# Advanced Programming Concepts in Python

**Goals:**

- Discuss some best practice for making your code readable and easy to debug
- Review various Data Structures in Python
- Learn about object-oriented programming in Python
- Discuss some more advanced features in Python (recursive functions, exception handling)

## Course Projects

- **Reminder: Project Proposals are due October 15th before class**
  - How many have chosen a group?
  - How many have an idea for a project?
- I encourage all groups to discuss their topic with me prior to the deadline.

## Review: Working with Databases

- When building applications, we need to be able to manipulate and store data from heterogeneous sources.
- This requires a large amount of interacting with **databases**.
- In particular, we need to define a **schema**, or set of relational tables that defiines our data.
- We also need to write a series of INSERT, UPDATE and DELETE queries, or stored procedures, that can be used to process new data as we receive it.
- We also need to be able to provide the data to our underlying model or application in a convenient way. This can be done via views or SELECT statements.

## Review: Different Clauses in SQL SELECT Statements

- SQL SELECT components:
  - SELECT: defines the columns return in your query
  - FROM: defines the underlying tables in your query
  - WHERE: defines how we restricts rows in our query
  - GROUP BY: defines how our data is aggregated
  - HAVING: defines how we restrict the groups return in our query
  - ORDER BY: defines how we sort the results
- Note that the clauses must show up in your query in this order. Switching any of these clauses will produce a syntax error.
- These clauses only define how data is returned. No tables or underlying data is modified when running these queries.

## Review: Most Useful Features of SQL Programming

- JOINs
- Aggregation via GROUP BY clauses
- Sub-Queries
- Pivoting data with CASE statements
- INSERT, UPDATE and DELETE queries

## Review: References for Database Programming

- There are many features of SQL that we don't have the time to cover in this course. Here are a few references of SQL programming for those interested in diving deeper:
  - Database Programming in Python
  - SQL Programming
  - SQL Server Docs
  - PostgreSQL Docs

## Review: Lifecycle of a quant software project

- Stages of a quant software project:
  - Data Parsing / Collection
  - Data Cleaning
  - Database Design & Data Storage
  - Model Implementation
  - Model Validation
  - Presentation of model results via an interface

## Review: What we have covered so far?

- So far, we have discussed:
  - Data Collection & Cleaning
  - Interacting with databases
  - Working in a multi-programmer environment
- For the next few weeks, we will focus on the model implementaiton in Python and C++.
- Today we will focus on Python, and next week we will start C++.
- If you haven't already, please make sure your preferred C++ development environment is installed prior to class next week.

## Best Practices: Suggestions on Making Code Readable

- **Documentation**: each user-defined function and/or class that you write should be well documented. You should explain the parameters to the function and its return value.

  You should also describe the purpose of the function and any limitations / expectations of the user.

- **Naming Conventions**: variables, functions and classes should all be named consistently and as intuitively as possible.

  For example, when I am creating a database, I often name all tables with the prefix "tbl_"

  Similarly, In Python/C++ any function that retrieves a dataset from a database could all start with the prefix "get"

## Best Practices: Suggestions on Making Code Readable

- If you are implementing a model, it is helpful to define variable names that are consistent with the documentation on the model.
  - For example, consider the Black-Scholes formula:

$$c_0 = \tilde{\mathbb{E}}\left[e^{-rT}\left(S_T - K\right)_+\right] = \Phi(d_1)S_0 - \Phi(d_2)Ke^{-rT} \tag{1}$$

$$\begin{aligned} d_1 &= \frac{1}{\sigma\sqrt{T}}\left(\ln\frac{S_0}{K} + \left(r + \frac{\sigma^2}{2}\right)T\right), \\ d_2 &= d_1 - \sigma\sqrt{T} \end{aligned} \tag{2}$$

  - Variables named d1 and d2 would be a better names for the left hand side of (2) than x and y, for example.

## Best Practices: Suggestions on Making Code Readable

- It is helpful to write your code such that complex math formulas span multiple lines.

  This makes the code more readable and perhaps more importantly enables us to place **breakpoints** in the middle of the calculation.

```python
def callPx(s_0, k, r, sigma, tau):
    sigmaRtT = (sigma * math.sqrt(tau))
    rSigTerm = (r + sigma * sigma/2.0) * tau
    d1 = (math.log(s_0/k) + rSigTerm) / sigmaRtT
    d2 = d1 - sigmaRtT
    term1 = s_0 * cdf(d1)
    term2 = k * math.exp(-r * tau) * cdf(d2)
    return  term1 - term2
```

## NumPy Module

- NumPy is an important module for performing calculations in Python.
- It has robust implementations of several important data types, including Arrays and Matrices, which we will use frequently.
- It also has a large library of Mathematical tools in it including:
  - Random Number Generation
  - Linear Algebra Functions (e.g., Matrix Inversion)
  - Fourier Transform Calculations
- The documentation for NumPy has more details about the extent of its capabilities: NumPy Docs

## Pandas Module

- The Pandas module is Python contains a set of data structures and some useful accompanying functionality.
- This module is designed for working with relational data (i.e. data from a database)
- It has useful structures for time series and cross-sectional data analysis
- The documentation for Pandas has more details: Pandas Docs

## Pandas Objects

- All Pandas objects are **derived** from numpy array objects meaning that they **inherit** the functionality of numpy arrays
- All Pandas object give us the ability to:
  - Index and Slice our data
  - Append to our data
  - Repeat our data
  - Iterate through our data with a for loop
- In addition, they also provide much more functionality, as we will see when we discuss the pandas.series and pandas.dataframes objects later in this lecture.

## Python Data Structures

- The most common data structures that you will use in Python are:
  - Tuples
  - Arrays
  - Matrices
  - Lists
  - Dictionaries
  - Series
  - DataFrames
- Generally speaking, there are functions/methods for converting between these data structures.
- Let's spend a minute reviewing these structures as they are central to programming effectively in Python.

## Tuples

- Tuples are a flexible and easy to use data structure that can handle heterogeneous data with multiple underlying types.
- Tuples can be nested: we can have multi-dimensional tuples.

```
        tuple1 = ('SPY', 290.31)
        tuple2 = ('QQQ', 181.24)
        tuples = (tuple1, tuple2)
```

- Each tuple in a multi-dimensional tuple can have a different number of items, and have different data types in each index.

```
        tuple1 = ('SPY', 290.31, 1)
        tuple2 = (181.24, 'QQQ')
        tuples = (tuple1, tuple2)
```

- This high level of flexibility make tuples useful structures for dealing with unstructured data.

## Lists

- Lists are similar to tuples and also can handle generic, heterogeneous data in a robust way.
- Lists can also be nested and multi-dimensional.
- Lists are created using brackets rather than parenthesis.

```
list1 = ["SPY", 1, 2]
list2 = [0, "QQQ"]
lists = [list1, list2]
```

- Mathematical operators are not defined for lists, even for list with numeric types.
- The $+$ operator, in particular, is defined as *concatenate* rather than *sum*.

## Differences between Lists and Tuples

- The biggest difference between lists and tuples is that lists are mutable whereas tuples are immutable.
- This means we can modify lists element-by-element but cannot do so with tuples.
- This makes lists more useful, generally speaking, but makes tuples particularly useful for returning data from a function, where we would generally unpack the tuple immediately after returning from the function.

## Dictionaries

- **Dictionaries** are an important data structure in Python that consists of a series of key-value pairs.
- The keys in a dictionary are **immutable**. What does this mean?
- The values in a dictionary can be modified for a specific index using the [] operator. If the item does not already exist in the dictionary, then it will be added.
- Dictionaries are extremely useful objects for keeping lookup / mapping tables.

  For example, we might keep a dictionary that stores instrument id's and instrument tickers, making it easy to retrieve the id for any instrument.

## Dictionaries

- A blank dictionary can be created using the following syntax:

```
        dict = {}
```

  The [] operator could then be used to add items.

- A dictionary with items can also be created by specifying the key value pairs in brackets:

```
        dict = {0: "SPY", 1: "QQQ", 2: "IWM"}
```

- As with other built-in classes, Dictionaries come with a set of built-in functions & methods, such as:
  - print / len
  - del
  - keys / values
  - in

## numpy Arrays

- numPy arrays are multi-dimensional array objects.
- numeric data is generally stored in numpy arrays, not lists and tuples
- numPy arrays are mutable, meaning we can modify the elements after we have assigned them.

```
        import numpy as np

        a = np.array([1,2,3])
        a[1] = 3
```

- Unlike for lists, mathematical operators are defined on arrays. For example, the + operator is defined as addition, and the * operator is defined as multiplication.

## numpy Arrays

```
a = np.array([1,2,3])
b = np.array([4,5,6])
c = a + b
```

- Mathematical operations are applied **element-wise**.

```
a = np.array([1,2,3])
b = np.array([4,5,6])
c = a * b
```

- Note that these operators return the element-wise sum or product rather than modifying either input array.
- In addition to the built-in + and * operators, a set of functions including sum() and mean() are defined for numpy arrays.

```
a = np.array([1,2,3])
sum_a = a.sum()
```

## Differences between Lists and numpy Arrays

- The main difference between a list and a numpy array is the presence of mathematical operations such as the $+$ operator and the * operator.
- I tend to use lists for less structured data and arrays for numeric data.
- numPy arrays can store non-numeric data, but it will cause errors when utilizing the built-in mathematical operations.

## numpy Matrices

- numpy Matrices inherit from numpy arrays and as a result provide similar functionality.
- A numpy matrix will behave very similarly to a 2-dimensional numpy array.
- The main difference is in how matrix operations are handled.
- The * operator of a numpy matrix performs matrix multiplication, rather than element by element multiplication.

```
import numpy as np
mat1 = np.matrix('1 2; 3 4 ')
mat2 = np.matrix('2 4; 6 8')
mat = mat1*mat2
```

- Notice that this will behave differently than the same operation on a numpy array.

## Matrix Calculations in Arrays vs. Matrices

```
mat1 = np.matrix('1 2; 3 4 ')
mat2 = np.matrix('2 4; 6 8')
mat = mat1*mat2
print(mat)
arr1 = np.array(mat1)
arr2 = np.array(mat2)
arr = arr1*arr2
print(arr)
```

This code will return

```
[[14 20]
[30 44]]
[[ 2  8]
[18 32]]
```

To get the numpy array class to perform matrix multiplication, we can use the **numpy.dot** function.

## Working with Covariance Matrices

- The numpy module has built in functions for computing covariance and correlation matrices, numpy.cov and numpy.corrcoef respectively.
- Calculating and working with covariance matrices is a common task in quant finance, especially in:
  - Portfolio Optimization / Construction
  - Risk Management
  - Simulation
- Certain transformations of a covariance matrix will also appear frequently in particular:
  - Inversion
  - Square root (and other powers)
- We need methods to handle matrices that are not full rank.

## pandas Series

- A pandas series is a one dimensional array with an index.
- pandas series are useful for time-series analysis, where the index would correspond to the date field.
- The index does not have to be a date field however.

```python
import pandas as pd
s = pd.Series(data=np.random.random(size=5),
                 index=['2014','2015','2016','2017','2018'])
```

- As we saw with dataframes last week, pandas series structures have a lot of helpful functions for aggregating the data, selecting subsets and performing other manipulations.
- More info on the pandas series structure can be found here: Pandas Series

## pandas DataFrames

- DataFrames are tabular data structures where both the rows and columns can be named.
- Like pandas.series, they contain an index. In financial applications, this will often be a date.
- There can be many columns of data, and there are useful functions for merging or joining the data.
- Many useful aggregation, joining and subselecting functions exist for Dataframes.

```
import pandas as pd
df = pd.DataFrame(data=np.random.random(size=(5,3)),
        index=['2014','2015','2016','2017','2018'],
        columns=['SPY', 'QQQQ', 'IWM'])
```

## Calculating returns using head / tail

- The **head** and **tail** functions return the first n rows and last n rows respectively.
- These functions have a natural application to computing rolling, overlapping returns with period length equal to n.
- To do this, we can create an array of current values using the tail function (with parameter -n) and an array of previous values using the head function.

## Merging DataFrames

- An important feature of Dataframes is the ability to merge columns on an index (usually a date). Here is a simple example:

```
dfSPY = pd.DataFrame(data=np.random.random(size=(5,1)),
          index=['2014','2015','2016','2017','2018'],
          columns=['SPY'])
dfQQQ = pd.DataFrame(data=np.random.random(size=(5,1)),
          index=['2014','2015','2016','2017','2018'],
          columns=['QQQ'])
dfSPY.join(dfQQQ)
```

- As we can in SQL, we can define different JOIN types when doing this.

## Concatenating/Appending DataFrames

- Another important feature of Dataframes is the ability to concatenate or append rows. Here is a simple example:

```python
dfSPYOld = pd.DataFrame(data=np.random.random(size=(5,1)),
                        index=['2009','2010','2011','2012','2013'],
                        columns=['SPY'])
dfSPYNew = pd.DataFrame(data=np.random.random(size=(5,1)),
                        index=['2014','2015','2016','2017','2018'],
                        columns=['SPY'])
dfSPY = pd.concat([dfSPYOld, dfSPYNew])
```

- Note that we passed a list of Dataframes to the concat function.

## Recursive Functions

- Recursive functions are functions that call themselves.
- Each successive call within the recursive function should simplify the calculation until it ultimately reaches a point where the function returns a value.
- When we write recursive functions, we need to make sure that all paths of the calculation reach an end-point.
- Most programming languages have a limit to the depth of recursion that is allowed in order to prevent infinite recursion.
- Later in the course you will have an opportunity to use a recursive function to compute the value of an option using a **binomial tree**.
- **Question**: What other examples can you think of where recursion might be appropriate?

## Recursive Functions: Factorial Example

- A simple example of a recursive function is one that computes a number's factorial

```python
def recur_factorial(n):
    if n == 1:
        return n
    else:
        return n*recur_factorial(n-1)
```

- If we were to replace the above code with the following:

```python
def recur_factorial(n):
    return n*recur_factorial(n-1)
```

Then we would receive a maximum recursion depth exception.

## Recursive Functions: Comments

- In practice, I haven't found recursive functions to be of critical importance and have only used them a few times.
- Having said that, recursive functions are often used as programming interview questions and are worth being familiar with.
- Recursive functions can be difficult to debug as finding the failing location to place a breakpoint can be challenging.
- When you do use recursive functions, it is important to write accompanying simplified test cases to catch any programming issues.

## Statistical Analysis in Python

- As quants, our jobs are generally centered on performing some sort of statistical analysis.
  - This could mean performing a set of econometric tests on an underlying data.
  - It could also mean understanding the underlying distribution for a complex derivative instrument.
  - Question: what is the fundamental difference between the two?
- In either case, we're going to continue to rely on a fundamental set of quant tools to perform these tasks.
- Python is a convenient home for these types of tasks as it provides a significant amount of built-in functionality.
- Let's briefly discuss some of the quant tools that are available to us in Python

## Random Number Generation

- We can use the numpy module to generate random numbers.
- The most common distributions that we will want to randomly sample from are:
  - Uniform
  - Normal
- The function **numpy.random.rand** can be used to generate uniform random normals between 0 and 1.

```
rnds = np.random.rand(100, 100)
```

- The function **np.random.normal** can be used to generate normal random variables:

```
mat_shape = (100, 100)
norm_rnds = np.random.normal(0.0, 1.0, mat_shape)
```

## Random Number Generation: Comments

- Note that the return value for both function calls is a multi-dimensional array.
- The example above generated samples from a standard normal distribution. Modifying the first two parameters enables us to generate samples from a normal with a different mean and variance.
- The numbers that we are generating will be *uncorrelated*. Later we will see how we need to generate *correlated normal random variables* when we are working with multi-asset problems.
- Multi-variate random normals with a specified covariance matrix can be generated using the **numpy.random.multivariate_normal** function.

## Linear Regression

- The sklearn module has a function for fitting linear regression models via OLS:

```python
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model

Xs = np.random.rand(100, 2)
Ys = np.random.rand(100, 1)

# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(Xs, Ys)

print(regr.coef_)
```

- Notice that in order to perform the regression we needed to first create an instance of a class (named regr above) and then utilize the fit function.
- Linear regression will be one of the most common tasks that you will perform as a quant.
- As you've seen in your homework, we generally rely on linear regression when we are doing factor modeling.

  Where else might we use it?

## Time Series Analysis

- Python also has built-in libraries/modules for handling most common time series techniques. Here are a few examples:
- **Stationarity Tests**: The most common stationarity test is the augmented Dickey-Fuller test, which can be found in the statsmodels module. The function is: **statsmodels.tsa.stattools.adfuller**
- **ARMA Models**: The most common method for checking time series for mean-reversion or momentum. In Python these models can be calculated using the **statsmodels.tsa.ARMA** function.
- In your homework, you have check for mean-reversion using a traditional regression approach. This assumes returns follow an AR(1) process.

## Time Series Analysis: Comments

- These types of techniques are commonly used in forecasting and alpha research.
- If we are able to find tradeable market quantities that are stationary, then the reward is high. As a result, this is at the heart of the work for many buy-side quants.
- Generally speaking, those who work with equities, will focus on ARMA models and those who work with options and volatility will focus on GARCH models.
- These types of models also have applications in risk management.

## Exception Handling in Python

- Errors in your code could happen for many reasons.
- When they do happen, they trigger **exceptions** in your code.
- Depending on the type of error that occurs, a different type of exception will be thrown.
- A few simple ways to generate exceptions are:
  - Divide a number by Zero
  - Open a file that doesn't exists
  - Try to change a single character in a string
  - Access a value in a list or array beyond its length
- Exceptions occur even in well-written code.
- As a result, it is best practice to handle these exceptions explicitly in your code.

## Try Except Finally Statements

- In order to handle exceptions in Python, we use the except keyword to define the code that is run after an exception occurs:

```
try:
    somecode()
except:
    handle_exception()
```

- The handle_exception() function might be a place where we:
  - Write a detailed explanatory message to the console
  - Write a warning message to a log file
  - Exit an application safely
  - Modify some variables prior to re-trying the code
- NOTE: It is our job to make sure that any exception that occurs in our code is easy to identify and fix not just by us but by any user.

## Raising Exceptions

- In addition to handling exceptions that are thrown in various parts of our code, we may also want to force an exception to be thrown in certain situations to avoid the code continuing and producing a misleading or dangerous result.
- As an example, we might want to raise an exception in our Black-Scholes formula calculation if the $\sigma$ that is passed to the function is negative.
- In order to do this, we could add the following check to the beginning of the callPx function on slide 11:

```
    if (sigma < 0):
        err = ValueError("Negative sigma provided")
        raise err
```

## Object-Oriented Programming

- Python as a language has robust support for class and object-oriented programming concepts.
- In MatLab or R we generally write code that is functional, that is, we define a set of functions to complete the relevant task.
- When performing exploratory analysis, this is often the most efficient way to write code.
- While this is sometimes convenient, it is generally not as scalable or robust.

## Principles of Object-Oriented Programming

- Object-oriented programming relies on a set of classes, or objects.
- These objects can be related to each other.
- We could have a class with an attribute equal to another class.
- We can also have classes **inherit** functionality and structure from other classes.
- We can use classes to **encapsulate** and **abstract** the data in our classes.

## Classes in Python

- Classes define complex user-defined data types.
- These data types can contain within them a variety of embedded functionality.
- They can contain certain **properties or attributes** which help us define our data type.
- These attributes may be visible to the outside world, or they may only be visible within the class.
- They can also define **functions or methods** that act on them.
- Again, these functions that could only be called internally (i.e. as part of the constructor when the object is being initialzied) or may be functions that may be called by the user at any point.

## Classes in Python

- For example, we might create a *Black-Scholes stochastic process* class:
  - What attributes would we define for this class?
  - What functions could we define?

## Defining Classes in Python

- Class definitions in Python consists of the class keyword, at least one constructor, a set of attributes and a series of member functions:

```
        class Position:
            def __init__(self):
                self.shares = 0
                self.price = 0
```

- Notice that the constructor takes self as a parameter.
- All attributes for the class must be defined in the constructor. They can then be referenced later in functions.
- Note that referencing an attribute in Python requires self.attribute syntax.

## Constructors

- A constructor is a piece of code that runs when an object is **initialized**.
- In Python, a constructor is defined using the following init syntax:

```
    def __init__(self):
        initialize(self)
```

- A default constructor is one that takes no arguments (aside from self). Any member data in the class would then be set to default values.
- All constructors take a parameter *self* which refers to the current instance of the class.
- For each class, we may choose to define default values for some or all of the parameters in the constructor.

- The constructor is called when we initialize the object, for example:

```
pos = Position()
```

  would initialize an instance of our user-defined Position class, and the shares and price would be set to zero.

- We could create a constructor that takes shares and price as parameters as well:

```
class Position:
    def __init__(self, shrs=0, px=0):
        self.shares = shrs
        self.price = px
```

- This constructor would be then be called when we ran the following code:

```
pos = Position(100, 100)
```

## Destructors

- Just as we are able to define code that is executed each time an instance of an object is initialized, we are also able to define code that executes every time an object gets destroyed.
- This is a critical component of programming in C++, as we will see shortly, but less useful in Python as much of it is done automatically once things go out of scope.
- An example destructor would look like:

```
class Position:
    def __init__(self, shrs=0, px=0):
        self.shares = shrs
        self.price = px

    def __del__(self):
        print("destroying object")
```

## Class Attributes

- Attributes, or member data, are variables stored within a class.
- These attributes may have simple built-in types or be other user-defined complex types.
- You may store base classes or derived classes as attributes.
- We reference attributes using the syntax *self.attribute*
- Continuing with our Position example, we could read, or even modify the number of shares or price in code via the attribute:

```
pos = Position(100, 100)
s = pos.shares
pos.shares = 200
```

## Public vs. Private Attributes

- It is best practice to not modify attributes directly like this (Why?)
- Instead, we could define a setShares member function of the class
- Python does not fully support private attributes as C++ does but we still want to utilize the concept as it helps us write safer code.
- To do this it is standard to define attributes that should not be modified outside of their class with an __ prefix.

## Class Methods

- In addition to defining attributes in our classes we may want to define a set of actions for the class.
- These actions could be defined to modify the attributes safely.
- They could also be defined to perform calculations: for example we could add the following to the Position class:

```python
class Position:
    def __init__(self, shrs=0, px=0):
        self.shares = shrs
        self.price = px

    def mktValue(self):
        return self.shares*self.price

pos = Position(100, 50)
mktval = pos.mktValue()
```

## Class Methods

- Notice that self is again defined as a parameter to the mktValue function, as it was in the constructor.
- This is because it is a class method, and therefore has access to all of the attributes of the class.

## Class Methods vs. Global Functions

- Another way to create a market value function would have been to create a function outside of the Position class that took an instance of a Position object as a parameter.
- This enables us to accomplish the same result but there is an important subtle difference.
- In the first approach, the class was being accessed from inside itself, whereas in the second approach the class was being accessed externally.
- If we had attributes that we wanted to be private, then we would not want to use the second approach to access them.

## Class Methods vs. Global Functions

```
class Position:
    def __init__(self, shrs=0, px=0):
        self.shares = shrs
        self.price = px

    def mktValue(self):
        return self.shares * self.price

def mktValue2(pos):
    return pos.shares * pos.price

pos = Position(100, 50)
mktval = pos.mktValue()
mktval2 = mktValue2(pos)
```

## Special Class Methods & Operators

- All classes in Python have a set of pre-defined built-in functions, such as the $==$ operator.

```
        pos = Position(100, 50)
        pos2 = Position(100,50)
        print(pos == pos2)
```

- Note that by default the $==$ operator refers to two objects referring to the same instance, not having the same attribute values.
- We may also want to define certain additional convenient operators, such as the [] and $+$ operator.
- If you try this on our Position class as is you will see that an error message is produced.

## Operator Overloading

- When we change the definition of one of these built-in functions, it is called operator overloading.
- To do this, we need to create a function with a certain name that matches the name of the built-in operator.
- We saw one example of this with a constructor, which we defined by implementing the __init__ function.
- To implement the + operator, we would need to define a function called __add__

## Operator Overloading

```
class Position:
    def __init__(self, shrs=0, px=0):
        self.shares = shrs
        self.price = px

    def __add__(self, other):
        return Position(self.shares + other.shares, self.price)

    def mktValue(self):
        return self.shares * self.price

pos = Position(100, 50)
pos2 = Position(100,50)
pos3 = pos + pos2
```

## Inheritance

- Inheritance allows us to create a **class hierarchy**.
- In particular, it allows us to create classes that are modified, or specialized version of another class.
- Inheritance is based on the concept of **Base** and **Derived** classes.
  - Base Class: A higher level or more generic class with some basic functionality that is desired in the derived class.
  - Derived Class: A lower level or child class that defines a class with more specialization that the derived class.
- For example, we might have a base StochasticProcess class and derived classes for different stochastic processes (i.e. Black-Scholes, Bachelier, etc.)

## Generic Stochastic Process Class

```
import numpy as np
import pandas as pd
import math
from scipy.stats import norm

#Define the base class
class StochasticProcess:
def __init__(self, tau=1.0, S0=100.0, strike=100.0,
 rf=0.0, div=0.0):
    self.T = tau
    self.S = S0
    self.K = strike
    self.r = rf
    self.q = div

def price(self):
    print("method not defined for base class")

```

```
#Define the derived class
class BlackScholesProcess(StochasticProcess):
def __init__(self, sig=0.1, tau=1.0,
  S0=100.0, strike=100.0,
  rf=0.0, div=0.0):
    self.sigma = sig
    StochasticProcess.__init__(self, tau,
                                            S0, strike, rf, div)
def price(self):
    sigmaRtT = (self.sigma * math.sqrt(self.T))
    rSigTerm = (self.r + self.sigma * self.sigma/2.0) * self.T
    d1 = (math.log(self.S/self.K) + rSigTerm) / sigmaRtT
    d2 = d1 - sigmaRtT
    term1 = self.S * norm.cdf(d1)
    term2 = self.K * math.exp(-self.r * self.T) * norm.cdf(d2)
    return  term1 - term2

bsProcess = BlackScholesProcess(0.1)
px = bsProcess.price()
```

## Plotting in Python: matplotlib module

- The matplotlib module contains many useful plotting functions.
- It enables you to make many different types of plots, including:
  - Time Series / Line Chart via the matplotlib.pyplot.plot function
  - Histograms via the matplotlib.pyplot.hist function
  - Scatter plots via the matplotlib.pyplot.scatter function
- More info can be found here: https://matplotlib.org/gallery.html

## Plotting Examples

```
import matplotlib.pyplot as plt
import numpy as np
retsX = np.random.normal(0.0, 1.0, size=1000)
retsY = np.random.normal(0.0, 1.0, size=1000)

#time series plot
plt.plot(retsX)
plt.plot(retsY)

#histogram
plt.hist(retsX)
plt.hist(retsY)

#scatter plot
plt.scatter(retsX, retsY)
```