



**Course Number:** QF205

**Course Title:** Computing Technology for Finance

**Group:** G1-5 (Team 5)

S/N	Name	Student ID	Email
1.	Caleb TEO	01312760	caleb.teo.2017@business.smu.edu.sg
2.	Clarice Alyson HO	01354175	clariceho.2017@sis.smu.edu.sg
3.	KWON Soo Yeon	01395939	sykwon.2017@sis.smu.edu.sg
4.	Sean CHAI Shong Hee	01312737	seah.chai.2017@sis.smu.edu.sg

## TABLE OF CONTENTS

<b>Introduction to Options</b>	<b>3</b>
1.1 Call Options and Put Options	3
1.2 Types of Options	3
<b>Black Scholes Model</b>	<b>3</b>
2.1 Black Scholes Call Option Mathematical Formula	4
2.2 Assumptions for Black Scholes Model	4
2.3 Black Scholes Partial Differential Equation	4
<b>Introduction to Python</b>	<b>5</b>
3.1 Getting Started with Python	5
<b>Literals</b>	<b>8</b>
4.1 Numeric Literals	8
4.2 Point-Floats	8
<b>Assignments</b>	<b>9</b>
5.1 Naming a Variable	9
5.2 How to Assign Value to a Variable	9
<b>Operators and Expressions</b>	<b>10</b>
6.1 Rules for Operators	11
6.2 ZeroDivisionError	12
<b>Logical Operators (And, Or)</b>	<b>12</b>
<b>Comparison Operators</b>	<b>13</b>
<b>Comments</b>	<b>14</b>
9.1 Syntax	14
9.2 Uses for Comments	15
<b>Conditional Statements</b>	<b>15</b>
10.1 If Statement	15
10.2 Else Statement	16
10.3 Elif Statement	16
<b>Loops</b>	<b>17</b>
11.1 For statement	17
11.2 Nested For Loop	19
11.3 While Loop	20
<b>Lists</b>	<b>21</b>
12.1 What are containers?	21
12.3 Creating list with objects	22
12.4 Accessing items within the list with indexes	22
12.5 Looping through lists	24
12.6 Looping through list with range() and len()	25

12.7 Adding objects to existing list	26
12.8 Removing objects from existing list	26
12.9 List Comprehension	27
<b>Matrix Interpretation with Lists</b>	<b>28</b>
13.1 Converting Matrices into Lists	28
13.2 Creating Matrices in Python using List Comprehension	30
<b>Functions</b>	<b>31</b>
14.1 Naming a function	32
14.2 What functions should contain	32
14.3 How to use functions	32
14.4 Built-in Functions	33
<b>Classes</b>	<b>33</b>
15.1 Attribute References	33
15.2 Instantiation	33
<b>Import</b>	<b>34</b>
<b>Designing a Graphical User Interface</b>	<b>35</b>
17.1 PyQt5	36
17.2 Creating PyQt5 scripts	36
17.3 Understanding the PyQt5 code	38
17.4 if <code>__name__ == '__main__':</code>	39
17.5 Qt Designer	39
17.6 Connecting our PyQt5 script to the Qt ui file	42
<b>References</b>	<b>43</b>
<b>Appendix</b>	<b>44</b>
19.1 Backend Code for Black Scholes PDE Calculator	44
19.2 Backend Code for Black Scholes PDE Calculator PyQt5 Application	47

## **1. Introduction to Options**

Options are financial instruments that are derivatives based on the value of underlying assets (Downey, 2020). An option contract can take the form of a call option or put option, and these options give buyers the right to buy or sell an amount of the underlying assets at a predetermined price. Every option contract will have a purchase date and an expiration date. Buyers can choose to exercise their option based on the stipulated dates on their option contracts. Options that are not exercised will expire without any value after their expiration dates. This results in time decay, where an option will be worth less tomorrow than it is today if the price of its underlying asset is stagnant.

Options are mainly used for speculation and hedging. Speculators use options to wager on future prices of underlying assets, where they buy options with the expectation that prices of underlying assets will go up in the future (Phung, 2020). This is because the value of an option increases as the price of its underlying asset increases. On the other hand, hedging with options helps manage risks associated with investing in an asset. Investors use options to limit potential losses that might be incurred from investing in an asset (Yates, 2020). This helps reduce the risks of sudden price declines of assets.

### **1.1 Call Options and Put Options**

Call options give buyers the right, but not the obligation, to buy an amount of an asset at a specified price within a specified time period. Meanwhile, put options give holders the right, but not the obligation, to sell an amount of an asset at a set price (also known as the strike price) within a specified time period.

### **1.2 Types of Options**

There are a variety of options that are available for traders. Standard options include American-style options, European-style options, Exchange-traded options and Over The Counter (OTC) options (A Digital Blogger, 2019). Buyers can exercise American options at any time between its purchase date and expiration date. On the other hand, European options can only be exercised upon their expiration. American options thus command higher premiums as compared to European options, since they can be exercised early. An Exchange-traded option refers to a publicly traded option contract. Meanwhile, OTC options are less accessible to the public, and are typically only traded between two private parties and are customised according to the needs of the two parties.

## **2. Black Scholes Model**

The Black Scholes Model (a.k.a Black-Scholes-Merton model) is a mathematical model that is used to determine the price of an option (Kenton, 2020). It does so by estimating the variations of financial assets over time. The Black Scholes Model typically takes in five input variables, and these input variables are the strike price, the current asset price, the time to expiration, the risk-free interest rate and the volatility of the underlying asset. There are also

six assumptions that Black Scholes Model operates under. Today, the model is commonly used to determine fair prices of options.

## 2.1 Black Scholes Call Option Mathematical Formula

$$C = S_t N(d_1) - K e^{-rt} N(d_2)$$

where:

$$d_1 = \frac{\ln \frac{S_t}{K} + (r + \frac{\sigma_s^2}{2}) t}{\sigma_s \sqrt{t}}$$

and

$$d_2 = d_1 - \sigma_s \sqrt{t}$$

$K$  - Strike Price (K refers to the buying or selling price of the option when it is exercised)

$S$  - Current Stock Price (S refers to the current price of an underlying asset)

$t$  - Time to maturity (T refers to the time to expiration of the option)

$r$  - Risk-free interest rate

$N$  - A normal distribution

## 2.2 Assumptions for Black Scholes Model

The Black Scholes Model can be used when the following assumptions have been made:

- 1) The option is European and can only be exercised at expiration
- 2) No dividends are paid out during the life of the option
- 3) Markets are efficient (i.e, market movements cannot be predicted)
- 4) There are no transaction costs in buying the option
- 5) The risk-free rate and volatility of the underlying asset are known and constant
- 6) The returns on the underlying asset are normally distributed

The above assumptions also serve as limitations to the Black Scholes Model. These assumptions can lead to price deviations in the real world, where the risk-free rate and volatility may not be constant. In actuality, there are also dividend payouts for underlying assets over the life of the option.

## 2.3 Black Scholes Partial Differential Equation

To better account for dividend yields of underlying assets, Black Scholes Partial Differential Equation (PDE) should be used. The Black Scholes PDE accounts for the underlying asset's dividend yield in a continuous manner, leading to more accurate derivations of option prices. For this project, the team made use of Implicit and Explicit Finite Difference Method (FDM) to calculate Black Scholes PDE.

## How to use the report

Now that we have a basic understanding of options and what we are trying to achieve with Black Scholes PDE, we will now begin to talk about how we can automate calculations for Black Scholes PDE using Python. The main purpose of this report is to teach readers basic Python programming so that readers will have the tools needed to reconstruct the Python code for our Black Scholes PDE calculator. As this report serves as a learning guide for Python, it will not dive into the Mathematics behind Black Scholes PDE.

## 3. Introduction to Python

Python is a free-to-use interpreted, object oriented, high-level programming language that is widely used in the world today. Python's design philosophy places an emphasis on code readability, programming modularity and code reusability. For both new and experienced programmers alike, Python is very easy to pick up because of its easy to learn syntax. As a new user, Python is a great language to kickstart your programming journey.

### 3.1 Getting Started with Python

#### a. Installations

To learn Python, we will be installing Anaconda. **Anaconda®** distribution allows you to launch applications and easily manage conda packages, environments, and channels without using command-line commands. Download Anaconda (Python) using the following links and follow the download instructions.

Windows:

[https://repo.anaconda.com/archive/Anaconda3-2020.07-Windows-x86\\_64.exe](https://repo.anaconda.com/archive/Anaconda3-2020.07-Windows-x86_64.exe)

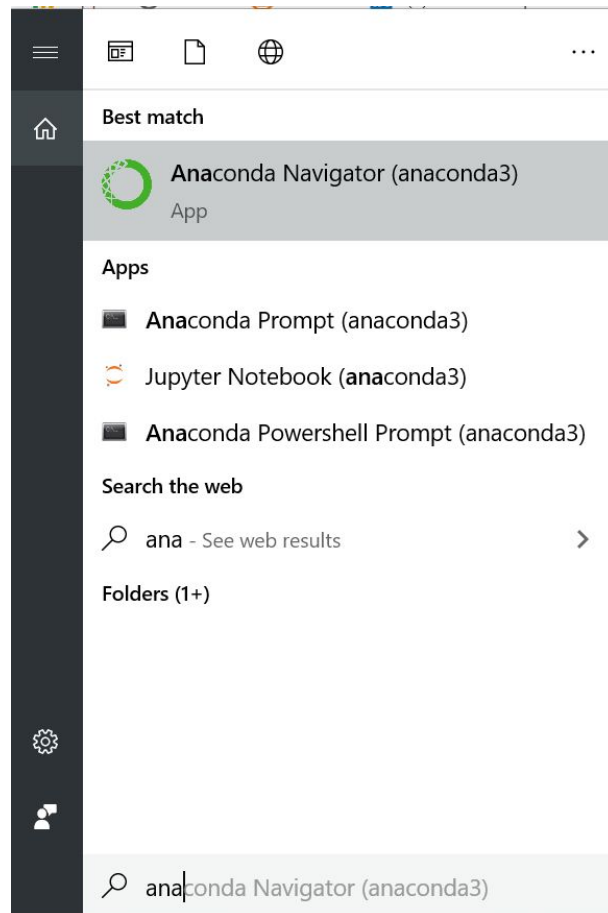
Mac

OS:

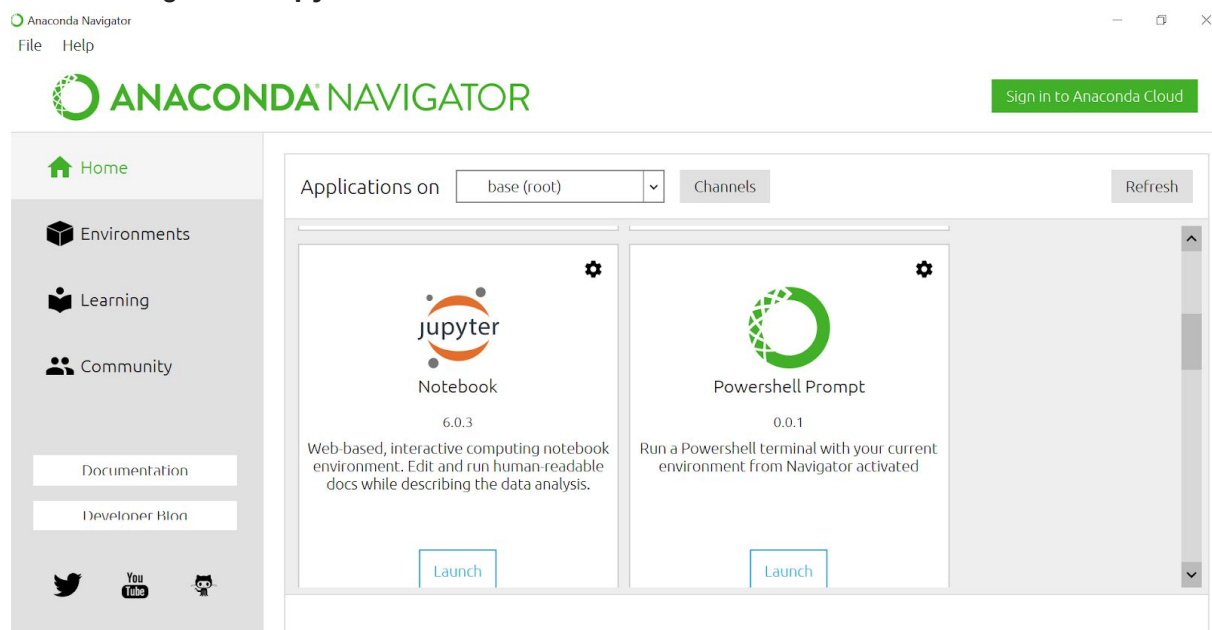
[https://repo.anaconda.com/archive/Anaconda3-2020.07-MacOSX-x86\\_64.pkg](https://repo.anaconda.com/archive/Anaconda3-2020.07-MacOSX-x86_64.pkg)

#### b. Terminal

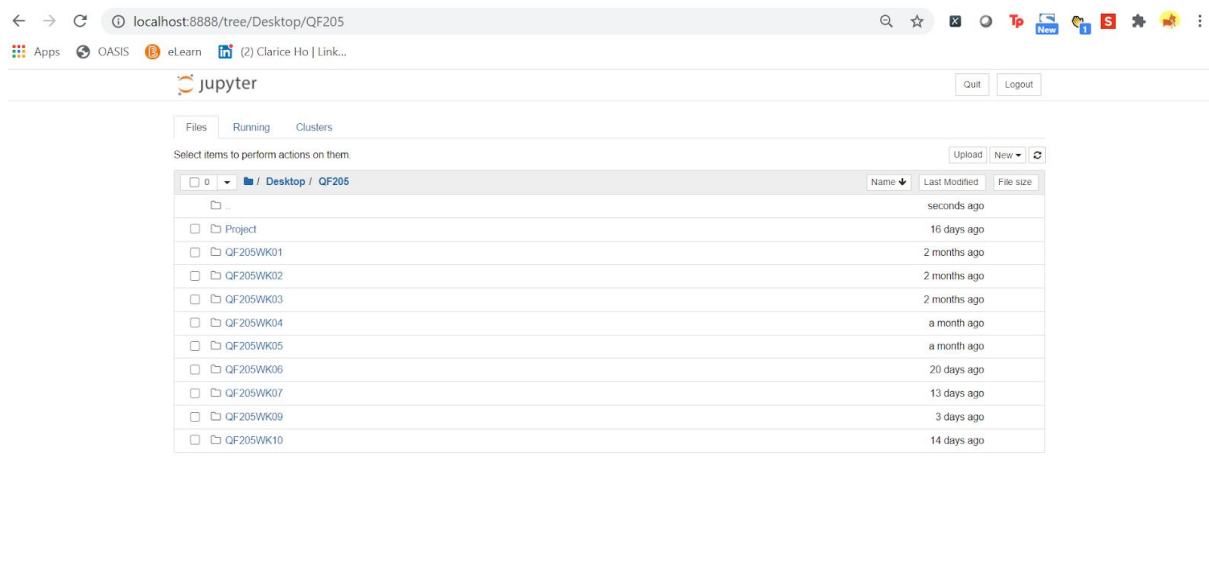
**Open** Anaconda Navigator using the Windows start menu and select **Anaconda Navigator**. The Anaconda Navigator will load.



Navigate to **Jupyter Notebook** and click **Launch**.



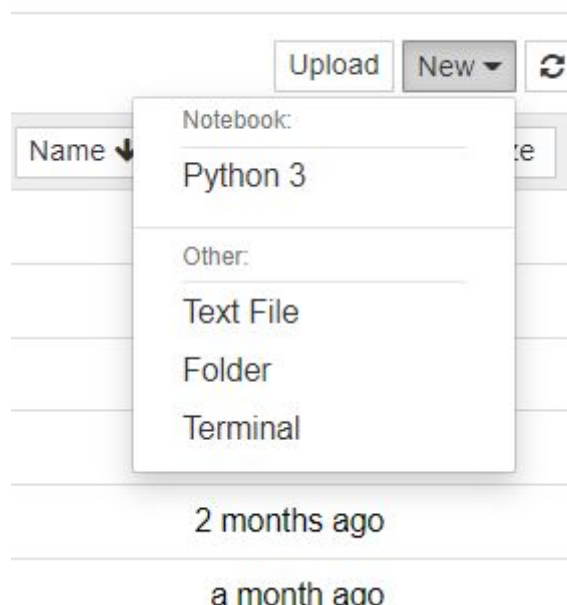
The Jupyter Notebook will open in the browser as such:



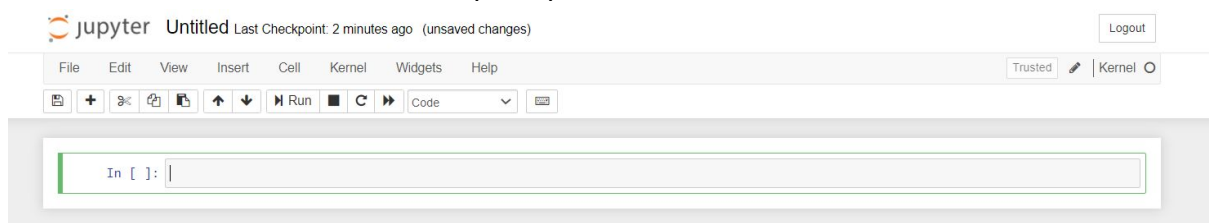
### c. Editors

We will now create a new Python File.

On the top right corner, click **New** and click **Python 3**.



You will see a new window open up as such:



Now we will type our first code. Type `print("Hello World!")` and then press Shift+Enter.



```
In [2]: print("Hello World!")
Hello World!

In [ ]:
```

You've got your first code! We will be using Jupyter Notebook to write all our codes. Jupyter notebook allows us to run your codes separately in each cell.

## 4. Literals

In Python, literals are utilised to represent values. This is useful for us as it allows us to represent mathematical variables in an equation in Python. For instance, inputs like share price ( $S$ ), strike price ( $K$ ), risk-free rate ( $r$ ), and time ( $t$ ) are required to derive a price for a European Option. Given values for these variables, we end up with something like the following:

$$S = \$30, K = \$15, r = 4\%, t = 1 \text{ year}$$

With literals, we are able to assign values to these variables and apply methods to solve for the price of a European option which would ultimately be represented by  $C$ .

According to the Python Reference Manual, literals are notations for constant values of some built-in types like numerical and string literals. Numeric literals help us input values to perform the calculations in an option pricing.

### 4.1 Numeric Literals

These literals include integers, floating point numbers and complex numbers. We are mainly concerned with the latter two in this report. Integers are numbers that are without decimal points (1, 4, 23 etc.) while floating-point numbers are numbers with decimal points (9.0, 2.59 etc.)

We utilise floating point numbers in the application and will be discussing this below. Floating point literals are used to represent variables in the Black Scholes formula such as those represented in Fig.X. The benefit of floating-point literals is that it is able to represent numbers with decimal places as opposed to an integer. There are two types of floating-point literals: point-floats and exponent floats.

### 4.2 Point-Floats

Considering the previous example, we are able to use a point-float to represent a risk-free rate of 4% by assigning  $r$  to the value 0.04. On the other hand, exponent floats utilise "e" to represent Euler's number which is approximately 2.718.

In order to use these floating-point literals, the user must follow rules to conform to the syntax required in order for Python to recognise it.

#### a. Leading Zeroes

In exponent-floats, leading zeros in the integer and exponent-parts of the literal are allowed. Thus, writing  $002e^{-03}$  is equivalent to  $2e^{-3}$ . In point-floats, leading zeros before the decimal point and up until a non-zero digit, are ignored. Thus, the first two zeros in 004.3 is ignored as they are in front of “4”.

#### b. Underscores

Underscores are allowed in forming valid floating-point literals. This is useful when we are dealing with lengthy numbers and improves readability. However, only one underscore is allowed between any two digits. Otherwise, it will be a valid floating-point literal.

Thus,  $0\_2e^{-3}$  is allowed while  $0\_2.2$  is not allowed.

## 5. Assignments

Variables in Programming are like containers. They are used to store different data types or in some cases, references to the data.

### 5.1 Naming a Variable

There is a specific set of rules when naming a variable. It can only start with a letter or underscore. However, within the variable name, it can include alphabetic letters, underscores and numbers. The variable names are case sensitive, and calling a variable with the wrong casing will result in a NameError.

```
In [2]: I_am_case_sensitive=100
        print(i_am_Case_sensitive)

-----
NameError                                Traceback (most recent call last)
<ipython-input-2-dd6d251273d3> in <module>
      1 I_am_case_sensitive=100
      2
----> 3 print(i_am_Case_sensitive)

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

(Figure 5.1)

### 5.2 How to Assign Value to a Variable

To create a variable, we assign a value to a variable we have named and Python will set an appropriate type to it. To assign a value, we use the “=” as the operator to assign a value to our named variable.

```
In [1]: sport='tennis'

print(sport)
print(type(sport))

tennis
<class 'str'>
```

(Figure 5.2)

In this example, we assign “tennis” to the variable “sport”. Hence, when we print “sport”, it gives us the value “tennis”. Python groups this under string type.

In this project, assignments were frequently used to define the variables required for option pricing as shown below.

```
delta_t = self.t/N
s_max = 2 * self.K
delta_s = s_max/M
```

(Figure 5.3)

## 6. Operators and Expressions

In Python, operators have special symbols that perform computations. The values that operators act on are called operands. This is much like what we know in math, with the operators that perform addition, subtraction, multiplication and division with +, -, \*, and /.

Operator	Example	Meaning	Result
+(unary)	+a	Unary Positive	a In other words, it doesn't really do anything. It mostly exists for the sake of completeness, to complement Unary Negation.
+(binary)	a + b	Addition	Sum of a and b
-(unary)	-a	Unary Negation	Value equal to a but opposite in sign
-(binary)	a - b	Subtraction	b subtracted from a
*	a * b	Multiplication	Product of a and b
/	a / b	Division	Quotient when a is

			divided by b. The result always has type float.
%	a % b	Modulo	Remainder when a is divided by b
//	a // b	Floor Division (also called Integer Division)	Quotient when a is divided by b, rounded to the next smallest whole number
**	a ** b	Exponentiation	a raised to the power of b

Here are some examples of how operations and expressions can be used in Python, much like how a calculator works:

```
>>> a = 4
>>> b = 3
>>> +a
4
>>> -b
-3
>>> a + b
7
>>> a - b
1
>>> a * b
12
>>> a / b
1.3333333333333333
>>> a % b
1
>>> a ** b
64
```

(Figure 6.1)

When we perform division using '/', the result has a decimal place. This is because Python converts all ints to floats before performing a division. Strings cannot be computed and would need to be converted to a float or integer first before computing.

## 6.1 Rules for Operators

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. Python follows the same precedence rules for

its mathematical operators that mathematics does. It generally follows these set of rules:

- a. If there are parentheses, they are calculated first.
- b. Then multiply and divide from left to right.
- c. In the end, plus and minus are calculated from left to right.

## 6.2 ZeroDivisionError

```
In [3]: 4/0

-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-3-221068dc2815> in <module>
----> 1 4/0

ZeroDivisionError: division by zero
```

(Figure 6.2)

Numbers cannot be divided by Zero in Python as it would throw its own error called the ZeroDivisionError.

## 7. Logical Operators (And, Or)

There are logical operators supported by the Python programming language. Logical operators are used to combine conditional statements. The types of logical operators are shown and explained in the table below.

Operator	Description	Example
<i>and</i>	If both the operands are True, condition becomes True.	(a and b) is True.
or	If any of the two operands are non-zero, the condition becomes True.	(a or b) is True.
not	Used when reversing the logical state of its operand.	Not (a and b) is False.

When using logical operators, parenthesis serves as an important factor as parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Below is the example of logical operators.

```
1 x = True
2 y = True
3 z = False
4 print(x and z)
5 print(x or y)
6 print((x and y) or z)
7 print((x or z) or y)
```

False  
True  
True  
True

(Figure 7)

## 8. Comparison Operators

Similar to arithmetic operators, there are comparison operators in Python. Comparison operators, also called Relational operators, are used to compare values. It returns a Boolean value, either True or False, according to the condition. The types of comparison operators are shown and explained in the table below.

Operator	Meaning
>	Greater than.  Returns True if left operand is greater than the right.  Otherwise, returns False.
<	Less than.  Returns True if left operand is less than the right.  Otherwise, returns False.
>=	Greater or equals to.  Returns True if left operand is greater than or equals to the right.

<code>&lt;=</code>	<p>Less or equals to.</p> <p>Returns True if left operand is less than or equals to the right.</p>
<code>==</code>	<p>Equals to.</p> <p>Returns True if both operands are equal (the type must be the same as well).</p>
<code>!=</code>	<p>Not equal to.</p> <p>Returns True if the operands are not equal.</p>

Below is a simple example.

```

1  x = 10
2  y = 15
3  print(x < y)
4  print(x > y)
5  print(x >= y)
6  print(x <= y)
7  print(x == y)
8  print(x != y)

```

True  
False  
False  
True  
False  
True

(Figure 8)

## 9. Comments

### 9.1 Syntax

Comments in Python begin with a hash mark (#) and a whitespace character and continue to the end of the line.

```
# Print "Hello, World!" to console
print("Hello, World!")
```

(Figure 9.1)

Because comments are not executed by the computer, we will not see it in the output when the code is run. Comments are meant for humans to read to denote what the lines of code mean, and not for computers to execute.

## 9.2 Uses for Comments

- a. Block Comments: to understand what the next few lines of code mean and the function they carry out
- b. Inline Comments: Should be used sparingly and only when necessary and can provide helpful guidance for the person reading the program
- c. Commenting out code for testing

## 10. Conditional Statements

### 10.1 *If* Statement

Conditional statements are used for making decisions using conditions. The most fundamental conditional statement is the *if* statement. The syntax is as follows:

```
if (expression):  
    (statement)
```

The expression is evaluated in a Boolean context. The statement is a valid Python statement, which must be indented. If the expression is True, then the statement will be executed and otherwise, the statement will be skipped or not executed. Note that the *if* statement must be followed by a colon in order to be executed without Syntax error. Under one expression, multiple statements can be written, and those statements will be treated as one block of code. Because of this, the indentation plays a crucial role in Python programming. Let's take a simple example for better understanding.



```

1 x = 0
2 y = 5
3 if x < 6:
4     print("yes")
5     print("y is greater than x")
6

```

```

yes
y is greater than x

```

(Figure 10.1)

In this code, the expression is that x is less than y. Since x is assigned 0, which is less than the value assigned to y, 6, the code of block containing multiple statements will be executed.

## 10.2 Else Statement

Additionally, an *else* statement can be used to make another condition. *Else* statement is used to evaluate a condition and take one path if it is true but specify an alternative path if it is not. There can only be at most one *else* statement. The syntax is as follows:

```

if (expression):
    (statement)
else:
    (statement)

```

If expression is True, the first suite will be executed while the second is skipped. If False, the first suite is skipped and the second will be executed and the execution will resume after that. Note that *else* statement is followed by a colon and indented statement just like the *if* statement. Below is a simple example.

In this example, because x is smaller than 50, the second suite is skipped and the first suite in the *else* statement is executed, printing "x is small" as output.

## 10.3 Elif Statement

There is also an *elif* statement which is a way of saying "if the previous conditions were not true, then try this condition". Similar to the *else* statement, the *elif* statement is option but unlike *else*, there can be an arbitrary number of *elif* statements following an *if*. The implementation of order of conditional statements is: *if* -> *elif* -> *else*. The syntax is as follows:

```

if (expression_1):
    (statement_1)

```

```
elif (expression_2):  
    (statement_2)  
elif (expression_3):  
    (statement_3)  
else:  
    (statement_4)
```

As mentioned, the number of *elif* statements is arbitrary and there can be more or less number of *elif* statements as compared to the example given. Below is an example that implements all conditional statements.

In this example, because *a* is greater than *b*, the *if* and *elif* statements are skipped and only the *else* statement is executed.

```
1 a = 200  
2 b = 33  
3 if b > a:  
4     print("b is greater than a")  
5 elif a == b:  
6     print("a and b are equal")  
7 else:  
8     print("a is greater than b")
```

a is greater than b

(Figure 10.3)

In our application, conditional statements are heavily used, especially in the GUI file to check which option pricing approach is selected by the user.

## 11. Loops

### 11.1 For statement

For statement is used when you want to repeat a block of code for a fixed number of times in order. Also known as for-loop, it can be used to iterate various types – string, list, tuple, set or dictionary. When forming loops, it is important to pay attention to the indentation as the iteration will only occur for the statements within the loop.

Syntax is as follows:

```
For (variable) in (iterable):  
    (Statements)
```

Variable takes on the value of the next element in the iterable each time through the loop. It is suggested to give variable name a meaningful one. Iterable is a collection of objects – a list or tuple, for instance. Statements are the indented block of code that are executed. When the number of iterations to go through is specified, (iterable) will be replaced with `range(arguments)` where the arguments of the range constructor are integers. Range is a built-in function used to restrict the range of the loop. See the example below.

In this example, the variable is `i`, iterable is `range(5)` and statement is `print(i)`. If you want to specify the starting number, stopping number and the step, you could have 3 arguments in the range constructor. Take note that the value of stop is not included. Let's see a simple example.

```
1 for i in range(5):
2     print(i)
```

0  
1  
2  
3  
4

(Figure 11.1.1)

```
1 for i in range(2, 11, 2):
2     print(i)
```

2  
4  
6  
8  
10

(Figure 11.1.2)

For loop was often used in our application to find out the call and put option value within the given range as shown below.

```
for j in range(M + 1):
    call[N][j] = max(j * delta_s - self.K, 0)
    put[N][j] = max(self.K - j * delta_s, 0)
```

(Figure 11.1.3)

## 11.2 Nested For Loop

Nested loop is a loop that occurs within another loop. Python first encounters the outer loop, executes its first iteration. Then, it will then trigger the inner, nested loop and run to completion. Finally, the program returns back to the outer loop, completing the second iteration and triggering the nested loop again. Then, the nested loop runs to completion and the program returns back to the top of the outer loop until the sequence is completed. The syntax is as follows:

```
for (variable_1) in (iterable_1):  
  
    (Statement_1)  
  
    for (variable_2) in (iterable_2):  
  
        (Statement_2)
```

Let's implement a nested *for* loop for better understanding. In this example, the outer loop will iterate through a list of strings called *my\_str* and the inner loop will iterate through a list of integers called *my\_int*.

```
1 my_str = ['a', 'b', 'c']  
2 my_int = [1, 2, 3]  
3 for char in my_str:  
4     print(char)  
5     for i in my_int:  
6         print(i)
```

a  
1  
2  
3  
b  
1  
2  
3  
c  
1  
2  
3

(Figure 11.2.1)

The result shows that the program completes the first iteration of the outer loop by printing a, then triggers the completion of the inner loop, printing 1, 2, 3 consecutively. Once the inner loop is completed, the program returns to the outer loop printing b, then the inner loop again, printing 1, 2, 3 consecutively. Finally, it goes back to the outer loop printing c, then the inner loop, printing 1, 2, 3 consecutively.

Similar to *for* statement mentioned previously, nested *for* loop is also often used in our project to find the option pricing - specifically to formulate a matrix.

```
#3. For i=N-1, N-2,...,1,0, repeat 3.1 and 3.2
for i in range(N - 1, -1, -1):
    #3.1. Compute vector Fi=A*Fi+1
    for j in range(1, M):
        a = 0.5 * delta_t * (self.sigma * self.sigma * j * j - (self.r - self.q) * j)
        b = 1 - delta_t * (self.sigma * self.sigma * j * j + self.r)
        c = 0.5 * delta_t * (self.sigma * self.sigma * j * j + (self.r - self.q) * j)
        call[i][j] = (a * call[i+1][j-1] + b * call[i+1][j] + c * call[i+1][j+1])
        put[i][j] = (a * put[i+1][j-1] + b * put[i+1][j] + c * put[i+1][j+1])
    #3.2. Compute vector Fi
    call[i][0] = 0
    call[i][M] = s_max - self.K * exp(-self.r * (N-i) * delta_t)
    put[i][0] = self.K * exp(-self.r * (N-i) * delta_t)
    put[i][M] = 0
```

(Figure 11.2.2)

### 11.3 While Loop

Another type of loop is the *while* statement, also known as *while* loop. *While* loop can be imagined as a repeating conditional statement. After an *if* statement, the program continues to execute a block of code, but in a *while* loop, the program jumps back to the start of the while statement until the condition is False. Unlike *for* loop that executes a block of code a certain number of times, *while* loops are conditionally based, hence there is no need to know the number of times the code is repeated. Below shows the syntax of while loop.

while (a condition is True):

(Statement)

Let's now take a loop of a simple example of *while* loop.

```
1 i = 1
2 while i < 6:
3     print(i)
4     i += 1
```

1  
2  
3  
4  
5

(Figure 11.3)

In this block of code, the condition is that *i* is smaller than 6. Hence, the *while* loop will iterate till this condition is False. This means that the loop will be iterated 5 times

until *i* is greater or equal to 6 and in each loop, the value of *i* increments by 1 from 1 to 2 to 3 to 4 to 5.

Although *while* statement was not used in our project, it is a very crucial and useful tool when coding in Python.

## 12. Lists

### 12.1 What are containers?

In Python, data can be stored in objects called containers. Containers are objects that can hold an arbitrary number of other objects. Containers are iterable. This means that we can loop through a container to return each individual object inside the container. There are four data collection types: *list*, *tuple*, *set*, *dictionary*. These data collection types are containers that can hold other objects in them.

Imagine you now have an object *x* but you are unsure if it is a Container. You can check if *x* is a container by using the following code:

```
In [ ]: from collections import abc
        isinstance(x, abc.Container)
```

(Figure 12.1)

This code imports abstract base classes (*abc*) from a module called *collections*. *abc.Container* lets us test whether *x* is a container by checking if it provides the `__contains__()` method.

### 12.2 What are lists?

A list is a collection which is ordered and changeable. Lists are written at square brackets `[]` in Python.

In the following code, we will be creating a new list called *mylist*. A list is denoted by two square brackets, and elements within the list will be placed in the square brackets.

```
In [1]: mylist = []
        type(mylist)

Out[1]: list
```

(Figure 12.2)

The `type()` function tells us the type of the object that is within its parentheses. In this case, `type(mylist)` tells us that *mylist* is an object of type *list*.

### 12.3 Creating list with objects

Lists are mainly used as containers to store other data objects. Here is an example of how lists look like when they have objects stored within them.

```
['object1', 'object2', 1, 2]
```

(Figure 12.3.1)

This is a list that contains 4 objects. The first object is a string 'object1'. The second object is a string 'object2'. The third and fourth objects are integers 1 and 2 respectively.

How do we create a list of objects? We can do it quite simply by inserting objects inside the list, where each object is separated by a comma (,) inside the square brackets of the list.

```
In [5]: mylist = ["object1", "object2", 1, 2]
```

(Figure 12.3.2)

This code creates a list called *mylist* with 4 objects: 'object1', 'object2', 1, 2.

## 12.4 Accessing items within the list with indexes

Objects within the list have index numbers which corresponds to their order within the list. Index numbers always start with 0. In a list, an object with index 0 signifies that it is the first object within the list.

```
mylist = ["object1", "object2", 1, 2]
```

(Figure 12.4.1)

Conversely, an object with index -1 signifies that it is the last object of the list.

```
mylist = ["object1", "object2", 1, 2]
```

(Figure 12.4.2)

We can access objects within a list through indexing. Indexing can be done by indicating the index of an object within square brackets after calling the list.

```
In [7]: mylist[0]  
Out[7]: 'object1'
```

(Figure 12.4.3)

The above code calls the first object in the *mylist*. To access the second object, we can do so by replacing 0 within the square brackets with 1 (*mylist[1]*). If we want to access the last object in the list, we can do so by replacing the index number with -1 (*mylist[-1]*).

We can also access multiple objects within a list by specifying a *range()* of indexes. Python will return the objects in the list based on the starting and ending index.

```
In [8]: mylist[1:3]
Out[8]: ['object2', 1]
```

(Figure 12.4.4)

The above code returns all objects in the list, starting with the second object (index 1) and ending with the third object (index 2). It is important to note that although we specified our range of indexes as 1:3, Python will only return objects starting from index 1 (included) and ending with index 3 (excluded).

Indexing can also be done without specifying a start or an end.

```
In [9]: mylist[:3]
Out[9]: ['object1', 'object2', 1]
```

(Figure 12.4.5)

The above code returns all objects from the start of the list and ends with the third object (index 2) of the list.

```
In [10]: mylist[1:]
Out[10]: ['object2', 1, 2]
```

(Figure 12.4.6)

The above code returns all objects starting from the second object (index 1) of the list up till the last object of the list.

```
return [Vcall, Vput]
```

(Figure 12.4.7)

In our code, we used lists to store the values returned from our explicit and implicit functions. Functions will be explained in detail in section 14.

## 12.5 Looping through lists

As we know, lists are containers that are iterable. What this means is that we can loop through lists to access each individual object within the list. In the above



sections, we've learnt the syntax for looping through iterables. Let's apply what we learnt on lists.

When looping through lists, we follow the same syntax used for loops.

```
for (variable_1) in (iterable_1):  
  
    (Statement_1)
```

Here, we can replace `variable_1` with a variable name that we like and we will replace `iterable_1` with the name of our list. Let's start off by using *mylist*, which contains 4 objects: 'object1', 'object2', 1, 2.

```
In [11]: for item in mylist:  
         print(item)  
  
object1  
object2  
1  
2
```

(Figure 12.5)

The above code illustrates how we loop through *mylist*. In this for loop, we assign an object in the list to the variable *item*. In the first iteration, *item* will take the value of the first object in the list, which is 'object1'. In the second iteration, *item* will take the value of the second object in the list, which is 'object2'.

Take note of the indentation in the second line of our code. Indentations help Python interpret statements within the for loop. Since *print(item)* is indented, Python will run *print(item)* for every iteration of the for loop, which is why we see all 4 objects in *mylist* being printed in our output.

## 12.6 Looping through list with `range()` and `len()`

Another way to loop through a list is with the use of Python's in-built *range()* and *len()* functions.

Range is a function that is used to generate a sequence of numbers. It returns a range object that is iterable. To elaborate, let's try out the following code:

```
In [16]: range(10)  
  
Out[16]: range(0, 10)
```

(Figure 12.6.1)

*range(10)* gives us a range object with values from 0 to 9 (excluding 10).

Length is a function that gives us the value of the length of the object of concern. It returns an integer based on the length of the object.

```
In [19]: len(mylist)
Out[19]: 4
```

(Figure 12.6.2)

Using the `len()` function for `mylist` will tell us that we have 4 objects in the list, hence we say that `mylist` has a length of 4.

What happens when we combine the two code blocks above? We get: `range(len(mylist))`. This effectively returns us a range object with values from 0 to 3 (excluding 4), since `len(mylist)` gives us a value of 4. With the use of `range()` and `len()`, we can loop through a list based on indexes.

```
In [20]: for index in range(len(mylist)):
        print(mylist[index])

object1
object2
1
2
```

(Figure 12.6.3)

Let us dissect the code above. In the first line, we are creating a range object for the length of `mylist`. We next apply a for loop on the iterable range object that contains a range of values from 0 to 3. As we loop through the range object, `index` will take the values 0, 1, 2 and 3 for each iteration of the loop. We then access each individual object inside `mylist` using index numbers. The first object can be accessed with `mylist[0]`, which is what happens in the first iteration. In the first iteration, `index` takes the value of 0. In the same iteration, we print `mylist[index]`, and since `index` has the value of 0, we are essentially printing `mylist[0]` for the first iteration. We thus get 'object1' for the first iteration.

## 12.7 Adding objects to existing list

Now that we know how to create a list and access the objects in a list, here comes the question: how do we add objects to an existing list without making a new one? To do so, we can make use of Python's in-built function `append()`.

```
In [7]: mylist.append("new_item")
        mylist

Out[7]: ['object1', 'object2', 1, 2, 'new_item']
```

(Figure 12.7.1)

In the above code, we added a new object called "`new_item`" into `mylist` using the `append()` function. We insert our new object inside the parentheses of the `append()` function, and our new object will be inserted into the end of our list. As we can see from our output, "`new_item`" is now the last object in `mylist`.

We can also insert objects into our list using the function `insert`.

```
In [8]: mylist.insert(0, "this_item")
        mylist
Out[8]: ['this_item', 'object1', 'object2', 1, 2, 'new_item']
```

(Figure 12.7.2)

The *insert* function takes in two parameters. The first parameter is the index where the object is to be inserted. The second parameter is the object that is to be inserted into the list. Here, we inserted “*this\_item*” into *mylist* at index 0.

## 12.8 Removing objects from existing list

There are several ways we can remove objects from a list. We will introduce two of the most common methods to do so. The first method makes use of the function *pop()*.

```
In [12]: print(mylist.pop())
        mylist
        new_item
Out[12]: ['this_item', 'object1', 'object2', 1, 2]
```

(Figure 12.8.1)

Calling the *pop()* function without any parameter removes the last object in the list. Here, we see that using *pop()* on *mylist* removes “*new\_item*”.

```
In [13]: print(mylist.pop(0))
        mylist
        this_item
Out[13]: ['object1', 'object2', 1, 2]
```

(Figure 12.8.2)

We can also specify an index in the list that we want to remove. Specifying *pop(0)* removes the object at index 0 of the list. Here, *pop(0)* removes “*this\_item*”, which is at index 0 of *mylist*.

What if we are unsure of the index of the object we want removed from our list? What if we only have the value of the object? Well, here is where the function *remove()* comes in handy.

```
In [16]: 1 mylist.remove('object1')
        2 mylist
Out[16]: ['object2', 1, 2]
```

(Figure 12.8.3)

As we can see, specifying the value of the object in the *remove()* function will remove the object with the exact same value in the list. *remove('object1')* will remove ‘*object1*’ in *mylist*.

## 12.9 List Comprehension

List comprehensions are used to create new lists from iterables. A list comprehension will return a list with objects resulting from evaluating the iterables within the list comprehension.

Supposed we have the following two lists that tells us the names of people who attended eventA and eventB.

```
eventA = ["Alice", "Bob", "Charlie", "Dylan"]
eventB = ["Anita", "Bob", "Dylan", "Josh"]
```

We want to create a new list called *eventAB* which tells us the names of people who attended both eventA and eventB. To do so, we will have to loop through eventA and eventB to determine if the name in eventA is also in eventB. If it is, we add the name into a new list.

```
In [20]: eventA = ["Alice", "Bob", "Charlie", "Dylan"]
        eventB = ["Anita", "Bob", "Dylan", "Josh"]

        eventAB = []
        for personA in eventA:
            for personB in eventB:
                if personA == personB:
                    eventAB.append(personA)
        eventAB

Out[20]: ['Bob', 'Dylan']
```

(Figure 12.9.1)

The above code correctly identifies that “Bob” and “Dylan” attended both eventA and eventB, and their names are added to the new list eventAB. However, this code is not easily understandable for people reading the code. We can simplify the above code using list comprehension.

```
In [39]: eventA = ["Alice", "Bob", "Charlie", "Dylan"]
        eventB = ["Anita", "Bob", "Dylan", "Josh"]

        eventAB = [personA for personB in eventB for personA in eventA if personA == personB]
        eventAB

Out[39]: ['Bob', 'Dylan']
```

(Figure 12.9.2)

In the above code, we compressed 5 lines of code into 1 line of code. We achieve the same list eventAB which contains the names of people who attended both eventA and eventB.

What exactly is list comprehension? As we can see, list comprehension creates a new list containing resulting objects obtained from evaluating the iterables within the list comprehension. In the code used above, we have the following list comprehension:

```
eventAB = [ personA for personB in eventB for personA in eventA if personA == personB ]
```

Starting from left to right, we first define the name of the list that we create from the list comprehension, *eventAB*. Every list comprehension has to start off with square brackets [ ]. This helps us create the new list *eventAB*. Next, we specify the variable *personA* that will be added to our new list. Once that is done, we continue with our for loop.

*for personA in eventA:*  
*for personB in eventB:*

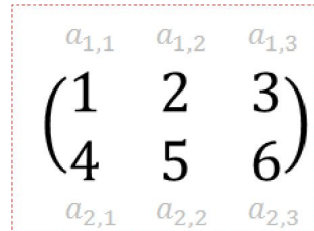
We used two for loops in the first example. In list comprehension, the for loop on the right will take precedence over the for loop on the left. Hence, we have: *for personB in eventB for personA in eventA*. In this statement, Python will loop through eventB for every iteration of the for loop in eventA. We then include our statement at the end of our list comprehension, just as in the first example. Thus we have:

*eventAB = [ personA for personB in eventB for personA in eventA if personA == personB ]*

## 13. Matrix Interpretation with Lists

### 13.1 Converting Matrices into Lists

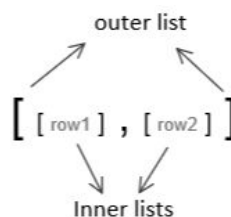
In this section, we will take a look at how we can interpret matrices with Python.



$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix}$$

(Figure 13.1)

Here, we have a matrix that has 2 rows and 3 columns. To interpret the above matrix with Python, we will be making use of nested lists (a list of lists). To do so, we first create a list to store inner lists [ ]. Next, we add inner lists into the outer list, such that every inner list constitutes a row in the matrix.

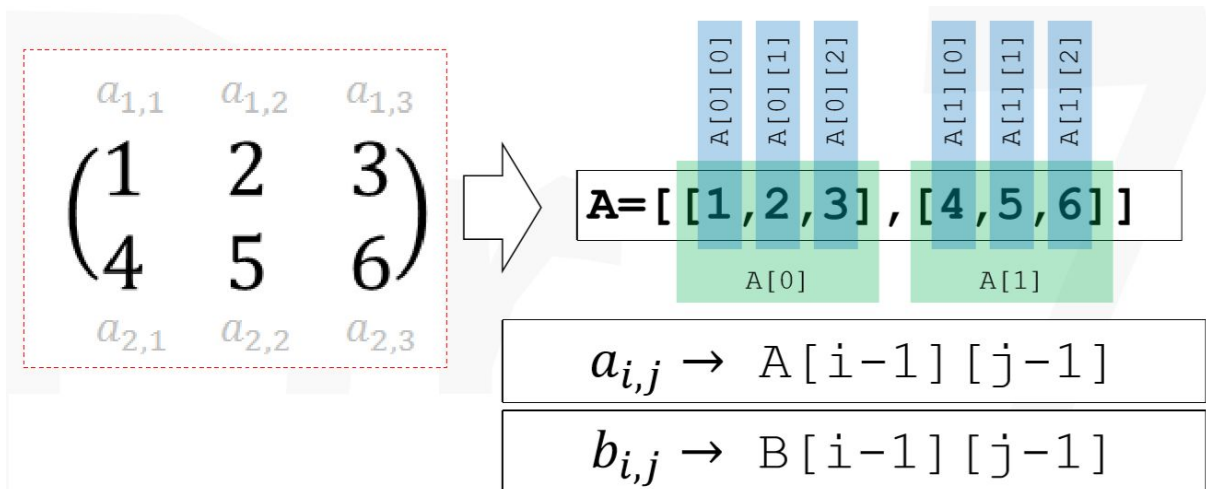


(Figure 13.1.1)

In the illustration above, we can see that the first inner list will make up the first row in our matrix, and the second inner list will make up the second row of our matrix. Next, we add in the values in the matrix into our inner lists. In the matrix above, the first row

has values 1, 2 and 3. Hence, we add into the first inner list values 1, 2 and 3: [1,2,3], []]

We do the same for the second row of our matrix and add in the values 4, 5 and 6 into the second inner list to get our resultant matrix in Python: A = [ [1,2,3], [4,5,6] ]. Viola! We have now interpreted a matrix into Python.



(Figure 13.1.2)

To access row 1 of our matrix, we access inner list 1, which is the first object at index 0 of our matrix A in Python: A[0]. To access row 2 of our matrix, we access inner list 2, which is the second object at index 1 of our matrix A in Python: A[1].

```
In [31]: A = [ [1,2,3], [4,5,6] ]
          print(A[0])
          print(A[1])

          [1, 2, 3]
          [4, 5, 6]
```

(Figure 13.1.3)

As we can see, A[0] gives a list with values [1, 2, 3], which corresponds to the first row of our matrix. A[1] gives a list with values [4, 5, 6], which corresponds to the second row of our matrix. In the same way, we can access each individual value of our matrix by accessing the individual values in our inner lists.

$$A_{\text{row}, \text{column}}$$

(Figure 13.1.4)

Mathematically, we can access the value 2 in our matrix by looking at row 1 and column 2 of our matrix. In matrix notation, this equates to  $A_{1,2}$ . In Python, we can access the value 2 of our matrix by looking at the second value of the first inner list: A[0,1]. Do you see a pattern?

$$a_{i,j} \rightarrow A[i-1][j-1]$$

(Figure 13.1.5)

In Python, we access the rows and columns in a matrix the same way as we do in mathematical notation, but we have to minus 1 from the row and column values. This is because the first object in a Python list always takes the index number of 0 instead 1, hence we compensate by deducting 1.

```
for j in range(M + 1):
    call[N][j] = max(j * delta_s - self.K, 0)
    put[N][j] = max(self.K - j * delta_s, 0)
```

(Figure 13.1.6)

Based on the formulas for the explicit and implicit method, we access the  $j$  column on row  $N$  in our call and put matrices in the code for Black Scholes PDE Calculator.

### 13.2 Creating Matrices in Python using List Comprehension

Manually typing out the code to create a matrix list in Python can be laborious. What if we have a matrix with 100 rows and 100 columns? Typing out 100 inner lists with 100 values in each of the inner lists will take up a lot of time. Luckily, there is a more efficient way of creating lists that represent matrices in Python.

```
In [43]: matrixA = []
         for rows in range(3):
             inner_list = []
             for columns in range(4):
                 inner_list.append(0)
             matrixA.append(inner_list)
         matrixA
Out[43]: [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

(Figure 13.2.1)

We can create matrices in Python using for loops. We create a specified number of new columns for every row for a specified number of rows, and this specified number is determined by the value we insert into the *range()* function. In the above code block, we have created a 3x4 matrix (a matrix with 3 rows and 4 columns). Suppose we want to create a matrix with 20 rows and 30 columns. We can do so simply by changing *range(3)* in the first for loop for rows to *range(20)* and *range(4)* in the second for loop for columns to *range(30)*.

In Section 10.9, we learnt how to use list comprehension to create a new list of objects. Now, we will use list comprehension to create a new list of lists that will represent a matrix. Let's start by creating a 3x4 matrix where all values in the matrix are 0.

```
In [48]: matrixA = [[0 for columns in range(4)] for rows in range(3)]
         matrixA
Out[48]: [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

(Figure 13.2.2)

In the above code, we used list comprehension to achieve the same matrix as we did in the first example. We can see that this is a much more efficient way of creating matrices. As we learnt in Section 10.9, the rightmost for loop will take precedence over the leftmost for loop. Hence, the above list comprehension will 4 columns with value 0 for every row, and will do so 3 times since we have specified that we want 3 rows.

```
call = [[0 for j in range(M+1)] for x in range(N+1)]  
put = [[0 for j in range(M+1)] for x in range(N+1)]
```

(Figure 13.2.3)

We made use of this concept in our code for Black Scholes PDE Calculator. We initialized empty matrices for call and put options made up of N+1 rows and M+1 columns before proceeding to coding out the formula for explicit and implicit methods.

## 14. Functions

Functions are an essential part of Python programming which make codes easier to read. They define a block of code and allow this block of code to be reused in a single line without having to type out the whole block of code again.

Any variable defined in a function cannot be retrieved outside of the function.

### 14.1 Naming a function

This follows the same set of rules as naming a variable. It can only start with a letter or underscore. Within the function name, it can include alphabetic letters, underscores and numbers. The function names are case sensitive.

### 14.2 What functions should contain

- a. The keyword *def*  
This defines the function.
- b. Parameters (to define what values to take in)  
A function can take in any number of parameters.
- c. ":"  
This denotes the start of the function.
- d. The *return* statement



A function can return value. The keyword *return* carries out this function and is written at the end of the function. This is the only value from the function that can be retrieved outside of the function. When there is a value that can be stored in a variable within the function and returned, we call this a returned function value.

### 14.3 How to use functions

This function takes in the parameters “name”, “school” and “fav\_drink” and returns a statement which is pieced together with this information.

```
In [17]: def fill_in_sentence(name,school,fav_drink):  
  
        sentence="Hi, my name is "+str(name)+\  
        " and I am currently studying in " +\  
        str(school)+". I love drinking "+str(fav_drink)  
  
        return sentence  
  
print(fill_in_sentence('john','SMU','coffee'))
```

Hi, my name is john and I am currently studying in SMU. I love drinking coffee

(Figure 14.3.1)

After defining the function and its parameters and the return statement, call the function by using the *print()* function with its parameters filled. It will return the value specified in the return statement of the function.

### 14.4 Built-in Functions

There are built in functions which Python has made which are very commonly used and helpful in shortening our code. These are some of the more common functions:

Built-in Functions	Description
<i>len()</i>	Returns the length of a string or list.
<i>print()</i>	Prints an object to the screen. Important to note that the object will be converted to a string when printed.
<i>round()</i>	Rounds a number to a specified number of decimals, if left out it will round it to zero decimals.
<i>sum()</i>	Calculates the sum of a list.
<i>type()</i>	Returns the type of a variable.

<code>max()</code>	Returns the largest number of a list.
--------------------	---------------------------------------

## 15. Classes

In English we have classes of things. For instance, we put animals into various classes like mammals, amphibians, reptiles and birds. In python, classes are similar. They are used to bundle data and functions. Creating a new class creates a new type of object, allowing new instances of that type to be made. An instance would be a member of a class just like how seagulls are instances of birds.

Class objects support 2 types of operations: attribute references and instantiation.

### 15.1 Attribute References

These are used to access and modify class members. Attributes of a class can be referenced using the name of the class followed by a “.” and the attribute. For instance, to refer to the “wing” attribute of a class named “bird”, we simply use “bird.wing”. Valid attribute names are all names in the class namespace when the class is created.

### 15.2 Instantiation

These are used to create a new instance of a class. Class instantiation uses function notation (e.g. `seagull = bird( )`). Many classes create objects with instances customized to a specific initial state. Thus, they would contain a special method named `__init__( )`. When a class defines its `__init__( )` method, class instantiation automatically invokes the `__init__( )` method for the new instance. In general, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of that class.

To illustrate these concepts, consider the following example.

```
In [1]: class Manor:
        species = 'Animal'
        def __init__(self, name):
            self.name = name
        horse = Manor('Boxer')
        donkey = Manor('Benjamin')

        print(horse.species, donkey.species)
        print(horse.name, donkey.name)

Animal Animal
Boxer Benjamin
```

(Figure 15.2.1)

In the figure above, there are two instances of the class “Manor”. For both instances, the attributes of the class are the same as can be seen in the way “horse.species” and “donkey.species” return “Animal”. However, instance variables are unique and

therefore different as can be seen in how “horse.name” will return “Boxer” and “donkey.name” returns “Benjamin”.

Now we will see how this is manifested in our application. In Fig. 15.2.2, there are two classes. In the “Option” class, there are 6 assignment statements in the `__init__()` method. Thus, when coding the Implicit method under “European Options”, we do not have to specify all the values for  $S$ ,  $K$ ,  $R$ ,  $T$  and  $\sigma$ . This is because the 6 parameters can be referred to by using “self” which is linked to the earlier assignments under the “Options” class. Thus, we only need to provide values for  $M$  and  $N$ .

```
class blackscholes():
    def __init__(self, S, t, K, r, sigma, q):
        self.S = float(S)
        self.t = float(t)
        self.K = float(K)
        self.r = float(r)/100
        self.sigma = float(sigma)
        self.q = float(q)

    def explicitfdm(self, M, N):
        delta_t = self.t/N
        s_max = 2 * self.K
        delta_s = s_max/M
```

(Figure 15.2.2)

This shows how classes can make things convenient by reducing the amount of times we would have to assign values to parameters under the different methods. Otherwise, we would have to write something like what we have in the Fig 15.2.3. This would have to be repeated for the explicit method as well.

```
def explicitfdm(S, K, r, q, T, sigma, M, N):
    delta_t = self.t/N
    s_max = 2 * self.K
    delta_s = s_max/M
```

(Figure 15.2.3)

## 16. Import

Importing statements lets the user access codes from other modules. This is an example of the way we used import in our application:

```
1 import sys
2 from PyQt5.Qtwidgets import QApplication, QWidget
3 app = QApplication(sys.argv)
4 w = QWidget()
5 w.show()
6 sys.exit(app.exec_())
```

(Figure 16.1.1)

For example, as shown below, we are importing `datetime` so we can use its functions from the module **`datetime`**, simply by writing one line of code. There are a few ways of importing code, but we will be going through two common examples.

**Import `datetime`:** This imports `datetime` and creates a reference to that namespace. To access the functions within `datetime`, the name of the module has to be referenced each time (`datetime.function()`). It makes sense to use this method when using more than one function from this module as it allows us to keep track of which module this is from and hence its purpose.

```
In [6]: import datetime

        today=datetime.date(2020,11,11)

        print(today)

2020-11-11
```

(Figure 16.2.1)

**From `datetime` import `date`:** This will import the module and create a reference to a specific function within the module. In this example, the **module is `datetime`** and the **function is `date`**. This enables us to use the called function but not the others in the module. This is more practical to use when we are only using one or two functions from the module, as the user would not need to write out the full line (`datetime.function()`) every time, but may get confused as to what the purpose of the function is if many functions are referenced to this module without being clearly written out.

```
In [8]: from datetime import date

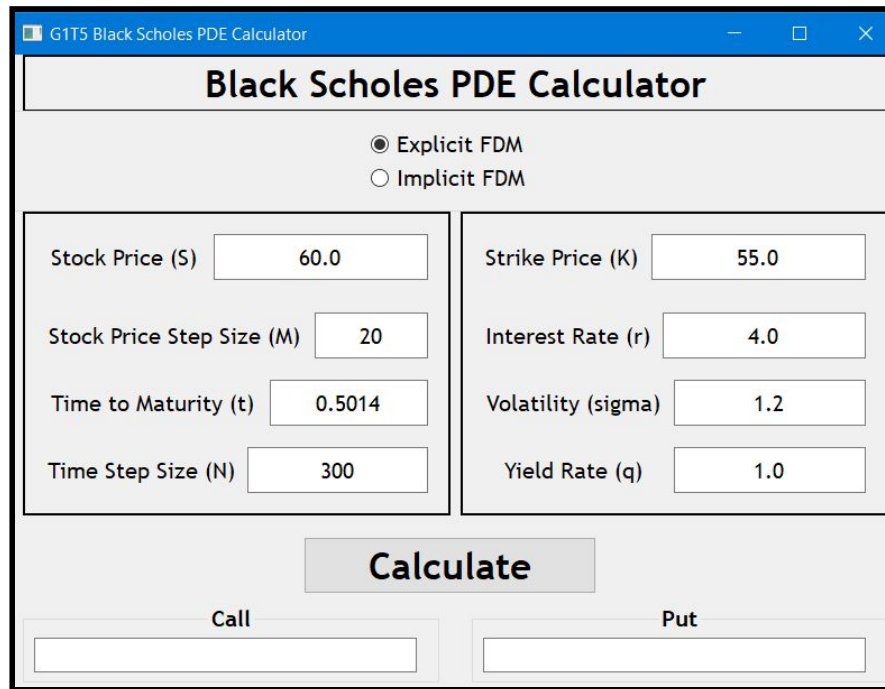
        today=date(2020,11,11)
        print(today)

2020-11-11
```

(Figure 16.2.2)

## 17. Designing a Graphical User Interface

A graphical user interface (GUI) is a system of interactive graphical components for a computer software (Hope, 2019). It lets users easily interact with the software without having to worry about what goes on in the back-end code of the software.



(Figure 17: GUI of Black Scholes PDE Calculator)

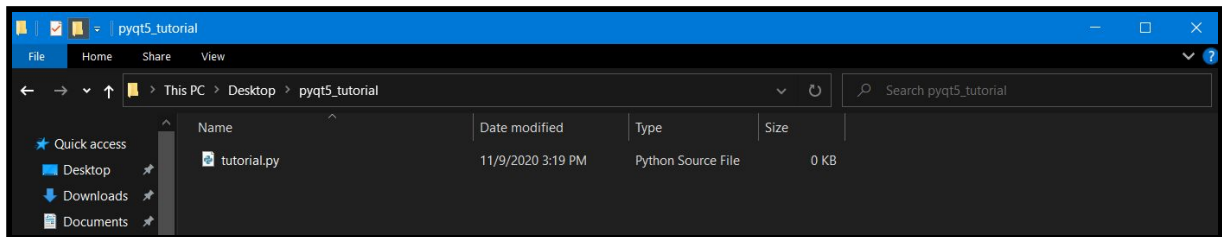
Here is an example of what a simple GUI looks like. To create this GUI for our Black Scholes PDE Calculator, we made use of PyQt5.

## 17.1 PyQt5

PyQt5 is a library that lets us use the Qt GUI framework from Python (Herrmann, n.d.). Through PyQt5, we can create simple GUIs with just a few lines of Python code. Before we begin exploring how PyQt5 can help us create GUIs, we need to first have a code editor to write Python scripts. We will eventually run our Python script from our Python interpreter (in this case, we will be using Anaconda Prompt).

## 17.2 Creating PyQt5 scripts

In this section, we will learn how to create PyQt5 scripts. To do so, we need a functional code editor to write our scripts. You may use any code editor for the job, but we recommend using [Visual Studio Code](#) because of its easy to use interface. Once we have installed our code editor, we next create a new Python file called 'tutorial.py' and save it at a location of our choice. In this example, we will be saving our Python file onto our desktop in a folder called pyqt5\_tutorial. Here's how the path to our local Python file will look like: C:\Users\Sean\Desktop\pyqt5\_tutorial.



(Figure 17.2.1)

Open `tutorial.py` in your code editor, and type the following code to create a blank window.

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget
app = QApplication(sys.argv)
w = QWidget()
w.show()
sys.exit(app.exec_())
```

Once you have the above code in `tutorial.py`, open up Anaconda Prompt. In Anaconda Prompt, navigate to the folder where `tutorial.py` is located at. To do so, simply type the following command in your Anaconda Prompt:

```
> cd <insert path of your tutorial.py file>
```

In our case, the path of our `tutorial.py` file is: `C:\Users\Sean\Desktop\pyqt5_tutorial`. Hence we type the following:

```
> cd C:\Users\Sean\Desktop\pyqt5_tutorial
```

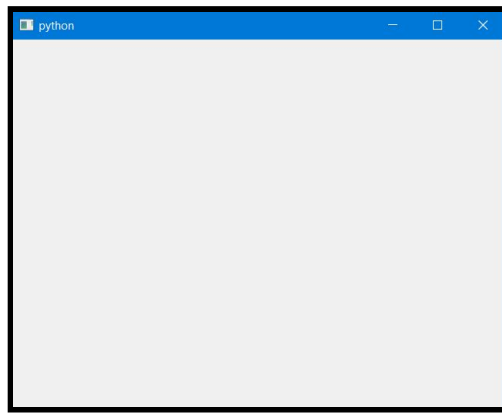
If you have successfully changed your directory in Anaconda Prompt, you should see something similar to this:

```
(base) C:\Users\Sean\Desktop\pyqt5_tutorial>_
```

(Figure 17.2.2)

We can then get Anaconda Prompt to run the Python script that we previously created. To do so, type the following command:

```
> python pyqt5_tutorial.py
```



(Figure 17.2.3)

If you have followed the above steps correctly, you should see the following empty window pop up. Congratulations! You have created your first GUI.

### 17.3 Understanding the PyQt5 code

```
1  import sys
2  from PyQt5.QtWidgets import QApplication, QWidget
3  app = QApplication(sys.argv)
4  w = QWidget()
5  w.show()
6  sys.exit(app.exec_())
```

(Figure 17.3.1)

To get a basic understanding of what happens behind the scenes, let us explore what the following lines of code do.

Lines 1-2: These 2 lines import all the necessary modules required for us to create a Qt GUI.

Line 3: Every PyQt5 application must have a QApplication object!

Line 4: The QWidget is the base class of all user interface objects in PyQt5. In Line4, we create a QWidget widget with its default “constructor”, which has no parent. A widget without a parent is called a window.

Line 5: Line 5 displays our QWidget object with the function show(). Without it, our empty window would not have popped up.

Line 6: With app.exec\_(), the program will enter the main loop where it receives events (or user commands) from the window system and dispatches them to the application widgets. The main loop ends when we execute the exit() method by clicking on the “X” button of our empty window.

#### 17.4 if `__name__ == '__main__':`

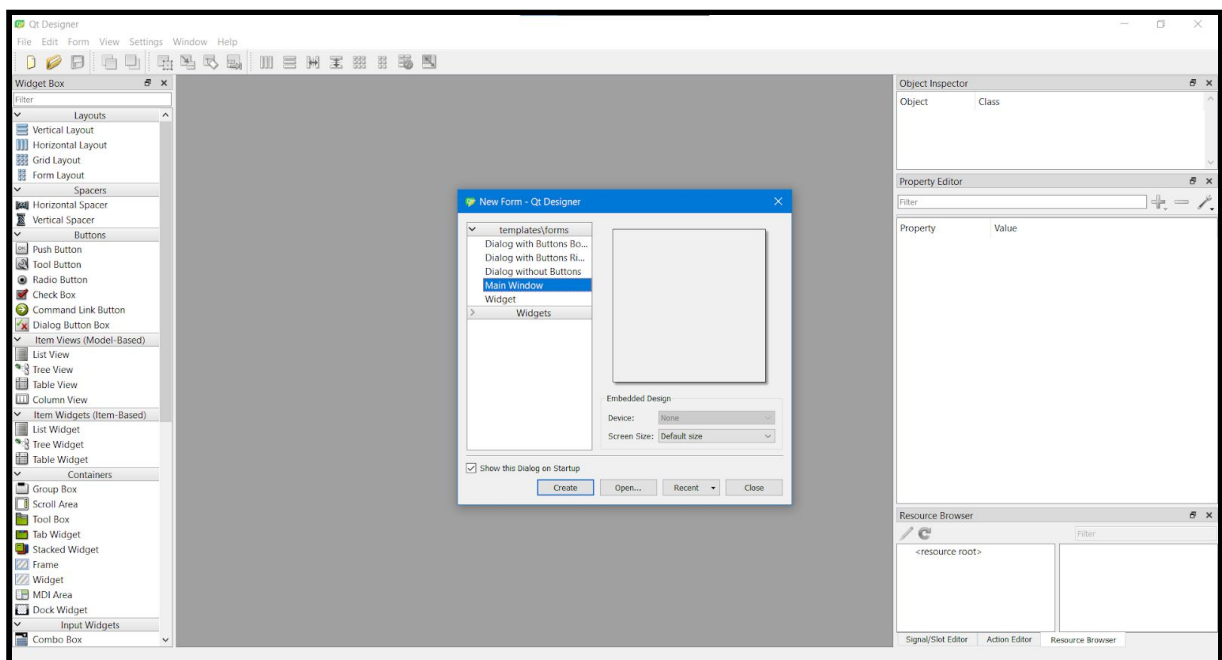
In later parts of our code, we will be using the following line of code:

```
if __name__ == '__main__':
```

This line allows us to run the Python file as a script as well as an importable module.

#### 17.5 Qt Designer

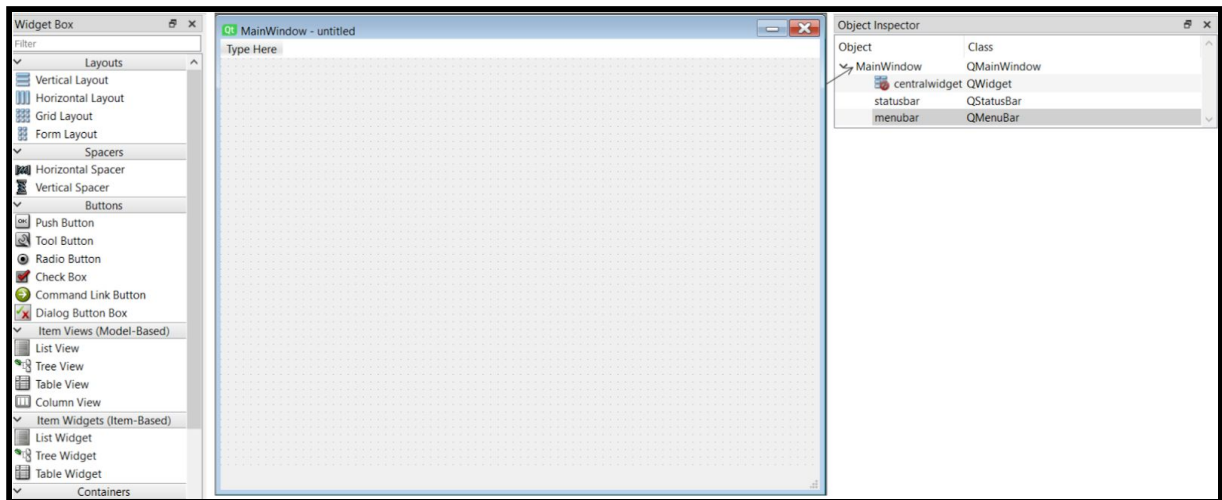
It can be rather unintuitive to type out the code for every individual PyQt5 widget in Python to create a functional GUI. Luckily, Qt Designer can help us create a GUI template intuitively. Our PyQt5 script can then read off this GUI template that we created with Qt Designer to create a functional GUI without us having to code out every single widget. Qt Designer can be downloaded [here](#).



(Figure 17.5.1)

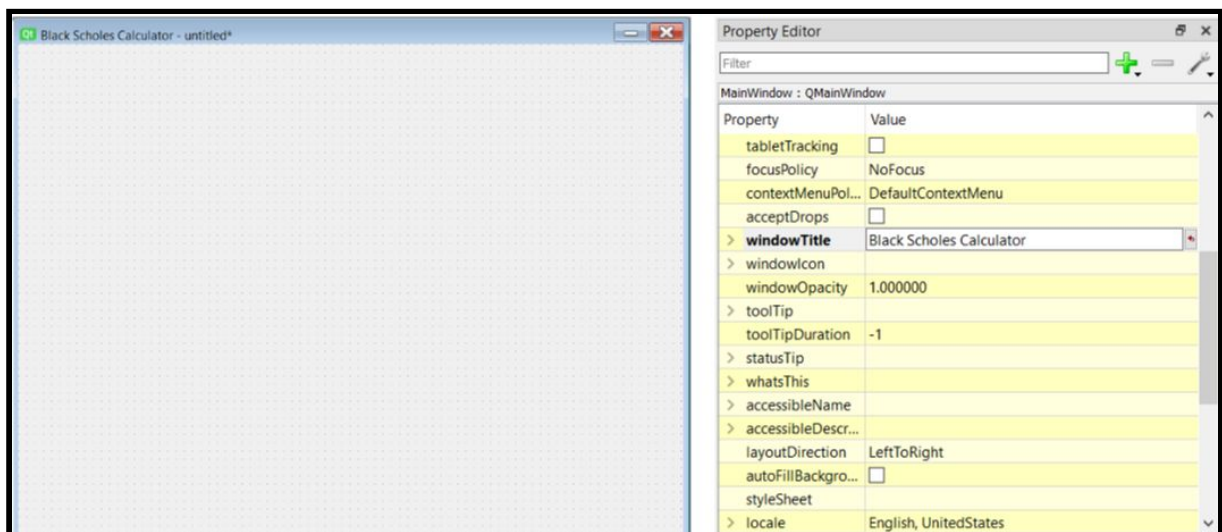
When we first start Qt Designer, we will be greeted with this page. To create our very own GUI, let us create a new Main Window.





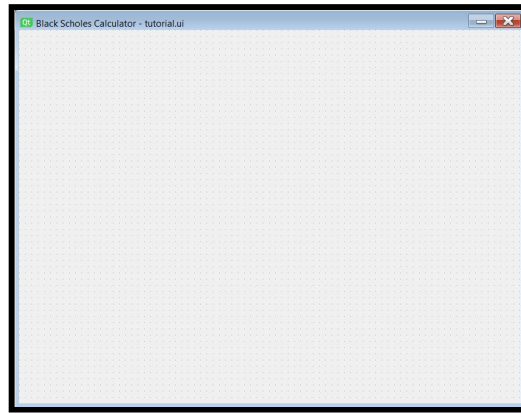
(Figure 17.5.2)

The Main Window is where we will add our widgets. On the right, we have our object inspector, which shows us the different elements and widgets we have in our GUI. We can drag and drop widgets from the widget box on the left onto our Main Window to design our GUI.



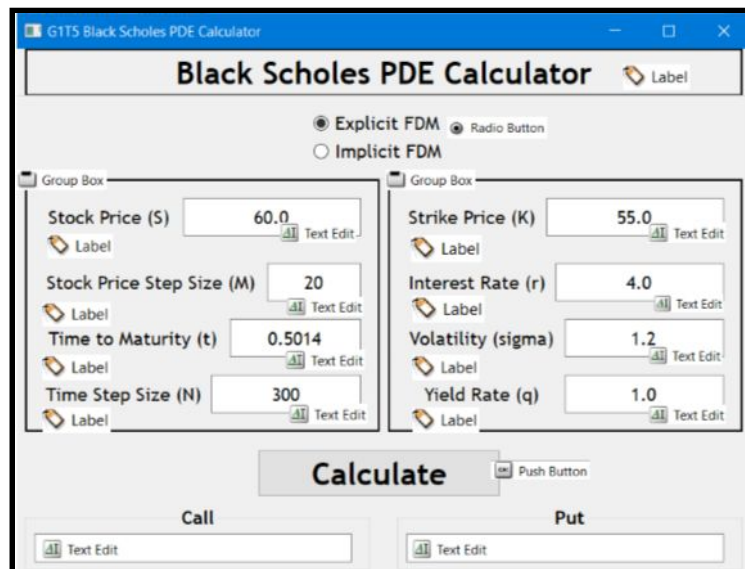
(Figure 17.5.3)

Let us first change the title of our window. We can do so in the Property Editor panel on the right on Qt Designer. The title can be changed under the *windowTitle* property.



(Figure 17.5.4)

Notice the file name of our GUI is *untitled*. To rename our GUI, simply save the UI file (by pressing Ctrl+S or going to File > Save to save it).



(Figure 17.5.5)

You may use the following widgets to recreate our Black Scholes Calculator, or design a new one for yourself. You may reposition the widgets by dragging them around the Main Window. You can adjust attributes such as border thickness, font and font size in the Property Editor panel.

## 17.6 Connecting our PyQt5 script to the Qt ui file

```
import sys
from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5 import uic

qtCreatorFile = "tutorial.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)

class Main(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
```

(Figure 17.6.1)

We can get our PyQt5 script to run the Qt ui file we created with the following code. The import statements are required to import necessary modules that are needed to run our PyQt application and ui file. You can specify the ui file you want to use by replacing “*tutorial.ui*” with your ui file name.

```
1  import sys
2  from PyQt5.QtWidgets import QMainWindow, QApplication
3  from PyQt5 import uic
4
5  qtCreatorFile = "tutorial.ui"
6  Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)
7
8  class Main(QMainWindow, Ui_MainWindow):
9      def __init__(self):
10         super().__init__()
11         self.setupUi(self)
12
13  if __name__ == '__main__':
14     app = QApplication(sys.argv)
15     main = Main()
16     main.show()
17     sys.exit(app.exec_())
```

(Figure 17.6.2)

Here is the standard code that will allow the PyQt5 file to run as a script. With that, you now have all the necessary tools required to recreate our Black Scholes PDE Calculator! Do check out our full code in the Appendix if you require a reference while recreating the calculator. Good luck and have fun!

## 18. References

Downey, L. (2020, August 28). The Essential Options Trading Guide. Retrieved November 09, 2020, from <https://www.investopedia.com/options-basics-tutorial-4583012>

Herrmann, M. (n.d.). PyQt5 tutorial 2020: Create a GUI with Python and Qt. Retrieved November 09, 2020, from <https://build-system.fman.io/pyqt5-tutorial>

Hope, C. (2019, November 16). What is a GUI (Graphical User Interface)? Retrieved November 09, 2020, from <https://www.computerhope.com/jargon/g/gui.htm>

Kenton, W. (2020, September 16). How the Black Scholes Price Model Works. Retrieved November 09, 2020, from <https://www.investopedia.com/terms/b/blackscholes.asp>

Phung, A. (2020, November 04). How Do Speculators Profit From Options? Retrieved November 09, 2020, from <https://www.investopedia.com/ask/answers/06/speculateoptions.asp>

Yates, T. (2020, August 29). What Are the Best Hedging Strategies? Retrieved November 09, 2020, from <https://www.investopedia.com/articles/optioninvestor/07/affordable-hedging.asp>

## 19. Appendix

### 19.1 Backend Code for Black Scholes PDE Calculator

```
import matrixOperations
from math import exp, floor

class blackscholes():
    def __init__(self, S, t, K, r, sigma, q):
        self.S = float(S)
        self.t = float(t)
        self.K = float(K)
        self.r = float(r)/100
        self.sigma = float(sigma)
        self.q = float(q)

    #Explicit FDM
    def explicitfdm(self, M, N):
        #1. Compute delta_t = t/N; delta_s = s_max/M, where s_max = 2*K
        delta_t = self.t/N
        s_max = 2 * self.K
        delta_s = s_max/M

        #2. Compute (Call option) fN,j=max(j*delta_s-K,0), (Put option)
        fN,j=max(K-j*delta_s), for j=0,1,...,M
        call = [[0 for j in range(M+1)] for x in range(N+1)]
        put = [[0 for j in range(M+1)] for x in range(N+1)]
        for j in range(M + 1):
            call[N][j] = max(j * delta_s - self.K, 0)
            put[N][j] = max(self.K - j * delta_s, 0)

        #3. For i=N-1, N-2,...,1,0, repeat 3.1 and 3.2
        for i in range(N - 1, -1, -1):
            #3.1. Compute vector Fi=A*Fi+1
            for j in range(1, M):
                a = 0.5 * delta_t * (self.sigma * self.sigma * j * j -
                (self.r - self.q) * j)
                b = 1 - delta_t * (self.sigma * self.sigma * j * j +
                self.r)
                c = 0.5 * delta_t * (self.sigma * self.sigma * j * j +
                (self.r - self.q) * j)
                call[i][j] = (a * call[i+1][j-1] + b * call[i+1][j] + c
                * call[i+1][j+1])
                put[i][j] = (a * put[i+1][j-1] + b * put[i+1][j] + c *
                put[i+1][j+1])
```

```

        #3.2. Compute vector Fi
        call[i][0] = 0
        call[i][M] = s_max - self.K * exp(-self.r * (N-i) *
delta_t)

        put[i][0] = self.K * exp(-self.r * (N-i) * delta_t)
        put[i][M] = 0

        #4. Find k, such that k*(delta_s) <= S <= (k+1)*(delta_s), i.e.
k=[S/delta_s]
        k = int(floor(self.S / delta_s))

        #5. Option price: V = f0,k + ((f0,k+1 - f0,k) / delta_s) * (S -
k*delta_s)
        Vcall = round((call[0][k] + (call[0][k+1] - call[0][k]) /
delta_s * (self.S - k * delta_s)),4)
        Vput = round((put[0][k] + put[0][k+1] - put[0][k]) / delta_s *
(self.S - k * delta_s),4)
        return [Vcall, Vput]

#Implicit FDM
def implicitfdm(self, M, N):
    delta_t = self.t/N
    s_max = 2 * self.K
    delta_s = s_max / M

    callh = [[0 for _ in range(N + 1)] for _ in range(M + 1)]
    puth = [[0 for _ in range(N + 1)] for _ in range(M + 1)]
    call = [[0 for _ in range(N + 1)] for _ in range(M + 1)]
    put = [[0 for _ in range(N + 1)] for _ in range(M + 1)]
    for j in range(M + 1):
        call[j][N] = max(j * delta_s - self.K, 0)
        put[j][N] = max(self.K - j * delta_s, 0)

    A = [[0 for _ in range(M + 1)] for _ in range(M + 1)]
    A[0][0] = 1
    A[M][M] = 1
    for j in range(1, M):
        A[j][j - 1] = 0.5 * delta_t * ((self.r - self.q) * j -
self.sigma * self.sigma * j * j)
        A[j][j] = 1 + delta_t * (self.sigma * self.sigma * j * j +
self.r)
        A[j][j + 1] = -0.5 * delta_t * (self.sigma * self.sigma * j
* j + (self.r - self.q) * j)

```

```

        A_inv = matrixOperations.inverse(A)
        for i in range(N - 1, -1, -1):
            # 3.1 0 for j=0, same for j=1toM-1, exponential discounting
            for highest payout
            for j in range(M+1):
                callh[j][i + 1] = call[j][i + 1]
                puth[j][i + 1] = put[j][i + 1]
            callh[0][i + 1] = 0
            callh[M][i + 1] = s_max - self.K * exp(-self.r * (N - i) *
delta_t)

            puth[M][i + 1] = 0
            puth[0][i + 1] = self.K * exp(-self.r * (N - i) * delta_t)
            # 3.2 store vectors from temp cols
            callh_cv = [[callh[j][i + 1]] for j in range(M + 1)]
            puth_cv = [[puth[j][i + 1]] for j in range(M + 1)]
            call_cv = matrixOperations.multiply(A_inv, callh_cv)
            put_cv = matrixOperations.multiply(A_inv, puth_cv)
            for j in range(M + 1):
                call[j][i] = call_cv[j][0]
                put[j][i] = put_cv[j][0]

        k = int(floor(self.S / delta_s))

        Vcall = round((call[k][0] + (call[k + 1][0] - call[k][0]) /
delta_s * (self.S - k * delta_s)),4)
        Vput = round((put[k][0] + (put[k + 1][0] - put[k][0]) / delta_s
* (self.S - k * delta_s)),4)
        return [Vcall, Vput]

```

## 19.2 Backend Code for Black Scholes PDE Calculator PyQt5 Application

```
import sys
from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5 import uic
from blackscholespde import blackscholes

qtCreatorFile = "blackscholes.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)

class Main(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.explicit_fdm.setChecked(True)
        self.Calculate.clicked.connect(self.blackscholescalc)

    def blackscholescalc(self):
        S = float(self.S.text())
        t = float(self.t.text())
        K = float(self.K.text())
        r = float(self.r.text())
        sigma = float(self.sigma.text())
        q = float(self.q.text())
        M = int(self.M.text())
        N = int(self.N.text())

        method = blackscholes(S, t, K, r, sigma, q)

        if self.explicit_fdm.isChecked():
            result = method.explicitfdm(M, N)
        else:
            result = method.implicitfdm(M, N)

        self.call.setText(str(result[0]))
        self.put.setText(str(result[1]))

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = Main()
    main.show()
    sys.exit(app.exec_())
```