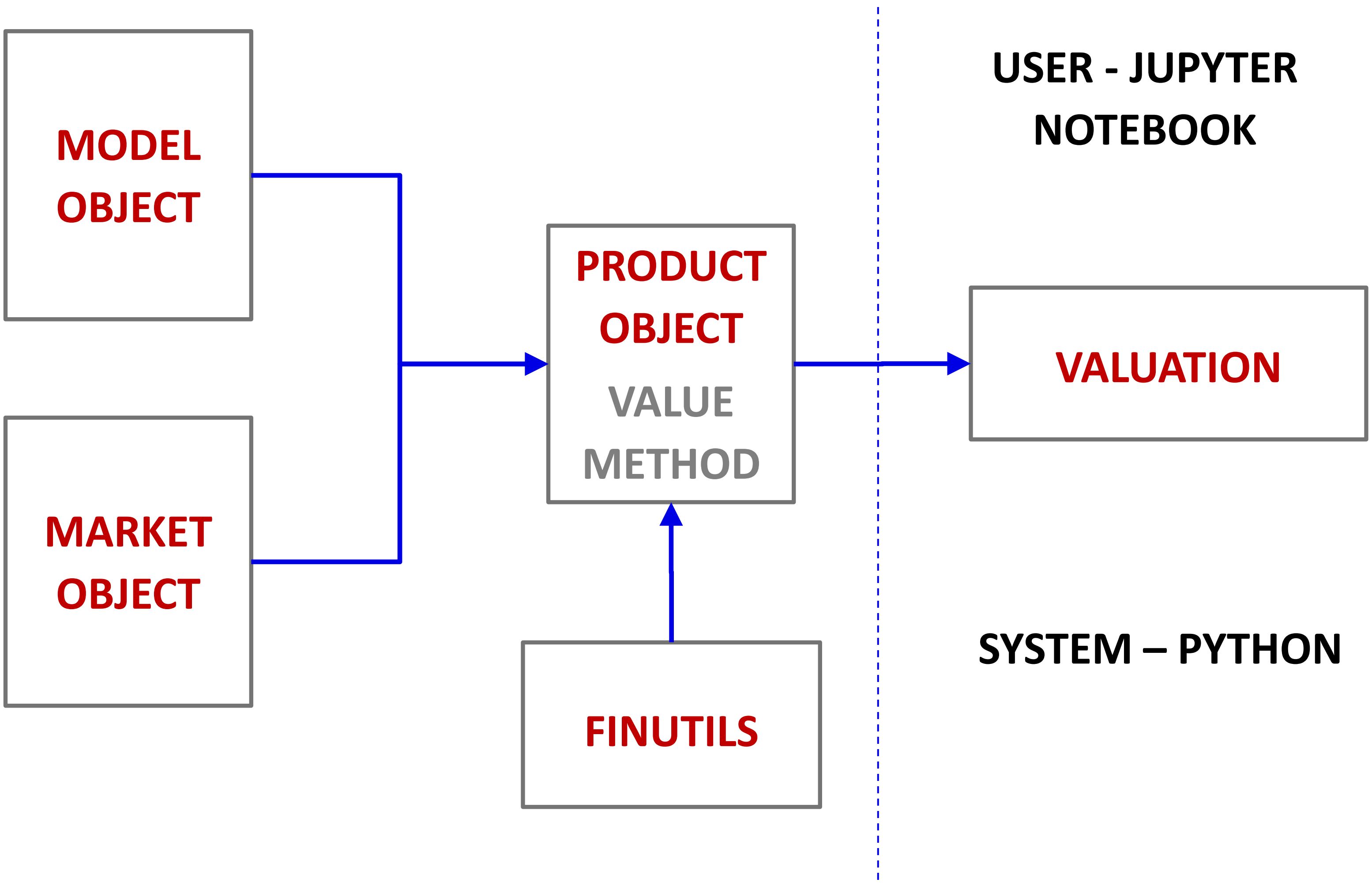
- ModelHeston
- ModelSABR

Modelling Highlights

- Bond yield-curve fitting with multiple parametric forms
- IBOR discount curves with different interpolation schemes
- Two curve construction using Overnight Index Swaps
- Trinomial Trees for interest rate option pricing
- Multi-factor Libor Market Model
- Convertible bond pricing model
- Valuation of Synthetic CDO tranches
- Full calibration to FX volatility surface
- Multi-process simulator with stochastic volatility
- Fast Sobol random number generator
- Variance reduction methods for path dependent options
- and lots more …

Design

- Stage I: Create a Product Object e.g. a call option
- Stage II: Value the product by passing in a model and market



Installing

- Use Pip to install financepy

```
pip install financepy
```

- If you import it then you get a message as follows

```
C:\Users\Dominic>python
Python 3.8.10 (tags/v3.8.10:3d8993a, May  3 2021, 11:48:03) [MSC v.1928 6
Type "help", "copyright", "credits" or "license" for more information.
>>> import financepy as fp
####################################################################
# FINANCEPY BETA Version 0.202 - This build:  16 Jul 2021 at 18:46 #
# **** NEW PEP8 COMPLIANT VERSION -- PLEASE UPDATE YOUR CODE  **** #
#      This software is distributed FREE & WITHOUT ANY WARRANTY    #
#  Report bugs as issues at https://github.com/domokane/FinancePy  #
####################################################################

>>>
```

- You are now ready to use it

Creating and Valuing a Call Option

```
valuation_date = Date(1, 1, 2015)
expiry_date = valuation_date.add_tenor("6M")
strike_price = 50.0
```

```
call_option = EquityVanillaOption(expiry_date, strike_price, FinOptionTypes.EUROPEAN_CALL)
```

```
stock_price = 50
volatility = 0.20
interest_rate = 0.05
dividend_yield = 0.0
```

```
discount_curve = DiscountCurveFlat(valuation_date, interest_rate)
dividend_curve = DiscountCurveFlat(valuation_date, dividend_yield)
model = BlackScholes(volatility)
```

```
call_option.value(valuation_date, stock_price, discount_curve, dividend_curve, model)
```

```
3.4276581469416914
```

```
print(call_option)
```

```
OBJECT TYPE: EquityVanillaOption
EXPIRY DATE: 01-JUL-2015
STRIKE PRICE: 50.0
OPTION TYPE: FinOptionTypes.EUROPEAN_CALL
NUMBER: 1.0
```

Vectorisation

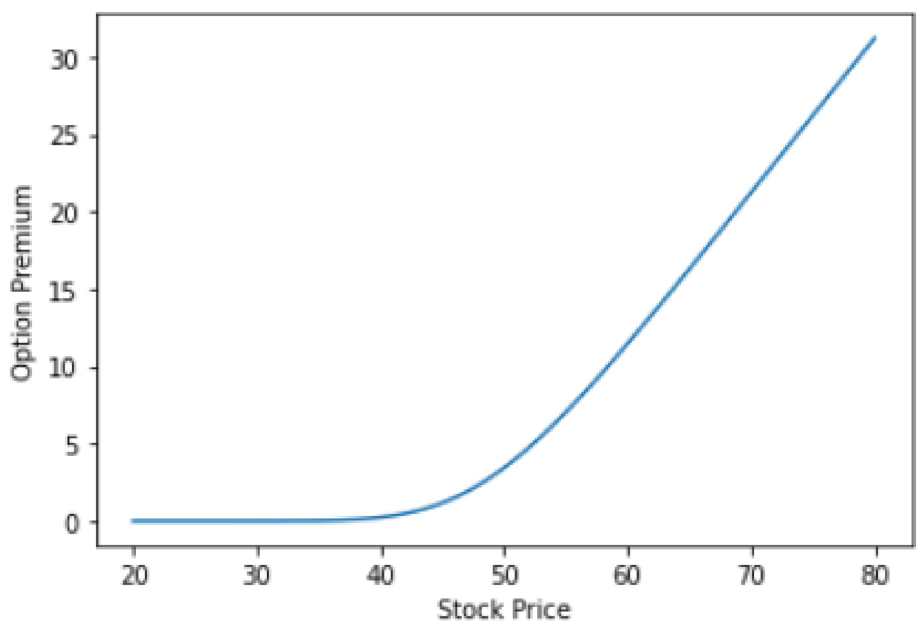
- Make stock prices a vector - can plot a function with a single call

```
stock_prices = np.linspace(20,80,100)
```

```
value = call_option.value(valuation_date, stock_prices, discount_curve, dividend_curve, model)
```

```
plt.plot(stock_prices, value)
plt.xlabel("Stock Price")
plt.ylabel("Option Premium")
```

```
Text(0, 0.5, 'Option Premium')
```



Interest Rate Swap

- We define a fixed-floating IBOR swap

```
swap_calendar_type = CalendarTypes.NONE
bus_day_adjust_type = BusDayAdjustTypes.FOLLOWING
date_gen_rule_type = DateGenRuleTypes.BACKWARD

fixed_coupon = 0.05
fixed_freq_type = FrequencyTypes.ANNUAL
fixed_day_count_type = DayCountTypes.THIRTY_E_360_ISDA

float_spread = 0.0
float_freq_type = FrequencyTypes.ANNUAL
float_day_count_type = DayCountTypes.THIRTY_E_360_ISDA

swapType = SwapTypes.RECEIVE
notional = ONE_MILLION

start_date = Date(20, 6, 2020)
maturity_date = start_date.add_tenor("5Y")
```

Creating the IborSwap

- We create the swap and can examine the payments

```
swap = IborSwap(start_date, maturity_date, swapType,
                fixed_coupon, fixed_freq_type, fixed_day_count_type,
                notional,
                float_spread, float_freq_type, float_day_count_type,
                swap_calendar_type, bus_day_adjust_type, date_gen_rule_type)
```

```
swap._fixed_leg.print_payments()
```

```
START DATE: 20-JUN-2020
MATURITY DATE: 20-JUN-2025
COUPON (%): 5.0
FREQUENCY: FrequencyTypes.ANNUAL
DAY COUNT: DayCountTypes.THIRTY_E_360_ISDA
PAY_DATE     ACCR_START   ACCR_END      DAYS  YEARFRAC    RATE       PAYMENT
21-JUN-2021  20-JUN-2020  21-JUN-2021   361  1.002778  5.000000     50138.89
20-JUN-2022  21-JUN-2021  20-JUN-2022   359  0.997222  5.000000     49861.11
20-JUN-2023  20-JUN-2022  20-JUN-2023   360  1.000000  5.000000     50000.00
20-JUN-2024  20-JUN-2023  20-JUN-2024   360  1.000000  5.000000     50000.00
20-JUN-2025  20-JUN-2024  20-JUN-2025   360  1.000000  5.000000     50000.00
```

We value the Swap using a Flat Curve at 6%

- We have several ways to construct market discount curves
- The simplest is the flat discount curve which takes a zero rate

```
from financepy.market.curves.discount_curve_flat import DiscountCurveFlat
```

```
valuation_date = Date(20,6,2018)
settlement_date = valuation_date
```

```
rate = 0.06
discount_curve = DiscountCurveFlat(valuation_date, rate, FrequencyTypes.ANNUAL,
                                   DayCountTypes.THIRTY_E_360_ISDA)
```

```
swap.value(settlement_date, discount_curve, discount_curve)
```

```
-37490.10023311118
```

- We value the 5% fixed receiver swap with a 6% discount rate
- The MTM is negative, as expected – a new swap would pay us 6% instead of the 5% we have locked in.

CDS Contracts can be created

- We create a CDS contract

```
trade_date = Date(3, 2, 2011)
effective_date = Date(4, 2, 2011)
settlement_date = Date(6, 2, 2011)
```

```
maturity_date = Date(20, 3, 2016)
cdsCoupon = 0.010
notional = ONE_MILLION * 10
long_protection = True
```

```
cds_contract = CDS(effective_date, maturity_date, cdsCoupon, notional, long_protection)
```

- This is $10m face amount of long protection 5Y contract with a 100bp running coupon

We Build a Swap Curve to Discount with

- Calibrate to deposits and swaps

```
depos = []
depoDCCType = DayCountTypes.ACT_360

depo = IborDeposit(effective_date, "1M", 0.002630, depoDCCType); depos.append(depo)
depo = IborDeposit(effective_date, "2M", 0.002870, depoDCCType); depos.append(depo)
depo = IborDeposit(effective_date, "3M", 0.003105, depoDCCType); depos.append(depo)
depo = IborDeposit(effective_date, "6M", 0.004608, depoDCCType); depos.append(depo)
depo = IborDeposit(effective_date, "9M", 0.006205, depoDCCType); depos.append(depo)

swaps = []
fixedDCCType = DayCountTypes.THIRTY_E_360_ISDA
fixedFreqType = FrequencyTypes.SEMI_ANNUAL
swapType = SwapTypes.PAY

swap = IborSwap(effective_date, "1Y", swapType, 0.007861, fixedFreqType, fixedDCCType); swaps.append(swap)
swap = IborSwap(effective_date, "2Y", swapType, 0.008799, fixedFreqType, fixedDCCType); swaps.append(swap)
swap = IborSwap(effective_date, "3Y", swapType, 0.013958, fixedFreqType, fixedDCCType); swaps.append(swap)
swap = IborSwap(effective_date, "4Y", swapType, 0.018825, fixedFreqType, fixedDCCType); swaps.append(swap)
swap = IborSwap(effective_date, "5Y", swapType, 0.023251, fixedFreqType, fixedDCCType); swaps.append(swap)
```

```
libor_curve = IborSingleCurve(effective_date, depos, [], swaps, interp_type = InterpTypes.LINEAR_FWD_RATES)
```

- You can do a two-curve bootstrap too – but this is beyond the scope of this course

Can then construct a CDS Issuer Curve

- We can calibrate to a set of CDS to create an issuer curve
- We need to pass in the IBOR discount curve
- This is a flat CDS curve at 70bps

```
cds1 = CDS(effective_date, "1Y", 0.0070)
cds2 = CDS(effective_date, "2Y", 0.0070)
cds3 = CDS(effective_date, "3Y", 0.0070)
cds4 = CDS(effective_date, "4Y", 0.0070)
cds5 = CDS(effective_date, "5Y", 0.0070)
```

```
cdss = [cds1, cds2, cds3, cds4, cds5]
```

```
recovery_rate = 0.40
```

```
issuer_curve = CDSCurve(effective_date, cdss, libor_curve, recovery_rate)
```

CDS Valuation

```
cds_contract.value(settlement_date, issuer_curve, recovery_rate)
```

```
{'full_pv': -157675.65231979737, 'clean_pv': -144897.87454201956}
```

```
cds_contract.print_flows(issuer_curve)
```

PAYMENT_DATE	YEAR_FRAC	FLOW	DF	SURV_PROB	NPV
21-MAR-2011	0.252778	25277.78	0.999660	0.998541	25232.30
20-JUN-2011	0.252778	25277.78	0.998602	0.995596	25131.29
20-SEP-2011	0.255556	25555.56	0.996684	0.992629	25283.05
20-DEC-2011	0.252778	25277.78	0.993950	0.989702	24866.09
20-MAR-2012	0.252778	25277.78	0.990722	0.986783	24712.25
20-JUN-2012	0.255556	25555.56	0.987955	0.983850	24839.98
20-SEP-2012	0.255556	25555.56	0.985537	0.980926	24705.56
20-DEC-2012	0.252778	25277.78	0.983489	0.978042	24314.52
20-MAR-2013	0.250000	25000.00	0.981223	0.975198	23922.17
20-JUN-2013	0.255556	25555.56	0.977037	0.972305	24277.21
20-SEP-2013	0.255556	25555.56	0.971051	0.969420	24056.89
20-DEC-2013	0.252778	25277.78	0.963393	0.966575	23538.44
20-MAR-2014	0.250000	25000.00	0.955926	0.963769	23032.31
20-JUN-2014	0.255556	25555.56	0.948942	0.960912	23302.84
22-SEP-2014	0.261111	26111.11	0.940687	0.958003	23530.84
22-DEC-2014	0.252778	25277.78	0.931648	0.955194	22494.81
20-MAR-2015	0.244444	24444.44	0.923029	0.952486	21490.87
22-JUN-2015	0.261111	26111.11	0.914059	0.949604	22664.28
21-SEP-2015	0.252778	25277.78	0.904536	0.946822	21648.76
21-DEC-2015	0.252778	25277.78	0.894215	0.944049	21339.06
21-MAR-2016	0.252778	25277.78	0.884138	0.941283	21036.78