<u>**QF205**</u>

<u>GROUP PROJECT REPORT</u>

<u>**SUBJECT NAME:**</u>

QF205 – Computing Technology for Finance

Python Programming and Its Applications in Option Pricing

<u>**ACADEMIC YEAR:**</u>

2023/2024, Semester 1

<u>**SECTION:**</u>

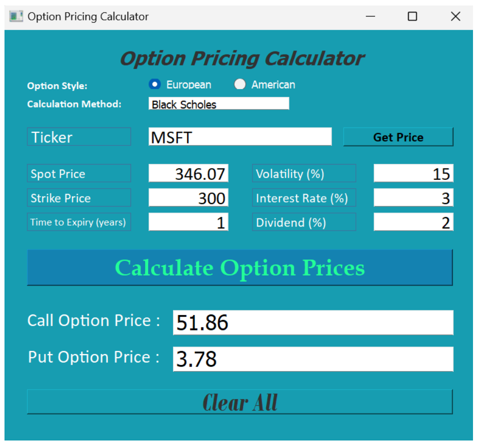
G2 Team 2

MATRICULATED NAME	STUDENT ID
(Jenna) Tiu Chien Yi	01439362
Kaydon Lim Jun Long	01397457
Seow Wei Xuan	01393321
Chew Wei Qian	01412415
Htoo Myat Naing	01395605

Contents

1. Abstract

In this report, we will see how to create an option pricing calculator using the Python programming language. By the end of this report, you will acquire the knowledge and skills necessary to construct an option pricing calculator, applying the renowned Black-Scholes and Binomial Tree methods, to assess both European and American options. The resulting calculator will feature a user-friendly graphical interface (GUI), as depicted in the image below.



Option Pricing Calculator

Option Pricing Calculator

Option Style: [✓ European] American

Calculation Method: Black Scholes

Ticker: MSFT

Spot Price: 346.07
Volatility (%): 15
Strike Price: 300
Interest Rate (%): 3
Time to Expiry (years): 1
Dividend (%): 2

Call Option Price : 51.86
Put Option Price : 3.78

Figure 1: Options Pricing Calculator (Black-Scholes)

Before delving into the Python programming language and the various knowledge points necessary to construct this calculator, let's begin by first gaining an understanding of what options are, its purpose, and how their calculations are performed.

2. Introduction to Options

Options are financial instruments that give you the right, but not the obligation, to buy or sell an underlying asset at a specified price, known as the "strike price," before or on a predetermined date, known as the "expiration date."

Let's break down this concept into simpler terms:

- **The Underlying Asset**: Imagine you want to buy a rare collectible toy or sell your bicycle. The collectible toy or the bicycle is the "underlying asset." In the financial world, this asset could be a stock (like shares of a company), a commodity (like gold or oil), or even an index (like the S&P 500).
- **The Right, but Not the Obligation**: Think of options as a special type of agreement. When you buy an option, you're essentially paying for a choice. It's like putting a down payment on a toy but not having to buy it if you change your mind later. Similarly, with options, you have the "right" to buy or sell the underlying asset, but you're not "obligated" to do so.
- **Strike Price**: The strike price is like the price tag on the toy or the price you agree upon to sell your bicycle. It's the price at which you can buy or sell the underlying asset if you decide to exercise your option. For example, if you have a call option (explained later) with a strike price of $50 for a stock, you can buy that stock for $50 per share if you choose to.
- **Expiration Date**: Just like toys might lose value or your bicycle might become outdated, options have a deadline. The expiration date is when your option contract expires. You must make your decision to buy or sell the underlying asset before this date. If you don't, the option becomes worthless.

Now, there are two main types of options:

- **Call Options**: These give you the right to buy the underlying asset at the strike price. Think of it like having the option to "call" the asset to yourself at the agreed-upon price. If the asset's market price goes above the strike price, you can buy it at the lower strike price and make a profit.
- **Put Options**: These give you the right to sell the underlying asset at the strike price. Think of it as having the option to "put" the asset to someone else at the agreed-upon price. If the asset's market price goes below the strike price, you can sell it at the higher strike price and potentially avoid losses.

Here's a simple example to tie it all together:

Suppose you have a call option for a toy with a strike price of $20, and the option expires in a month. If, before the expiration date, the toy becomes really popular, and its price goes up to $30, you can use your call option to buy it for $20 and sell it for $30, making a $10 profit.

2.1 Types of Options

For the purpose of this project, we will focus on European and American options. We will provide a detailed explanation of these options when we discuss their pricing. European and American options are types of financial contracts that give you the right, but not the obligation, to buy or sell an underlying asset (like stocks) at a specified price on or before a certain date.

The key difference is in when you can exercise your right:

1. **European Options**: You can only exercise them at the expiration date.
2. **American Options**: You can exercise them at any time before or on the expiration date.

In simple terms, European options have a fixed exercise date, while American options offer flexibility to exercise at any time. We will learn how to use the Black-Scholes method to calculate European options, and the Binomial Tree method to calculate both European and American options.

2.2 Uses of Options

Options serve several important purposes in the world of finance. Let's explore what options are used for in simple terms, without assuming any prior financial knowledge:

- **Managing Risk**: Options act as financial safety nets, similar to buying insurance for an outdoor event against rain. In finance, they protect investments from unexpected price movements. For instance, a "put option" allows an investor to sell a stock at a preset price, thereby limiting losses if the stock's value drops.
- **Speculating on Price Movements**: Options enable investors to bet on the future price of an asset without owning it. This is akin to betting on a sports team's victory. In financial terms, buying a "call option" profits from an asset's price increase, while a "put option" profits from its decrease.

In summary, options are versatile financial tools that offer solutions for managing risk, speculating on price movements, etc. They are like tools in a financial toolbox that investors and traders can use to achieve their specific goals and protect their financial interests. However, as this is not a paper on options strategy, we will not elaborate further on how to effectively use options to generate profit making strategies. We will show you how to price options instead and make a calculator for it.

2.3 How do you price an Option?

There are a few ways to price an Option. For this project, we will be teaching you how to price an option using the binomial model and the Black-Scholes method. The Black-Scholes model and the binomial tree method are two techniques to determine the value of financial options. Black-Scholes is a formula that works well for European options with continuous price movements, providing quick estimates. In contrast, the binomial tree method is more versatile, suitable for both European and American options with discrete price steps, offering precision in various market conditions, especially when early exercise is a consideration. The choice between these methods depends on the type of option and market dynamics. As using the binomial model and the Black-Scholes method can seem complex, let us break it down step by step to make it simple to understand.

2.3.1 Black-Scholes Method

The Black-Scholes method is a formula used to estimate the prices of options. This includes both call and put options. This model is particularly useful for European options, which can only be exercised at expiration, and it has played a crucial role in pricing and trading options in financial markets. Hence, we will use this method to calculate the price of European Call and Put options only. To calculate the option prices, we simply utilize the Black-Scholes formula shown below and plug in the respective numbers.

$$C = SN(d_1) - Xe^{-rT}N(d_2)$$

$$P = Xe^{-rT}N(-d_2) - SN(-d_1)$$

$$d_1 = \frac{\ln\left(\frac{S}{X}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

$$d_2 = \frac{\ln\left(\frac{S}{X}\right) + \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}$$

C = Call Option Price

P = Put Option Price

S = Underlying Stock Price

X = Strike Price of Option

r = Risk-Free Interest Rate

T = Time to expiration in years

σ = Volatility of the relative price change of the underlying stock price

N(x) = Cumulative distribution function of the standard normal distribution as shown below

$$\mathrm{N(x)} = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}}\, dt$$

S, r, T, are known values that are publicly available. X and σ on the other hand, depends on the trader's inputs and judgements. As this is a programming report and not an options strategy report, we will use arbitrary values for the calculations.

Sample Calculations:

Sample Values	
S	\$40
X	\$45
r	3% or 0.03
T	4 months
σ	40% or 0.4

Using the formulas, let's calculate d_1 and d_2:

$$d_1 = \frac{\ln\left(\frac{40}{45}\right) + \left(0.03 + \frac{0.4^2}{2}\right)\left(\frac{4}{12}\right)}{0.4\sqrt{\frac{4}{12}}} = -0.3512442$$

$$d_2 = \frac{\ln\left(\frac{40}{45}\right) + \left(0.03 + \frac{0.4^2}{2}\right)\left(\frac{4}{12}\right)}{0.4\sqrt{\frac{4}{12}}} = d_1 - 0.4\sqrt{\frac{4}{12}} = -0.5821843$$

Now, let's calculate the price of a call option:

$$N(d_1) = \int_{-\infty}^{d_1} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}\, dx = 0.3627026$$

$$N(d_2) = \int_{-\infty}^{d_2} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}\, dx = 0.2802213$$

$$C = (40)(0.3627026) - (45)e^{-(0.03)\left(\frac{4}{12}\right)}(0.2802213) = 2.023617$$

Now, let's calculate the price of a put option:

$$N(-d_1) = \int_{-\infty}^{d_1} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}\, dx = 0.6372974$$

$$N(-d_2) = \int_{-\infty}^{d_2} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}\, dx = 0.7197787$$

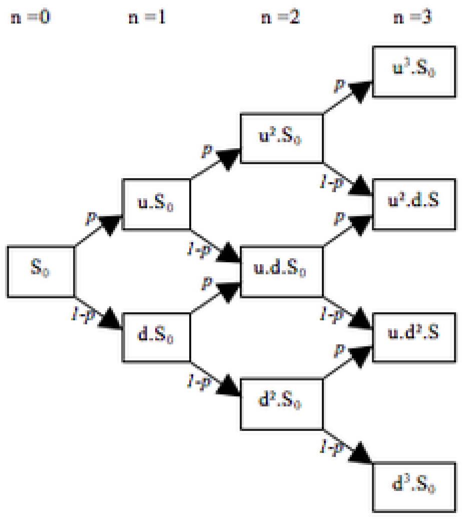
$$P = (45)e^{-(0.03)\left(\frac{4}{12}\right)}(0.7197787) - (40)(0.6372974) = 6.57586$$

We will obtain the values Call: \$2.023617 and Put: \$6.57586 for European options. Let us compare these results with the Binomial Method which will be exploring next.

2.3.2 Binomial Method

The binomial method, often referred to as the binomial options pricing model, is a computational approach used to estimate the value of financial options, such as call and put options. This method is named after its use of a tree-like structure (a binomial tree) to model the possible price movements of the underlying asset over time.

The binomial method breaks down the option's life into discrete time intervals and allows for the calculation of the option's value at different points in time as shown in Figure 2 below. It considers factors like the current stock price, the option's strike price, time to expiration, volatility, and the risk-free interest rate. Unlike the Black-Scholes model, the binomial method is more versatile as it can be applied to both European and American options, which can be exercised at any time before or at expiration.



$$p = \frac{e^{rt/n} - d}{u - d}$$

$$u = e^{\sigma\sqrt{t/n}}$$

$$d = e^{-\sigma\sqrt{t/n}}$$

Figure 2: Binomial Tree

The binomial tree above is created using the following steps:

1. Visualize a tree with branches and nodes. At each node, we'll calculate two values: the option's value if the price goes up (call it "Cu" for Call options or "Pu" for Put options) and the option's value if the price goes down (call it "Cd" for Call options or "Pd" for Put options).
2. Start with the current price of the underlying asset, usually at the tree's root.
3. Divide time into equal periods (e.g., days or months) until the option's expiration date. Each step down the tree represents one time period (T).
4. Calculate the probability of an upward or downward movement at each node using the formula p. You'll need the current price, volatility (how much the price tends to fluctuate), and the risk-free rate.

Now, let's calculate the option values at each node:

- For **Call options** ("Cu" and "Cd"): At the final nodes (closest to the expiration date), calculate the option value as the maximum of 0 and the difference between the asset's price and the strike price. Then, work your way up the tree, calculating option values at each node by considering the probabilities of moving up or down.
- For **Put options** ("Pu" and "Pd"): At the final nodes, calculate the option value as the maximum of 0 and the strike price minus the asset's price. Then, work your way up the tree, again considering probabilities.

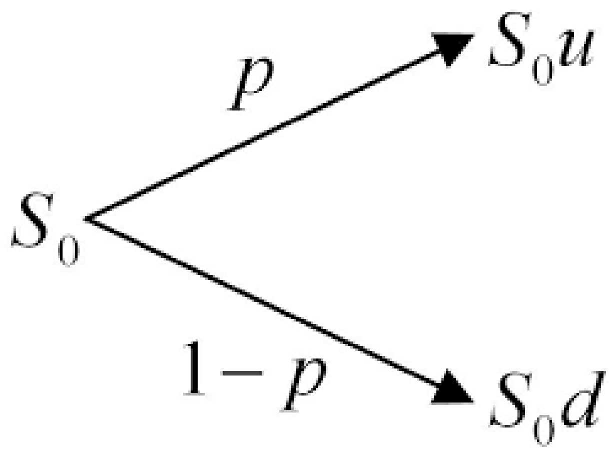


Figure 3: Binomial Tree with One Time Step

Sample Values	
S	$40
X	$45
r	3% or 0.03
T	4 months
σ	40% or 0.4

We will be using the samples values that were used in calculating options prices using the Black-Scholes Method.

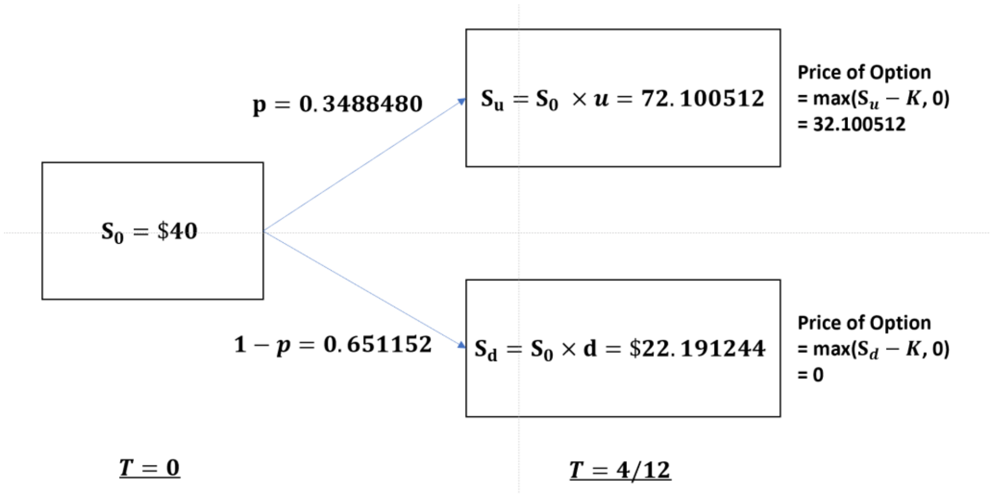
Let’s first calculate the values of u, d and p:

$$u = e^{0.4\sqrt{\frac{4}{12}}} = 1.8025128$$

$$d = e^{0.4\sqrt{\frac{4}{12}}} = 0.5547811$$

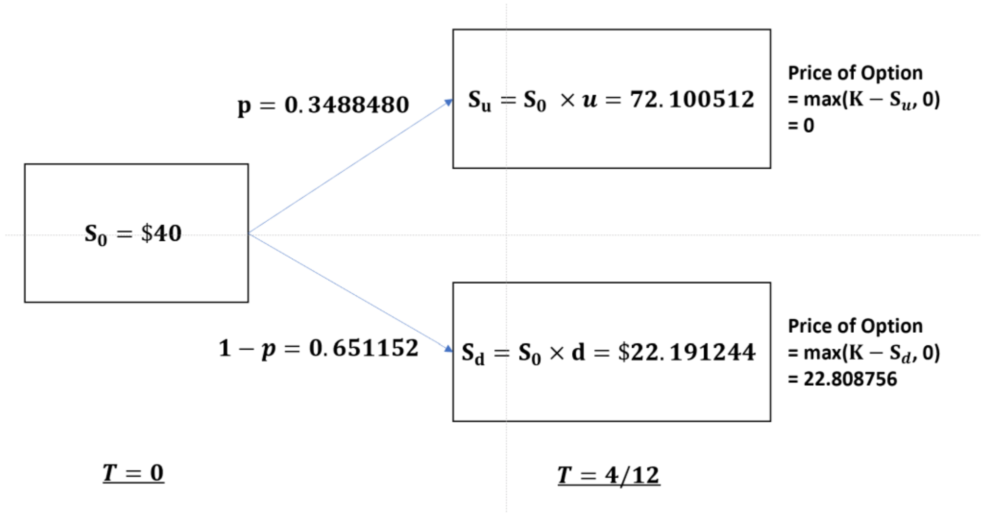
$$p = \frac{e^{(-0.03)\left(\frac{4}{12}\right)} - d}{u - d} = 0.3488480$$

Next, let's calculate the price of a Call Option for one timestamp using the Binomial method:



$$\text{Expected Price of Call Option} = \max(S_u - K, 0) \times p_u + \max(S_d - K, 0) \times p_d = 11.198199$$

Next, let's calculate the price of a Put Option for one time stamp using the Binomial method:



$$\text{Expected Price of Call Option} = \max(K - S_u, 0) \times p_u + \max(K - S_d, 0) \times p_d = 14.851967$$

From the above calculations, we can see that for one time stamp (T=4/12) we will obtain the values Call: \$11.198199 and Put: \$14.851967.

Now that you have option values at each node, you need to discount them back to the present value. Use the risk-free rate and the time period to calculate the present value of each option value at every node. At the initial node (the root of the tree), you'll find the option's price. This is the present value of the option's expected future payoffs. You can adjust the inputs, like the volatility or the time period, to see how they affect the option's price. This helps you understand how sensitive the option's value is to changes in these factors.

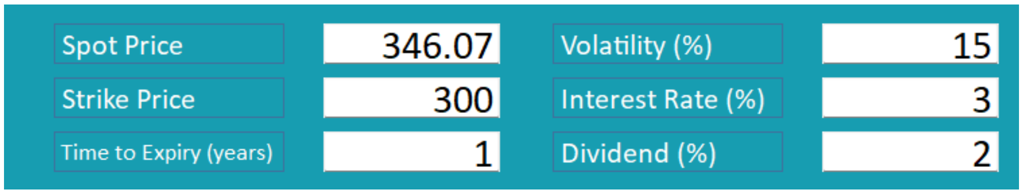
By repeatedly applying the binomial process, starting from the present and working forward to expiration, the method can provide a more accurate estimate of the option's value under various market conditions, especially when early exercise is a possibility. This makes it a valuable tool for pricing and managing options in financial markets.

Now that we understand Options and how to calculate it, we can observe that the process of calculating it is very tedious, especially with the binomial tree when there are many nodes involved. Therefore, building a calculator can help us perform the calculations more quickly. Before we delve into building the calculator, let's examine how the application functions and what the user will encounter when launching it.

2.4 How the Options Calculator is Used

The calculator's interface is displayed on what we call a Graphic User Interface (GUI). A GUI is a visual interface that allows users to interact with computers and software through graphical elements like icons and menus. This allows the utilization of the calculator in a more user-friendly way, without having to view complicated code that is performing the calculations for the option pricing.

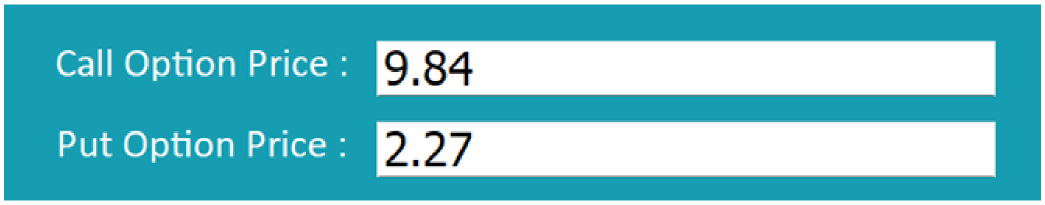
When a user uses the calculator, they will first need to key in the various inputs required to perform the option calculation, as mentioned above. The inputs required is shown in Figure 4 below.



Spot Price	346.07	Volatility (%)	15
Strike Price	300	Interest Rate (%)	3
Time to Expiry (years)	1	Dividend (%)	2

Figure 4: Calculator Inputs

Thereafter, the calculator will perform the calculations and generate the option prices which will be displayed in Figure 5 below.



Call Option Price : 9.84

Put Option Price : 2.27

Figure 5: Call and Put Option Price Ouptut

Now that we have an idea of how the application can be used, let us understand what Python programming language is and the knowledge points we need to use to construct the different parts of the calculator.

3. What is Python?

Python is a versatile and widely-used high-level programming language known for its simplicity and readability. It is a popular choice for various applications, from web development and data analysis to artificial intelligence. Python's extensive library and active community support make it a powerful tool for both beginners and experts in academia and industry.

3.1 Python Concepts Needed to Build your Own Options Pricing Calculator

As mentioned earlier, a GUI is a graphic user interface. This is where users will key in their inputs and obtain calculations. There is a straightforward method to create the GUI by using external packages, which are pre-built libraries of code created by other developers that you can readily integrate into your projects, saving you time and effort that would otherwise be spent writing code from scratch. These GUI packages enable you to easily build a graphical user interface without having to create it entirely on your own, as another developer has already done the groundwork, and you can customize it to suit your specific needs.

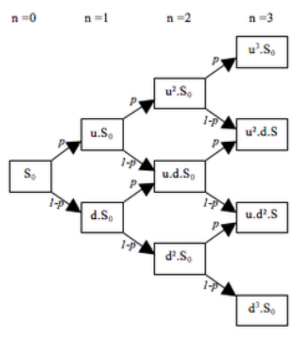
To begin, users of this GUI will be required to provide certain inputs for option calculations. We will guide you on how to store these inputs so that we can use them for calculations later. The inputs will be stored in what we call a *dictionary*. When the user keys in their inputs, the computer will read them as texts rather than numbers, therefore we need to convert these texts (*strings*) to numbers (*floats*) to perform calculations. Once we have our inputs converted to numbers, we can start calculating the option prices.

3.1.1 Black-Scholes Method

Firstly, let us go over the Black-Scholes method of calculating option prices. The Black-Scholes method is more intuitive to compute as it utilizes mathematical formulas rather than an algorithm to calculate. This method involves a deep dive into various *arithmetic operations*, alongside functions like log, sqrt, and exp. Just like any mathematical question, there are different steps to solve for the answer. To ensure that the calculations are clear and easy to reference, we will allocate the formulas to what we call a *variable*. You may think of variables as giving the formula a name. Every time we need the formula, rather than typing out the formula again, we can access it by calling its name, which in this case is a variable. Furthermore, just like a mathematical question, sometimes we require multiple steps or formulas to solve it. To make sure that the computer understands that all the workings are for a specific question, we can group the workings together into a *function*. We will show you how to store these formulas within a function so that we can solve for the option prices using the Black-Scholes method. After calculations are performed, we may get numbers with many decimal places. To make the calculator more user-friendly, we will delve into *rounding techniques*, ensuring option prices are accurate to 2 decimal places.

3.1.2 Binomial Tree Method

Next, let us look at the Binomial Tree method to calculate option prices. The Binomial Tree method is more advanced as it involves an algorithm to be able to calculate the option price. As mentioned earlier, the Binomial tree allows for price movements over discrete time periods by branching out into two possible paths at each step, upward and downward movement as shown in Figure 6.



$$p = \frac{e^{rt/n} - d}{u - d}$$

$$u = e^{\sigma\sqrt{t/n}}$$

$$d = e^{-\sigma\sqrt{t/n}}$$

Figure 6: Binomial Tree

As such, we will show you how to use *nested list with for loops*, because we must iterate through each time step (outer loop) and, at each step, compute the asset's potential values at all possible nodes or levels of that step (inner loop), considering both upward and downward movements from previous nodes. We will also show you an alternative way of writing for loops, known as *list comprehension*, which allows us to write concise code, keeping our codes short and sweet. A Binomial Tree comprises many nodes with different prices. To obtain the pricing at each note, we need to know how to reference it. As such, we will show you how to use *indexing*, a type of reference method to obtain the option price at the root of the binomial tree, or the final price we are trying to calculate. Additional things we will learn within the algorithm itself include how to use *max* and *range* functions. Their use case will be explained later. Earlier, we mentioned that the Binomial Tree method is more flexible as it allows us to calculate both European and American options. We can give the user the option to choose which type they would like to calculate, by using what we call *conditional statements* such as if-else to do so.

To sum up, programming this calculator has 2 key components: (1) Calculating the option prices (2) Building the GUI for display. As such, we will learn how to sort these two components into *Classes*, which is another way to categorize codes into sections for a cleaner structure. With that, let us deep dive into the aforementioned concepts and how it is coded in Python.

4. Introduction to Python Concepts

Before delving into the intricacies of the code, we will first explain some basic concepts for Python programming so that you will have a clearer understanding and subsequently be able to use it when creating your own code in the future. *We will go through Python basics using simple examples to give you a better understanding of how Python works. Thereafter, we will go through the code in detail to show you how the basic Python concepts you have learnt can be applied to create a working Options Calculator.*

4.1 Literals

In Python, literals are representations of constant values. It mainly comprises of two different types: 1) Numeric Literals and 2) String Literals. This is especially useful in the Black Scholes option pricing method as it allows me to represent mathematical variables such as stock price (S), strike price (X), risk-free interest rate (r), time to expiration (T) and volatility (σ) that are used in the equation. With literals, I am able to assign values to these variables and apply methods to solve for the eventual price of a European option.

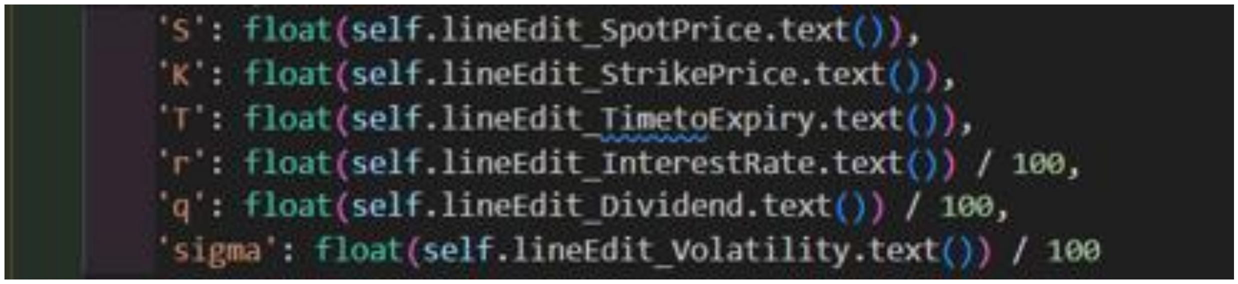
4.1.1 Numeric Literals

These literals include integers, floating point numbers and imaginary numbers. Integer literals are whole numbers without decimal places and have an unlimited length (e.g 1, 22, 333 etc.) while floating-point literals are numbers that contain one or more decimals (e.g 1.10, 2.0 etc.). Floating-point literals can be further broken down into point-floats and exponent-floats where the latter includes an 'e' to indicate the power of 10. Lastly, imaginary literals can be combined with a real number to form a complex number. Some examples include 3.14j, 3+5j where 'j' is the imaginary part. For this report, we will mainly be focusing on and utilizing floating point numbers as it is able to represent numbers with decimal places which provides more accuracy as compared to an integer.

As a whole, numeric literals do not include a sign. Hence, a phrase like '-1' is an expression that comprises of the unary operator '-' and the integer literal '1'. Knowing how literals work is is important as it teaches the state in which inputs are being stored. This allows us to make sense of the data that the calculator receives.

4.1.2 String Literals

These represent **text** and often are enclosed in matching single quotes (‘) or double quotes (“). In this report, it is especially crucial that you know how to convert string literals into floating-point literals as the computer will read inputs on the option price calculator as texts rather than numbers. To do so, you will have to the use the float () built-in function which will return a floating-point number constructed from a string. An example of this can be shown in Figure 7 below:



```
'S': float(self.lineEdit_SpotPrice.text()),
'K': float(self.lineEdit_StrikePrice.text()),
'T': float(self.lineEdit_TimetoExpiry.text()),
'r': float(self.lineEdit_InterestRate.text()) / 100,
'q': float(self.lineEdit_Dividend.text()) / 100,
'sigma': float(self.lineEdit_Volatility.text()) / 100
```

Figure 7: Converting String to Float

4.2 Arithmetic Operations

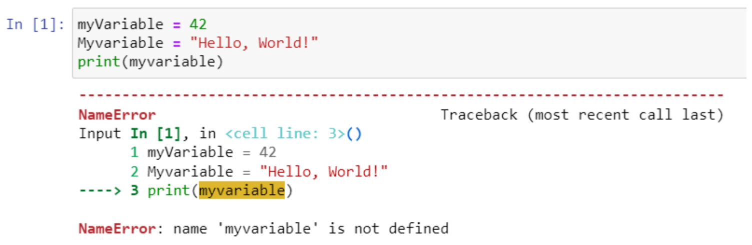
In Python, operators are used in conjunction with numeric literals to perform common mathematical operations. The numeric literals that the operators act on are also called operands. Below is a list of operators and its purpose:

Operator	Purpose	Example
+	Add two operands or unary plus	3 + 4 = 7
-	Subtract right operand from the left or unary minus	7 – 3 = 4
*	Multiply two operands	3 * 4 = 12
/	Divide left operand by the right one (always results into float)	12 / 4 = 3
%	Modulus – Remainder of the division of the left operand by the right one	13 % 4 = 1
//	Floor Division - Division that results in whole number adjusted to the left in the number line	4 // 3 = 1
**	Exponent – Left operand raised to the power of the right	4 **3 = 64

4.3 Variables

Variables refer to containers for storing different data types and it is used to reuse values that were already computed or defined earlier in the programming script.

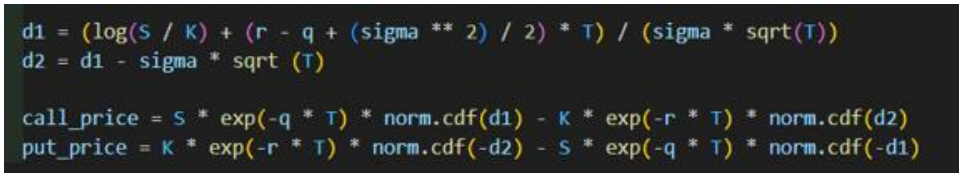
You would want to first start off with naming your variable and within the American Standard Code for Information Interchange (ASCII) range, variables can only begin with either a lower or uppercase alphabet or an underscore. The digits 0 through 9 are not allowed to be the first character but can be used subsequently within the variable name. Variable names are case-sensitive so it is pivotal to call out a variable with the right casing otherwise it will result in a NameError. This can be shown in Figure 8 below.



```
In [1]: myVariable = 42
        Myvariable = "Hello, World!"
        print(myvariable)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [1], in <cell line: 3>()
      1 myVariable = 42
      2 Myvariable = "Hello, World!"
----> 3 print(myvariable)

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

Figure 8: Case Sensitive Variable Names

To assign a value to a variable, the equal sign “=” will be employed which links the value to a variable and Python will set an appropriate type to it. Variables do not need to be declared with any particular type as they can easily be changed after they have been set. In this report, assignments were frequently used to define the variables required for option pricing as shown below:



```
d1 = (log(S / K) + (r - q + (sigma ** 2) / 2) * T) / (sigma * sqrt(T))
d2 = d1 - sigma * sqrt (T)

call_price = S * exp(-q * T) * norm.cdf(d1) - K * exp(-r * T) * norm.cdf(d2)
put_price = K * exp(-r * T) * norm.cdf(-d2) - S * exp(-q * T) * norm.cdf(-d1)
```

Figure 9: Assigning Value to Variable

4.4 Functions

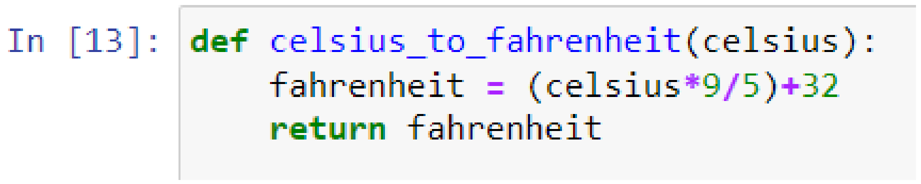
Similar to variables, functions define a block of code or set of instructions that you can reuse whenever you want to perform a specific task, helping to improve code readability and reduce repetitiveness. It can then return data as a result but there are four criteria that need to be met first:

- Firstly, a function needs to begin with the keyword *def* followed by the *function name* and *parentheses.*
- Secondly, any input parameters or arguments should be placed *within the parentheses*.
- Thirdly, the set of instructions within every function begins with a *colon (:)* and is *indented*.
- Lastly, the statement *return [expression]* exits a function

Let us use a simple example to illustrate the use of a function. If you are given a list of temperatures in degree Celsius as part of your science experiment but need to convert all of them to degrees Fahrenheit, you could employ the use of functions to help simplify the process.

Step #1: Define the Function

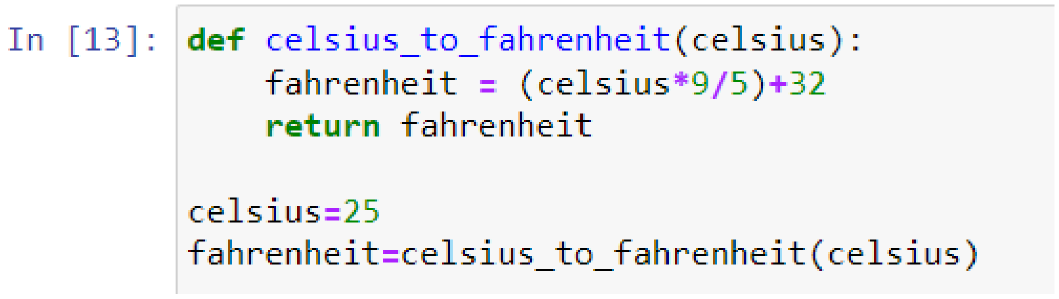
Knowing that the formula for converting degree Celsius to degree Fahrenheit is $°\mathrm{F} = \left(\frac{9}{5}\right)°\mathrm{C} + 32$, we can easily create a python function that takes a temperature in Celsius and returns the equivalent temperature in Fahrenheit, shown in Figure 10 below:



```
In [13]: def celsius_to_fahrenheit(celsius):
             fahrenheit = (celsius*9/5)+32
             return fahrenheit
```

Figure 10: Function to Convert Celsius to Farenheit

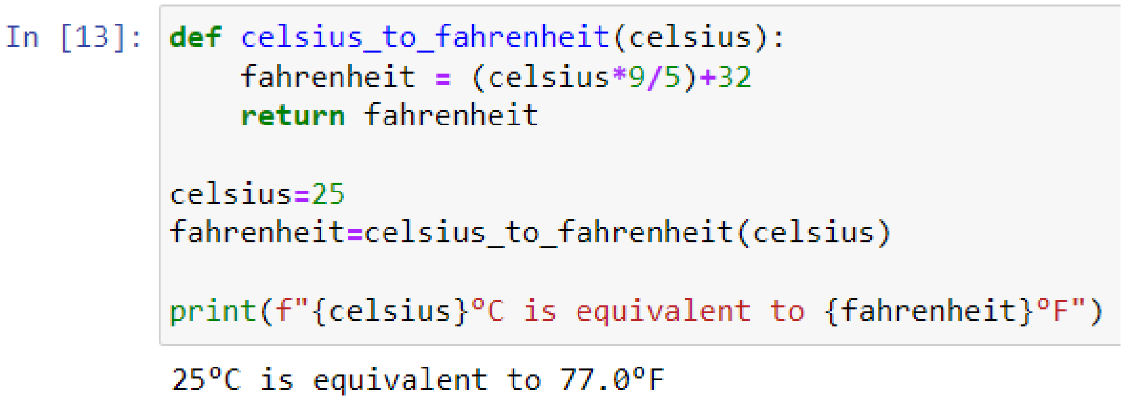
Step #2: Use the Function
In this case, the value of 25 is assigned to the variable 'celsius' and the 'celsius_to_fahrenheit' function has been called to calculate the equivalent temperature in Fahrenheit as shown in Figure 11 below.



```
In [13]: def celsius_to_fahrenheit(celsius):
             fahrenheit = (celsius*9/5)+32
             return fahrenheit

         celsius=25
         fahrenheit=celsius_to_fahrenheit(celsius)
```

Figure 11: Using the Function

Step #3: Get the result
The function calculates the equivalent temperature in Fahrenheit and returns it in the format determined by you within the parentheses of the print() function.

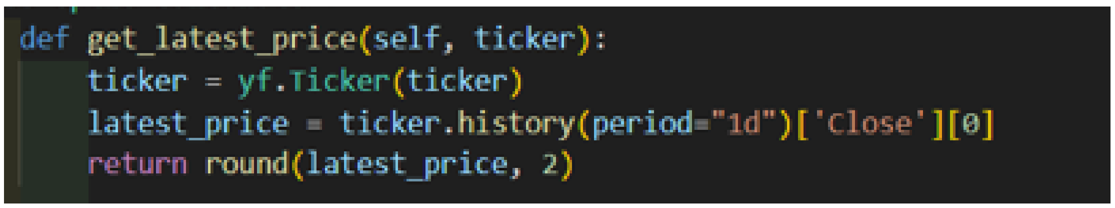


```
In [13]: def celsius_to_fahrenheit(celsius):
             fahrenheit = (celsius*9/5)+32
             return fahrenheit

         celsius=25
         fahrenheit=celsius_to_fahrenheit(celsius)

         print(f"{celsius}°C is equivalent to {fahrenheit}°F")
```

```
25°C is equivalent to 77.0°F
```

Figure 12: Calculates Temperature and Print

In this report, we often utilized functions such as the one below to retrieve the latest closing price of a specific stock using its ticker symbol. The price is then further rounded to two decimal places before being returned.



```
def get_latest_price(self, ticker):
    ticker = yf.Ticker(ticker)
    latest_price = ticker.history(period="1d")['Close'][0]
    return round(latest_price, 2)
```

Figure 13: get_latest_price( ) Function

Additionally, there are some built-in functions that Python has made to help simplify and shorten codes. Below is a list of some common built-in functions, some of which we have also incorporated in our code:

Built-in Functions	Description	Examples

range()	Returns an immutable sequence of numbers, starting from 0 and increments by 1 (by default) unless otherwise stated	`In [20]: list(range(1,7))`
		`Out[20]: [1, 2, 3, 4, 5, 6]`

max()	Returns the highest value out of a number of values	`In [18]: max(1,2,4,5)`
		`Out[18]: 5`

round()	Returns number rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted, it returns the nearest integer	`In [18]: round(875.2453,2)`
		`Out[18]: 875.25`

type()	Returns the type of a variable	`In [21]: x=42`
		`type(x)`
		`Out[21]: int`

sum()	Calculates the sum of a list	`In [22]: numbers=[1,2,3,4,5]`
		`sum(numbers)`
		`Out[22]: 15`

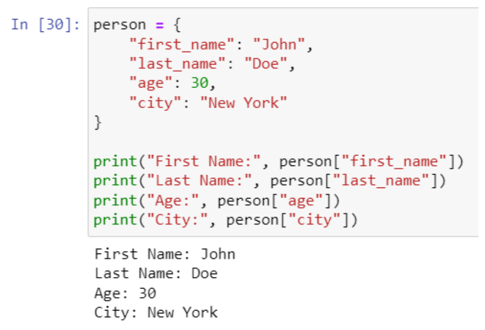
len()	Returns the length of an object	`In [23]: my_string='Hello,world'`
		`len(my_string)`
		`Out[23]: 11`

4.5 Data Structures

In Python, there are various data structures that are used to organize and store data, some of which also come as built-in functions. In this section, we will mainly be focusing on 4 data structures: Dictionary, List, Set and Tuple.

4.5.1 Dictionary

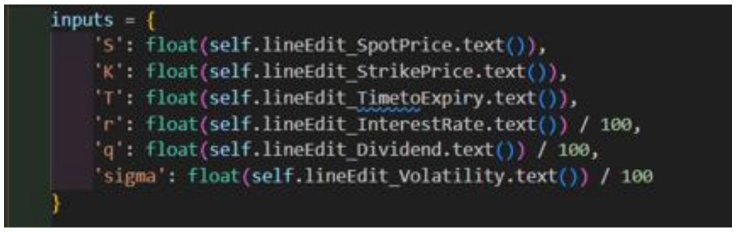
Dictionaries are used to store data values in key: value pairs, with the requirement that each key is unique. The set of key: value pairs also have to come within curly brackets and are separated using commas. The main operations of a dictionary are storing a value with some key and extracting the value given the key. For example, if you would like to store information about a person, you can create a dictionary to organize and access information about an individual, as shown in Figure 14 below.



```
In [30]: person = {
             "first_name": "John",
             "last_name": "Doe",
             "age": 30,
             "city": "New York"
         }

         print("First Name:", person["first_name"])
         print("Last Name:", person["last_name"])
         print("Age:", person["age"])
         print("City:", person["city"])

         First Name: John
         Last Name: Doe
         Age: 30
         City: New York
```

Figure 14: Dictionary

Likewise, in our code, we have used the dictionary to store inputs for the Black Scholes option pricing equation in Figure 15 below:

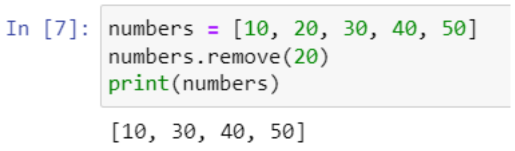


```
inputs = {
    'S': float(self.lineEdit_SpotPrice.text()),
    'K': float(self.lineEdit_StrikePrice.text()),
    'T': float(self.lineEdit_TimetoExpiry.text()),
    'r': float(self.lineEdit_InterestRate.text()) / 100,
    'q': float(self.lineEdit_Dividend.text()) / 100,
    'sigma': float(self.lineEdit_Volatility.text()) / 100
}
```

Figure 15: Dictionary Option Calculator

4.5.2 List

List is a collection of comma-separated items between square brackets, all stored in a single variable. They may contain different types of data like string or integer but usually all items have the same type. They are also known as mutable sequences whereby elements can be added, removed or changed. The following is a list of integer literals and illustrates the addition and removal of elements:



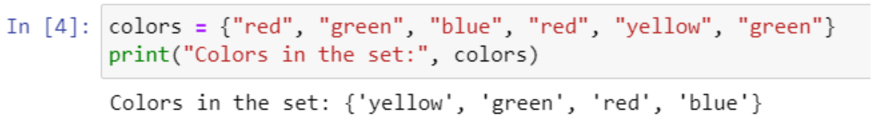
```
In [7]: numbers = [10, 20, 30, 40, 50]
        numbers.remove(20)
        print(numbers)

        [10, 30, 40, 50]
```

Figure 16: List

4.5.3 Set

Set refers to an unordered collection with no duplicate elements. It can easily be created using curly brackets or the set() function and is mainly used for membership testing and eliminating duplicate entries. As opposed to lists, sets are unindexed and hence, a programmer will not be able to access each individual element. A simple example of a set is shown below:

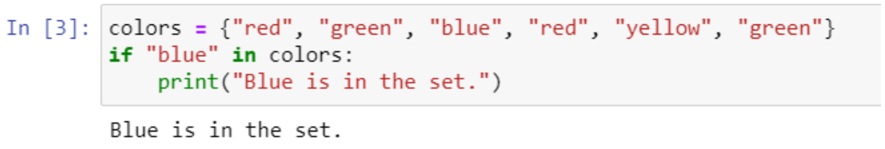
When the set is being printed out, only unique values are returned and any duplicates are removed.



```
In [4]: colors = {"red", "green", "blue", "red", "yellow", "green"}
        print("Colors in the set:", colors)

        Colors in the set: {'yellow', 'green', 'red', 'blue'}
```

Figure 17: Set

Sets are also known for membership testing to check whether an element is present in the set. In this case, to check if “Blue” is in the set, you may follow the code below:

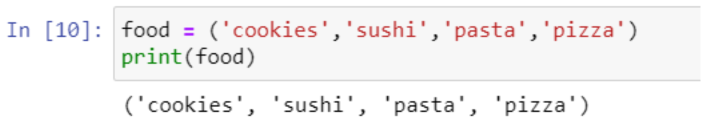


```
In [3]: colors = {"red", "green", "blue", "red", "yellow", "green"}
        if "blue" in colors:
            print("Blue is in the set.")

        Blue is in the set.
```

Figure 18: Checking if something is in the Set

4.5.4 Tuple

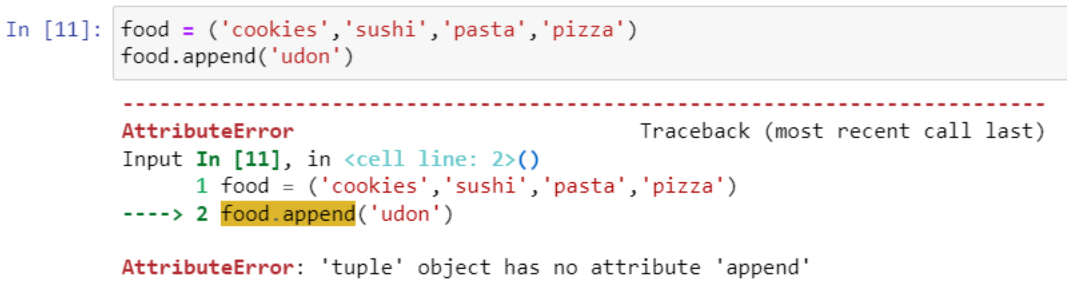
Similar to a list, tuple is an ordered collection of elements but the main difference is that the elements cannot be changed, added or removed. It is also created using circular brackets rather than square brackets and the commas between each element is compulsory. Tuples are also indexed which allow a programmer such as yourself to access individual elements. A simple example of a tuple is shown below in Figure 19:



```
In [10]: food = ('cookies','sushi','pasta','pizza')
         print(food)

         ('cookies', 'sushi', 'pasta', 'pizza')
```

Figure 19: Tuple

As mentioned, tuples are immutable sequences which means that elements can neither be added nor removed. By doing so, it will result in a TypeError as shown in Figure 20 below.



```
In [11]: food = ('cookies','sushi','pasta','pizza')
         food.append('udon')

         ---------------------------------------------------------------------------
         AttributeError                            Traceback (most recent call last)
         Input In [11], in <cell line: 2>()
               1 food = ('cookies','sushi','pasta','pizza')
         ----> 2 food.append('udon')

         AttributeError: 'tuple' object has no attribute 'append'
```

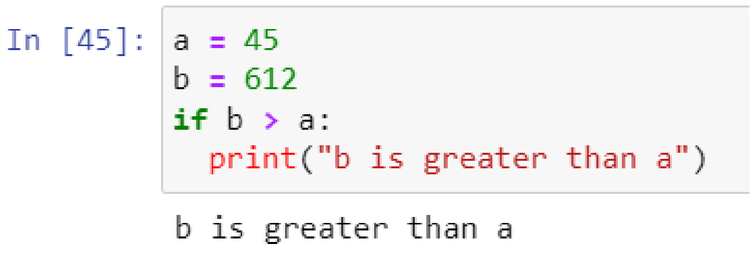
Figure 20: Immutable Tuple

4.6 Conditional Statements

Conditional Statements are used in Python to make decisions based on whether a given condition is 'True' or 'False'. It is fundamental to control the flow of the program and make it responsive to different situations which will help you create a more flexible and interactive code.

4.6.1 If Statement

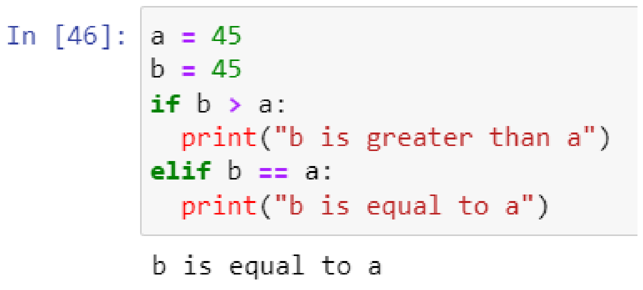
The most fundamental of conditional statements is the *if* statement. It evaluates the expression one by one until one is found to be true in which a result will be returned and no other part of the *if* statement is executed. In Python, it is critical that the *if* statement is followed by a colon and an indentation in the next line otherwise it will return a SyntaxError. This is illustrated below in Figure 21:



```
In [45]: a = 45
         b = 612
         if b > a:
           print("b is greater than a")

         b is greater than a
```

Figure 21: If Statements

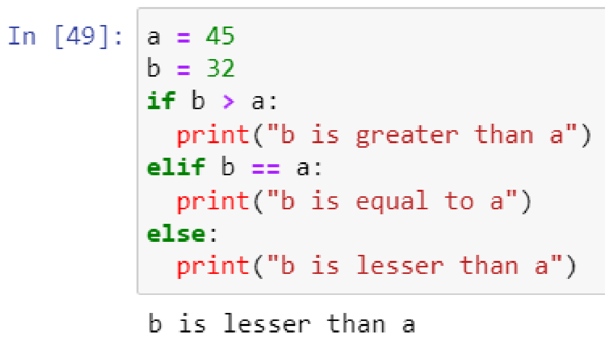
To supplement the if statement, there is also the *elif* keyword which translates to “if the previous conditions were not true, try this other condition”



```
In [46]: a = 45
         b = 45
         if b > a:
           print("b is greater than a")
         elif b == a:
           print("b is equal to a")

         b is equal to a
```

Figure 22: Elif

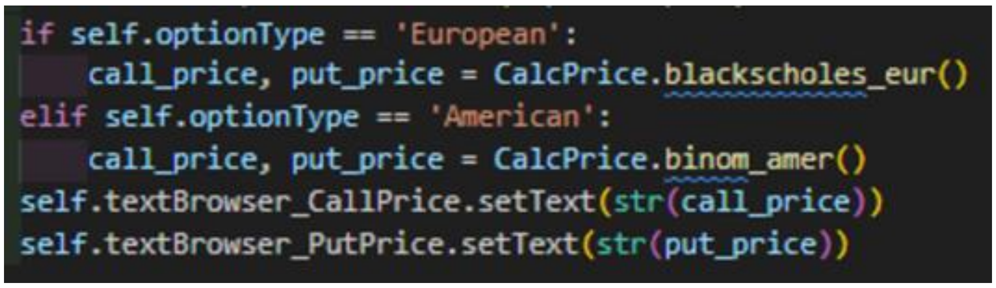
On the other hand, the *else* keyword catches anything that was not caught by the preceding conditions.



```
In [49]: a = 45
         b = 32
         if b > a:
           print("b is greater than a")
         elif b == a:
           print("b is equal to a")
         else:
           print("b is lesser than a")

         b is lesser than a
```

Figure 23: Else

If-elif statements were used in our binomial tree code to give users autonomy to decide which type of option they would like to calculate between European and American options as shown in Figure 24 below.



```
if self.optionType == 'European':
    call_price, put_price = CalcPrice.blackscholes_eur()
elif self.optionType == 'American':
    call_price, put_price = CalcPrice.binom_amer()
self.textBrowser_CallPrice.setText(str(call_price))
self.textBrowser_PutPrice.setText(str(put_price))
```

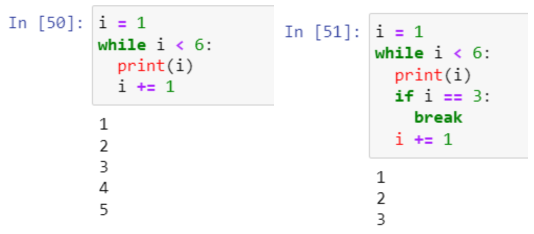
Figure 24: Conditional Statement Options Calculator

4.7 Loop

Python has two primitive loop commands: *while* loops and *for* loops.

4.7.1 While Loop

With the *while* loop, we can execute a set of statements as long as a condition is true. Unlike the *if* statement, the *while* loop can be imagined as a repeating conditional statement where it will repeatedly test the expression and execute a block of code unless the expression is false in which the loop will automatically terminate and execute the *else* clause. A *break* statement can also be incorporated to terminate the loop without executing the *else* clause as shown in Figure 25 below.



```
In [50]: i = 1
         while i < 6:
           print(i)
           i += 1

1
2
3
4
5
```

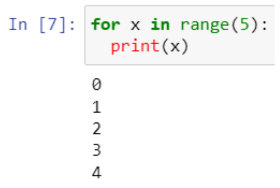
```
In [51]: i = 1
         while i < 6:
           print(i)
           if i == 3:
             break
           i += 1

1
2
3
```

Figure 25: Break Statments

4.7.2 For Loop

The second Python loop is the *for* loop which is used to iterate over the elements of a sequence or other iterable object. An iterable is an object which can be looped over, such as lists, tuples, sets, dictionaries, strings, etc. Variable takes on the value of the next element in the iterable each time through the loop. When all the elements in the iterable have been exhausted, the *else* clause is executed and the loop terminates. Likewise, a *break* statement can also be incorporated to terminate the loop. While they may be remotely similar, *for* and *while* loops are different in that *for* loops are designed for iterating over a sequence of items but *while* loops are used when the number of iterations is unknown and there is a presence of a specific condition that needs to be met.

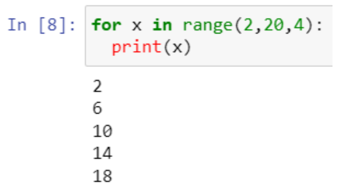
To loop a set of code a specific number of times, we can use the range() function. The range () function returns a sequence of numbers, starting from 0 by default and increments by 1 (unless otherwise stated) until it ends at a specified number. Let us see an example below that incorporates the range() function:



```
In [7]: for x in range(5):
          print(x)

0
1
2
3
4
```

Figure 26: range( ) Function

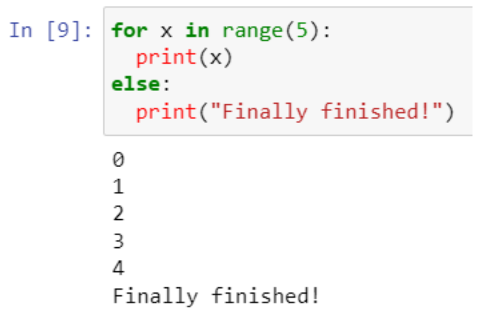
It is possible to also specify the starting value and increment value by adding two more parameters to the range() function. In this example, the starting number is 2 and the increment value is 4 as shown in Figure 27 below.



```
In [8]: for x in range(2,20,4):
          print(x)

2
6
10
14
18
```

Figure 27: Specifying Step

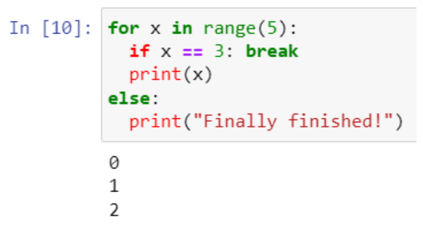
By adding an *else* clause in a *for* loop, it specifies a block of code to be executed only when the loop is finished. In this case, the words “Finally finished!” will only be printed after all numbers from 0 to 5 have been printed out.



```
In [9]: for x in range(5):
          print(x)
        else:
          print("Finally finished!")

        0
        1
        2
        3
        4
        Finally finished!
```

<u>Figure 28: Adding Else Clause</u>

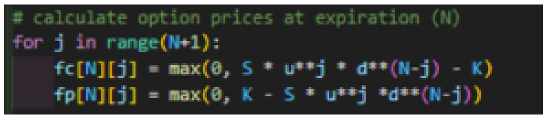
However, if we incorporate a *break* statement, the *else* block will not be executed as the loop terminates after the *break* statement.



```
In [10]: for x in range(5):
           if x == 3: break
           print(x)
         else:
           print("Finally finished!")

         0
         1
         2
```

<u>Figure 29: Break Statement</u>

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



```
# calculate option prices at expiration (N)
for j in range(N+1):
    fc[N][j] = max(0, S * u**j * d**(N-j) - K)
    fp[N][j] = max(0, K - S * u**j *d**(N-j))
```

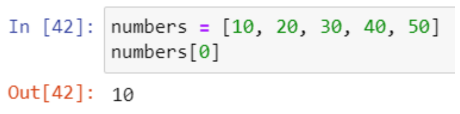
<u>Figure 30: For Loop Options Calculator</u>

4.8 Advanced Concepts

We will now be delving into more advanced concepts that were not covered in the previous sections but were commonly used in our Binomial Tree code.

4.8.1 List Indexing

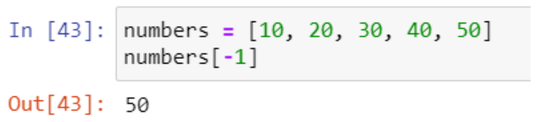
Firstly, as an extension of lists which was covered above, list indexing allows you to identify and refer to the position of a specific element in a list. In our example above, our list had 5 elements, namely [10, 20, 30, 40, 50]. In Python, the first element in the list takes the value of “0” and to get the first element, you will have to do the following:



```
In [42]: numbers = [10, 20, 30, 40, 50]
         numbers[0]

Out[42]: 10
```

<u>Figure 31: List Indexing</u>

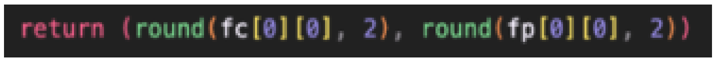
Indexes may also be negative numbers to represent the start of counting from the right. It begins from -1 which is associated with the last element and decreases progressively as it works its way to the first element. This code should pain a clearer picture of the concept:



```
In [43]: numbers = [10, 20, 30, 40, 50]
         numbers[-1]

Out[43]: 50
```

Figure 32: Beginning from the End

List indexing was used in our code to obtain the option price at the root of the binomial tree as shown:



```
return (round(fc[0][0], 2), round(fp[0][0], 2))
```

Figure 33: List Indexing Option Calculator

4.8.2 List Comprehension

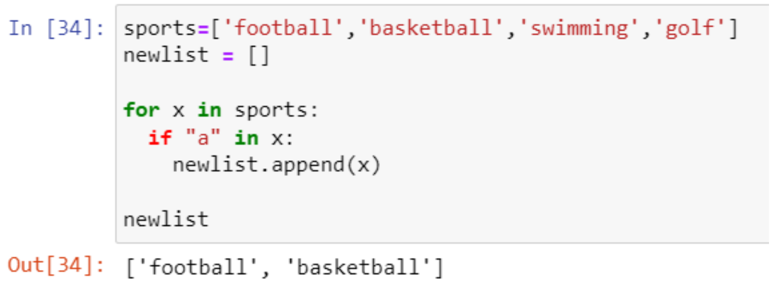
Another extended feature of a list is list comprehension which offers a shorter syntax when you want to create a new list based on the values of an existing list. To create this new list, you will generally have to follow this syntax:

newlist = [*expression* for *item* in *iterable* if *condition == True*]

However, the condition is optional and can be omitted while the iterable can take on the form of a list, tuple or even the range() function.

Let's say you have a list of sports but you want a new list containing only sports with the letter 'a' in it, you can employ list comprehension rather than writing a for statement with a conditional test inside.

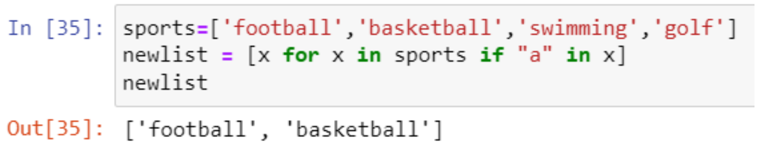
Without list comprehension, your code will look something like this:



```
In [34]: sports=['football','basketball','swimming','golf']
         newlist = []

         for x in sports:
           if "a" in x:
             newlist.append(x)

         newlist

Out[34]: ['football', 'basketball']
```

Figure 34: No List Comprehension

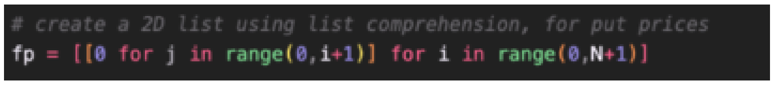
However, with list comprehension, you can do all that with only one line of code:



```
In [35]: sports=['football','basketball','swimming','golf']
         newlist = [x for x in sports if "a" in x]
         newlist

Out[35]: ['football', 'basketball']
```

Figure 35: With List Comprehension

An example of list comprehension can also be found in our Binomial Tree code as shown below:

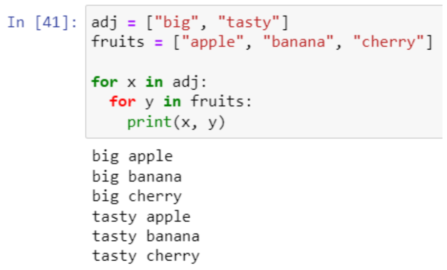


```
# create a 2D list using list comprehension, for put prices
fp = [[0 for j in range(0,i+1)] for i in range(0,N+1)]
```

Figure 36: List comprehension Option Calculator

4.8.3 Nested For Loop

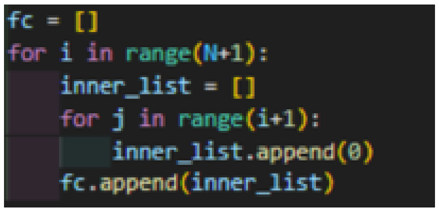
A nested for loop is essentially a for loop inside a for loop. The “inner loop” will be executed one time for each iteration of the “outer loop”.



```
In [41]: adj = ["big", "tasty"]
         fruits = ["apple", "banana", "cherry"]

         for x in adj:
           for y in fruits:
             print(x, y)

         big apple
         big banana
         big cherry
         tasty apple
         tasty banana
         tasty cherry
```

Figure 37: Nested For Loop

This was how we used it in our binomial tree option pricing method where an empty list was first created as denoted by the empty square brackets “[]” to hold a 2D list which is used for creating a matrix-like structure. The nested for loops are then used to compute the asset’s potential values at all possible nodes (inner loop) of each time step (outler loop) while taking into consideration the upward and downward cases from previous nodes.

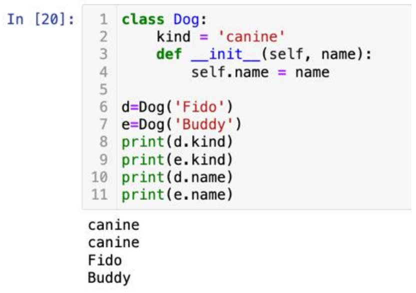


```
fc = []
for i in range(N+1):
    inner_list = []
    for j in range(i+1):
        inner_list.append(0)
    fc.append(inner_list)
```

Figure 38: Up and Down Nodes

4.9 Classes

Python operates as an Object-Oriented Programming language, where objects possess distinct properties and methods. In Python, a class functions akin to an object constructor, serving as a "blueprint" that defines the structure for creating objects with specific attributes and behaviors. An example of class implementation can be found in the code shown below:



```
In [20]: class Dog:
             kind = 'canine'
             def __init__(self, name):
                 self.name = name

         d=Dog('Fido')
         e=Dog('Buddy')
         print(d.kind)
         print(e.kind)
         print(d.name)
         print(e.name)

         canine
         canine
         Fido
         Buddy
```

Figure 39: Classes

In the code example, a class “Dog” was implemented. The class has a constructor method (‘__init__’) that initializes an instance of the class. It takes two parameters, ‘self’ and ‘name’. Within the constructor, an instance attribute ‘name’ is created for each instance of the class and is initialized with the value passed to the constructor.

There are two key classes used in this project, which are the Main Class and OptionCalculator Class. The class, ‘Main’ is used to create the project’s Graphical User Interface, while the class ‘OptionCalculator’ will be used to calculate the price of an option.

4.10 Import

4.10.1 Libraries

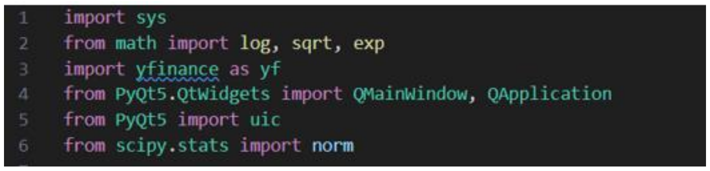
Python’s standard libraries, also known as modules or packages, are a collection of pre-written codes that provide a set of functions for various tasks. It allows programmers like you and me to perform a wide range of tasks without having to type out the code from scratch, hence helping us to save time and effort.

There are two key standard libraries that were used in this project which are the sys library and math library. Sys is a module that provides access to some variables and functions used while math is a module that provides access to mathematical functions such as log, square root and exponential.

In addition to the standard library, Python also has an ecosystem of external libraries that can be installed and used to perform tasks. One such example that we used in our code is yfinace which fetches stock market data from Yahoo Finance. Scipy.stat is also a module from an external library which is used for a wide range of statistical functions and distributions for statistical analysis.

4.10.2 Importing Statements

Functions provided by the libraries cannot be used without first importing them hence, we will need to import these libraries which can be easily done as shown below:



```
import sys
from math import log, sqrt, exp
import yfinance as yf
from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5 import uic
from scipy.stats import norm
```

Figure 40: Import Statements

5. Designing the Graphical User Interface (GUI)

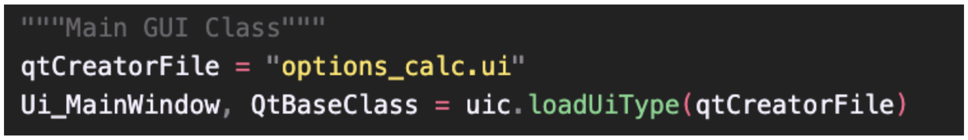
5.1 Introduction to PyQt5

PyQt5 is a robust Python library that facilitates the development of graphical user interfaces (GUIs) with seamless integration into Python applications. Leveraging the Qt framework, PyQt5 empowers developers to create sophisticated desktop applications with a wide array of features and functionalities. Offering a comprehensive set of tools and widgets, PyQt5 streamlines the process of building interactive and visually appealing applications. With its versatility and cross-platform compatibility, PyQt5 enables developers to craft dynamic user interfaces for Windows, Linux, and macOS environments, making it an invaluable resource for those seeking to enhance the user experience of their Python applications.

5.1.1 PyQt5.uic

PyQt5.uic is a module within the PyQt5 library that stands for "User Interface Compiler." It provides functionality for dynamically loading and working with user interface files created using the Qt Designer tool. Qt Designer allows developers to design and layout graphical user interfaces visually. The .ui files generated by Qt Designer contain the structure and design of the user interface.

In our application, the UI File "options_calc.ui" is loaded to set up a Graphical User Interface (GUI) for the Option Pricing Application. The visual tool, Qt Designer, allows you to design and build user interfaces for desktop applications. The code is written as shown:



```
"""Main GUI Class"""
qtCreatorFile = "options_calc.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)
```

Figure 41: Main GUI Class

The code begins by identifying and naming a user interface file named "options_calc.ui" and assigning it to a variable called "qtCreatorFile." This file is essentially a visual blueprint created using a design tool called Qt Designer.

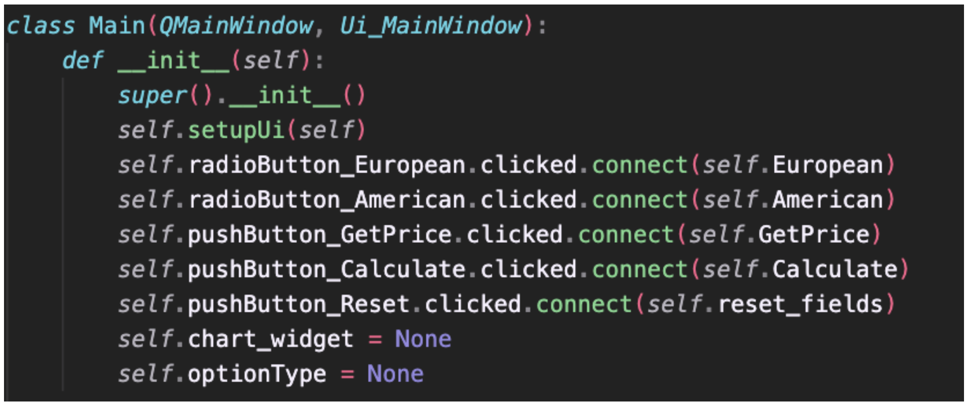
The next line of code utilises a module called "uic" to take the design from the "options_calc.ui" file and transform it into something the program can work with. This process produces two important things:

1. "Ui_MainWindow" represents the main window of the user interface, which is like the central part of the application where you interact with it.

2. "QtBaseClass" is the foundational class that "Ui_MainWindow" is based on, much like how a book's content (Ui_MainWindow) is built on top of its cover (QtBaseClass).

These two objects, "Ui_MainWindow" and "QtBaseClass," are crucial because they enable the program to create and control the user interface and its elements (like buttons and text fields) later in the script. Essentially, they help the program understand how the user interface should look and behave.

5.1.2 Creating the GUI



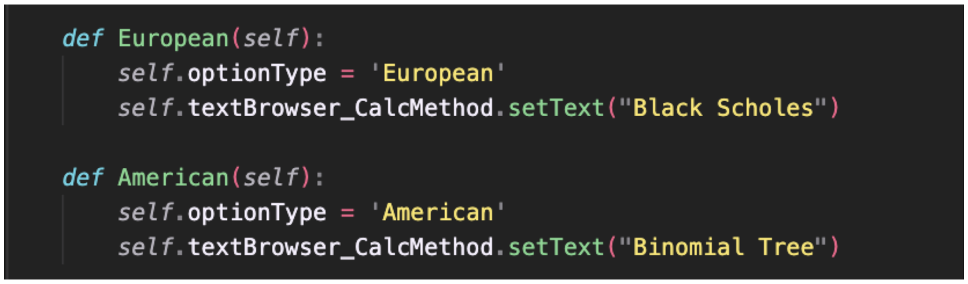
```
class Main(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.radioButton_European.clicked.connect(self.European)
        self.radioButton_American.clicked.connect(self.American)
        self.pushButton_GetPrice.clicked.connect(self.GetPrice)
        self.pushButton_Calculate.clicked.connect(self.Calculate)
        self.pushButton_Reset.clicked.connect(self.reset_fields)
        self.chart_widget = None
        self.optionType = None
```

Figure 42: Crafting the GUI

In our application, the class 'Main', inherits both 'QMainWindow' and 'Ui_MainWindow', and serves as the main window for the PyQt5 GUI. We first initiliase the Main class using the '__init__' method, followed by calling 'super().__init__()' to ensure that both parent classes, 'QMainWindow' and 'Ui_MainWindow' have been executed.

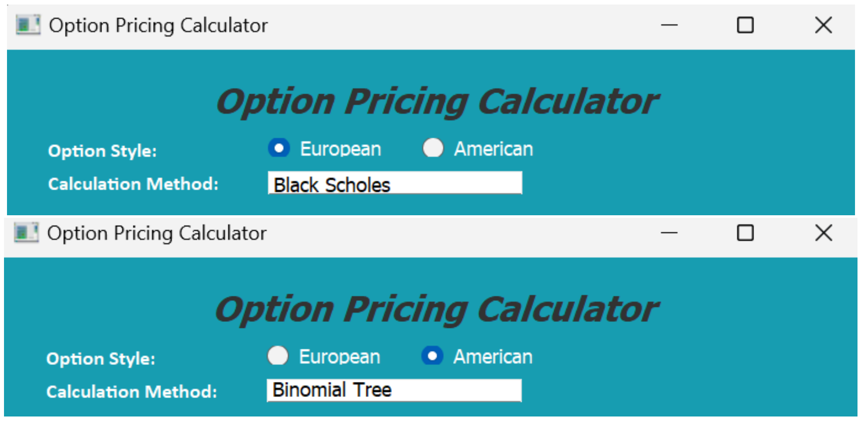
Subsequently, we create and connect the GUI's radio button to the corresponding methods, European and American.

Additionally, we also created and connected the GUI's push button to the corresponding methods, GetPrice, Calculate and reset fields.



```
def European(self):
    self.optionType = 'European'
    self.textBrowser_CalcMethod.setText("Black Scholes")

def American(self):
    self.optionType = 'American'
    self.textBrowser_CalcMethod.setText("Binomial Tree")
```

Figure 43: Push Buttons



Option Pricing Calculator

Option Pricing Calculator

Option Style: [✓ European] American

Calculation Method: Black Scholes

Option Pricing Calculator

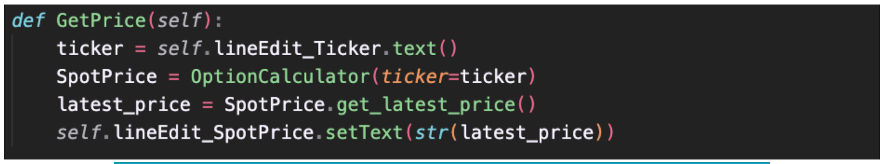
Option Pricing Calculator

Option Style: European [✓ American]

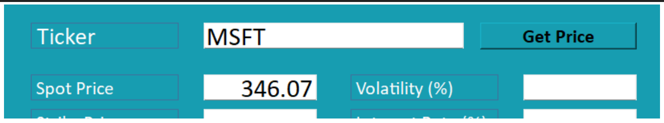
Calculation Method: Binomial Tree

Figure 44: Output of Push Buttons

With reference to the above code, 'European' and 'American' methods set and update the text field of the GUI according to the user's choice.



```
def GetPrice(self):
    ticker = self.lineEdit_Ticker.text()
    SpotPrice = OptionCalculator(ticker=ticker)
    latest_price = SpotPrice.get_latest_price()
    self.lineEdit_SpotPrice.setText(str(latest_price))
```



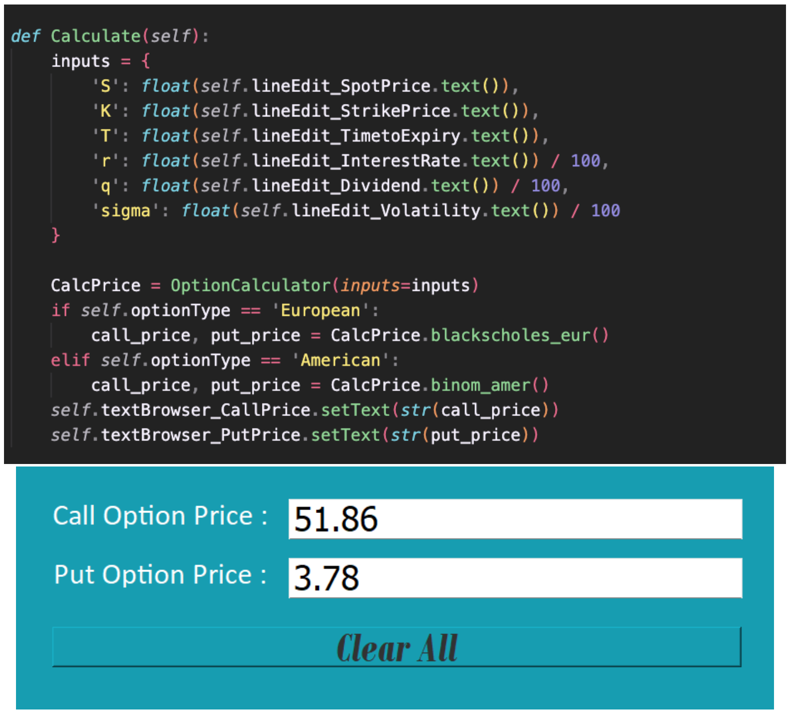
Ticker MSFT

Spot Price 346.07

Volatility (%)

Figure 45: Output of Ticker's Spot Price

In this function, the variable ‘ticker’ is assigned to the user’s input in the GUI. The ‘ticker’ is then passed to OptionCalculator as a parameter argument to get the latest price of the chosen company using the get_latest_price method in OptionCalculator. This value is assigned to ‘latest_price’ and is shown on text field next to Spot Price.



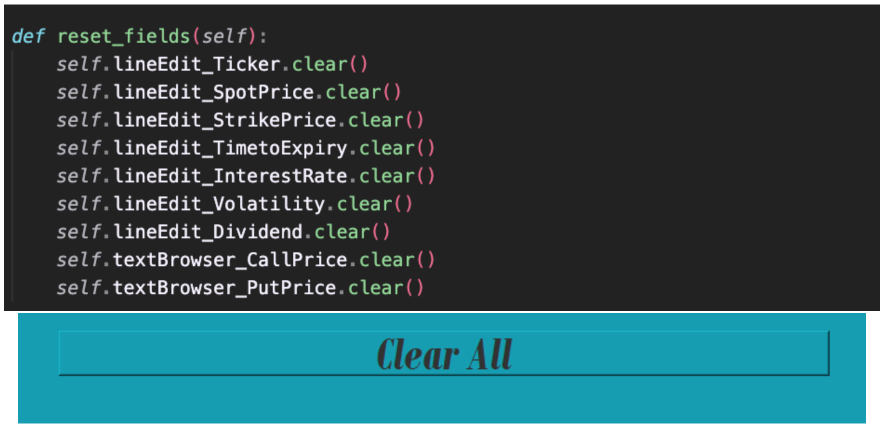
```
def Calculate(self):
    inputs = {
        'S': float(self.lineEdit_SpotPrice.text()),
        'K': float(self.lineEdit_StrikePrice.text()),
        'T': float(self.lineEdit_TimetoExpiry.text()),
        'r': float(self.lineEdit_InterestRate.text()) / 100,
        'q': float(self.lineEdit_Dividend.text()) / 100,
        'sigma': float(self.lineEdit_Volatility.text()) / 100
    }

    CalcPrice = OptionCalculator(inputs=inputs)
    if self.optionType == 'European':
        call_price, put_price = CalcPrice.blackscholes_eur()
    elif self.optionType == 'American':
        call_price, put_price = CalcPrice.binom_amer()
    self.textBrowser_CallPrice.setText(str(call_price))
    self.textBrowser_PutPrice.setText(str(put_price))
```

Call Option Price : 51.86

Put Option Price : 3.78

Figure 46: Calculate Function

The ‘Calculate’ function gathers user’s inputs from the GUI and calculates the option price using either ‘blackscholes_eur’ or ‘binom_amer’, a method in the OptionCalculator class. Subsequently, the call and put price of the option will be shown in the GUI.

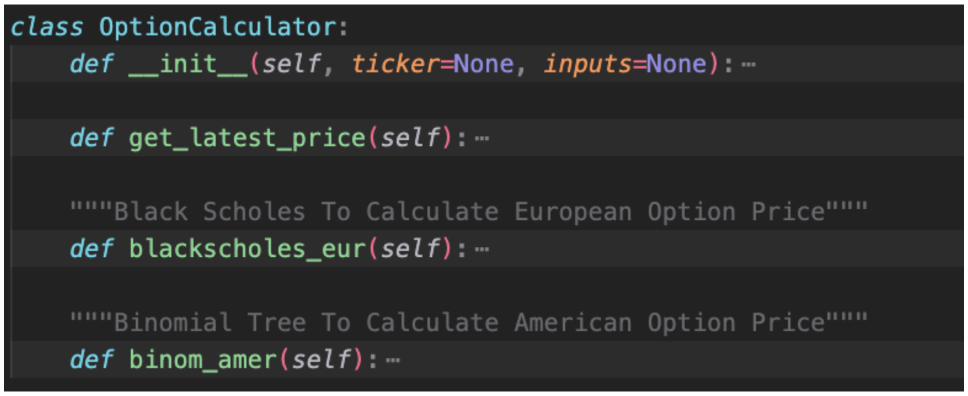


```
def reset_fields(self):
    self.lineEdit_Ticker.clear()
    self.lineEdit_SpotPrice.clear()
    self.lineEdit_StrikePrice.clear()
    self.lineEdit_TimetoExpiry.clear()
    self.lineEdit_InterestRate.clear()
    self.lineEdit_Volatility.clear()
    self.lineEdit_Dividend.clear()
    self.textBrowser_CallPrice.clear()
    self.textBrowser_PutPrice.clear()
```

Figure 47: Resetting Fields

Lastly, the reset_fields function clears all input fields and result text field from the GUI. The reset_fields function is linked to “Clear All” button in the application. This button essentially resets the application.

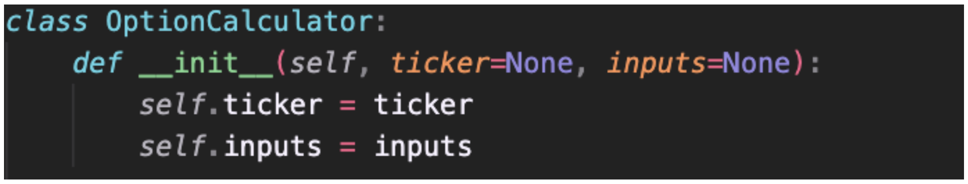
5.2 Options Pricing Class



```python
class OptionCalculator:
    def __init__(self, ticker=None, inputs=None):⋯

    def get_latest_price(self):⋯

    """Black Scholes To Calculate European Option Price"""
    def blackscholes_eur(self):⋯

    """Binomial Tree To Calculate American Option Price"""
    def binom_amer(self):⋯
```

Figure 48: Option Pricing Class

Imagine you're in a financial world where you can invest in options. The OptionCalculator Class acts as your mathematical assistant in the world. It helps you figure out how much these options are worth.



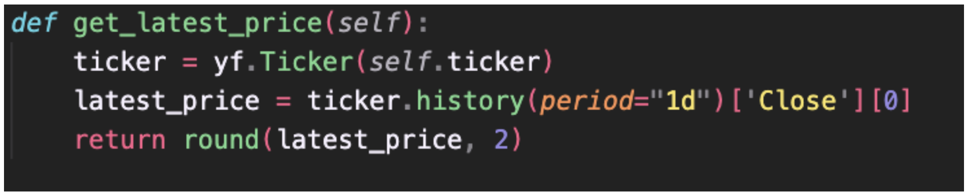
```python
class OptionCalculator:
    def __init__(self, ticker=None, inputs=None):
        self.ticker = ticker
        self.inputs = inputs
```

Figure 49: __init__()

In the above figure, we can see that the OptionsCalculator starts with an instantiation method, '*__init__()*'. This method is the construction of creating instances of the class.

There are 2 optional parameters, "*ticker*" and "*inputs*". The "*ticker*" parameter represents the stock symbol, and "*inputs*" is a dictionary containing various financial parameters needed for option pricing. Additionally, we can also see 2 assignment statements, "*self.ticker*" and "*self.inputs*".

5.2.1 Fetching Latest Stock Price ('get_latest_price ' method)

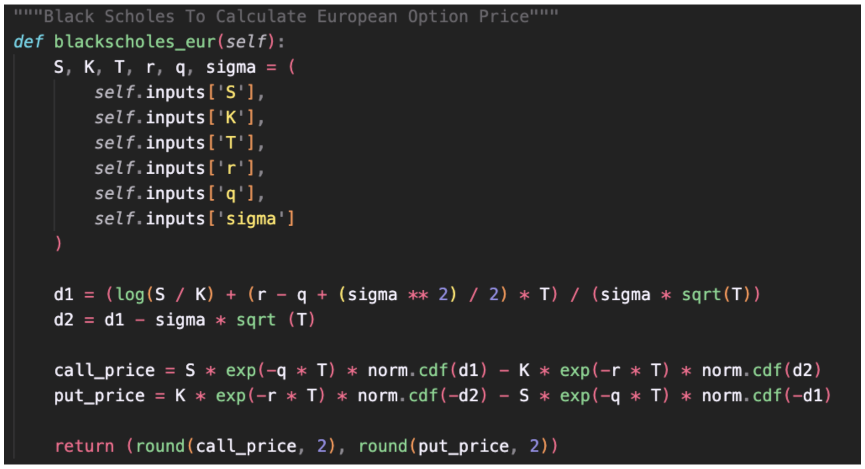


```python
def get_latest_price(self):
    ticker = yf.Ticker(self.ticker)
    latest_price = ticker.history(period="1d")['Close'][0]
    return round(latest_price, 2)
```

Figure 50: Fetching Latest Stock Price

This function checks the most recent price of the company's ticker via the yfinance library. The first line of code assigns the company ticker to the variable "*ticker*". Subsequently, the second line of code assigns the most recent closing stock price of "*ticker*" to the variable "*latest_price*". The obtained price is then rounded to two decimal places and returned.

5.3 Options Pricing: Black-Scholes Method for European Options (‘blackscholes_eur’ method)



```
"""Black Scholes To Calculate European Option Price"""
def blackscholes_eur(self):
    S, K, T, r, q, sigma = (
        self.inputs['S'],
        self.inputs['K'],
        self.inputs['T'],
        self.inputs['r'],
        self.inputs['q'],
        self.inputs['sigma']
    )

    d1 = (log(S / K) + (r - q + (sigma ** 2) / 2) * T) / (sigma * sqrt(T))
    d2 = d1 - sigma * sqrt (T)

    call_price = S * exp(-q * T) * norm.cdf(d1) - K * exp(-r * T) * norm.cdf(d2)
    put_price = K * exp(-r * T) * norm.cdf(-d2) - S * exp(-q * T) * norm.cdf(-d1)

    return (round(call_price, 2), round(put_price, 2))
```

Figure 51: Black Scholes Method

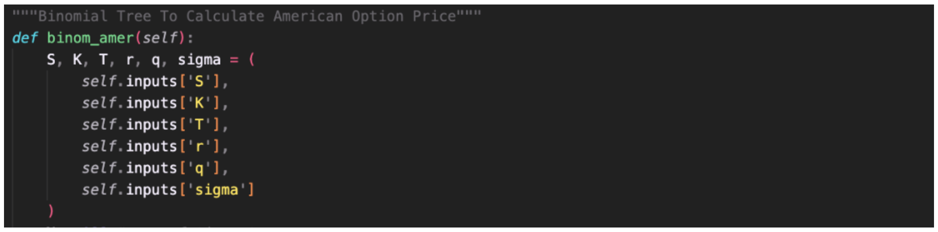
This function calculates European option prices using the Black-Scholes formula. It utilises the data stored in the “*inputs*” variable by indexing. Current stock price ‘*S*’, Strike price ‘*K*’, Time to Maturity ‘*T*’, risk-free rate ‘*r*’, dividend yield ‘*q’* and volatility ‘*sigma’* are used in this function to calculate respective call and put options prices.

At the heart of the Black-Scholes formula, we calculate the two values ‘d1’ and ‘d2’ by using the variables of ‘*S*’, ‘*K*’, ‘*T*’, ‘*r*’, ‘*q’* and ‘*sigma’.*

With *‘d1’* and *‘d2’* calculated, we proceed to compute the call and put option prices using the Black-Scholes formula.

Finally, the function rounds the respective call and put option prices to 2 decimal places and returns it.

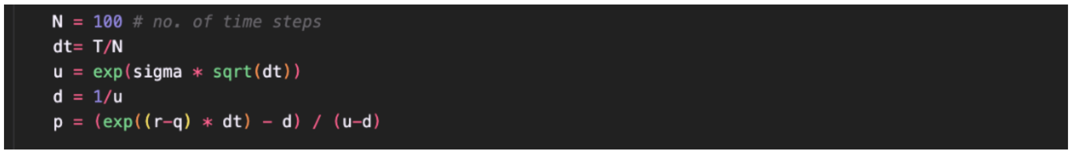
5.4 Options Pricing: Binomial Tree Method for American Options (*‘binom_amer’* method)



```
"""Binomial Tree To Calculate American Option Price"""
def binom_amer(self):
    S, K, T, r, q, sigma = (
        self.inputs['S'],
        self.inputs['K'],
        self.inputs['T'],
        self.inputs['r'],
        self.inputs['q'],
        self.inputs['sigma']
    )
```

Figure 52: Binomial Tree Method

Like the Black-Scholes method, Current stock price ‘*S*’, Strike price ‘*K*’, Time to Maturity ‘*T*’, risk-free rate ‘*r*’, dividend yield ‘*q’* and volatility ‘*sigma’* are used in this function to calculate respective call and put options prices.



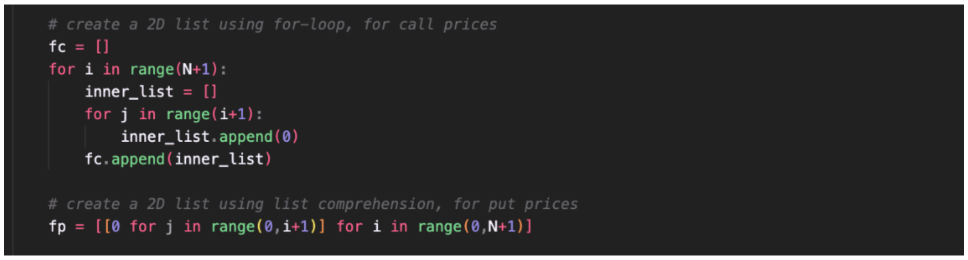
```
N = 100 # no. of time steps
dt= T/N
u = exp(sigma * sqrt(dt))
d = 1/u
p = (exp((r-q) * dt) - d) / (u-d)
```

Figure 53: Calculation

The function starts by defining variable '*N*' to be the number of time steps. The time increment '*dt*' is calculated by diving Time to Maturity '*T*' by the number of time steps '*N*'.

Two factors of '*u' and 'd*' represent the up and down movements of the stock price. These factors are calculated using the exponential function and square root of the time increment.

Next, the risk-neutral probability '*p*' is calculated based on the up and down factors. It represents the probability of an up movement in the binomial tree.

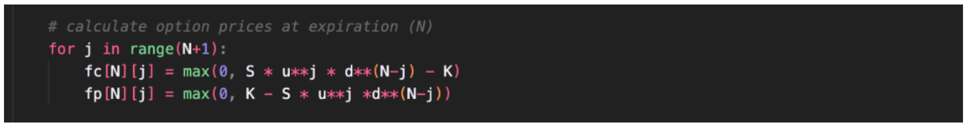


```
# create a 2D list using for-loop, for call prices
fc = []
for i in range(N+1):
    inner_list = []
    for j in range(i+1):
        inner_list.append(0)
    fc.append(inner_list)

# create a 2D list using list comprehension, for put prices
fp = [[0 for j in range(0,i+1)] for i in range(0,N+1)]
```

Figure 54: Risk Neutral Probability

The next few lines of code create two arrays, fc and fp using two different methods. We created these two arrays using different methods: A nested for-loop for array 'fc' and a list comprehension for array 'fp'.

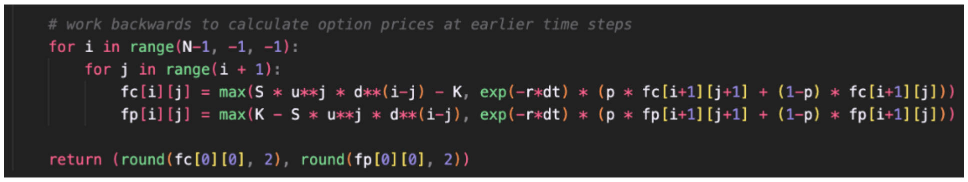
Array 'fc' will be used for call options, whereas array 'fp' will be used for put options.



```
# calculate option prices at expiration (N)
for j in range(N+1):
    fc[N][j] = max(0, S * u**j * d**(N-j) - K)
    fp[N][j] = max(0, K - S * u**j *d**(N-j))
```

Figure 55: fc vs fp

The function then calculates the option prices at the Time to Maturity 'N' for both call ('*fc*') and put ('*fp*') options.



```
# work backwards to calculate option prices at earlier time steps
for i in range(N-1, -1, -1):
    for j in range(i + 1):
        fc[i][j] = max(S * u**j * d**(i-j) - K, exp(-r*dt) * (p * fc[i+1][j+1] + (1-p) * fc[i+1][j]))
        fp[i][j] = max(K - S * u**j * d**(i-j), exp(-r*dt) * (p * fp[i+1][j+1] + (1-p) * fp[i+1][j]))

return (round(fc[0][0], 2), round(fp[0][0], 2))
```

Figure 56: Looping through Function

Next, the function works backward through the binomial tree to calculate the American option pricing by:

- Outer loop ('for i in range(N-1,-1,-1)':

The outer loop iterates backwards through the time steps, starting from the second last time step, N-1, and going down to the first time step, 0.

- Inner loop ('for j in range(i+1)'):

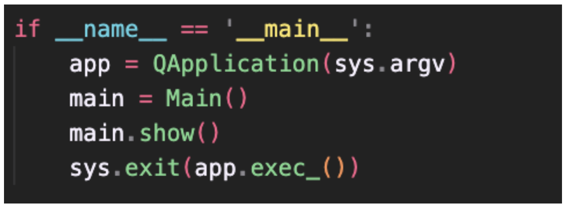
The inner loop iterates through the nodes at each time step, i. The range is set to 'i+1' to cover all possible nodes at the given time step.

- Updating option prices ('fc[i][j]' and 'fp[i][j]'):

Within the inner loop, the code updates the call option price and put option price for each node at the current time step. The option prices are calculated based on potential future values at the next time step. The calculation involve the stock price movements and the discounted future option values.

Finally, the estimated option prices for both call and put options are rounded off to 2 decimal places and returned.

5.5 UI Python Script



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

Figure 57: UI Python Script

This is the usual way to start a Qt-based application using PyQt.

In the following statement, “if __name__ == ‘__main__’:”, the Python idiom is to ensure that the code is executed only if the script is run directly, and not imported as a module in another python script.

The QApplication class serves as the central starting point for applications developed with the Qt framework. It plays a pivotal role in managing event loops, overseeing application resources, and providing access to the Qt API.

Next, sys.argv, a Python list, holds the command-line arguments passed to the script. The QApplication class relies on these arguments to initialize itself correctly, enabling seamless interaction with the application.

The instantiation of Main() represents the primary window or widget within the application. The Main class encapsulates the main functionality and user interface elements.

When main.show() is called, it triggers the display of the primary window, allowing users to interact with the application's features and functionalities.

The line sys.exit(app.exec_()) marks the initiation of the application's main event loop. This event loop actively listens for user interface events and responds accordingly. Importantly, sys.exit() ensures the proper termination of the Python interpreter when the user decides to exit the application, bringing the program to a graceful conclusion.

6. User Manual

This section will explain how you can use the code to run the program and calcualte the option price based on your selected option style, stock, and input values. Our code is stored in a python script file named “option_calc.py” and we have provided a UI file named “option_calc.ui”.

6.1 Running the Python Program

For Windows Users, use Command Prompt and navigate to the directory where “option_calc.py” is located using the `cd` command. When you are in the correct directory, execute the Python script by entering `python options_calc.py`. Press Enter and the GUI window will open up.

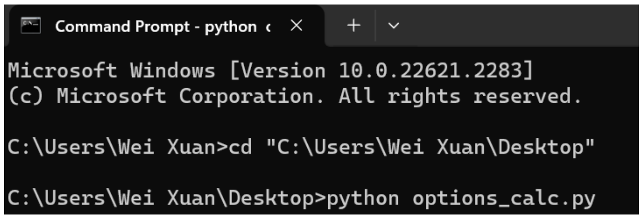

```
Command Prompt - python c
Microsoft Windows [Version 10.0.22621.2283]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Wei Xuan>cd "C:\Users\Wei Xuan\Desktop"

C:\Users\Wei Xuan\Desktop>python options_calc.py
```

Figure 58: Running the Python Program

For Mac Users, use Terminal and navigate to the directory where "option_calc.py" is located using the 'cd' command. When you are in the correct directory, execute the Python script by entering 'python3 option_calc.py'. Press Enter and the GUI window will open.

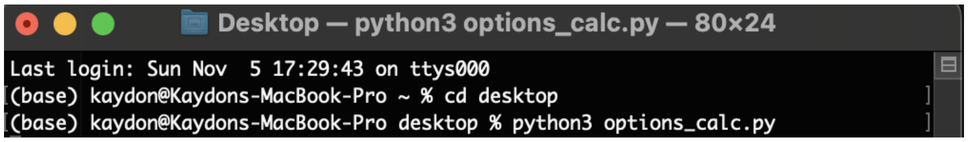
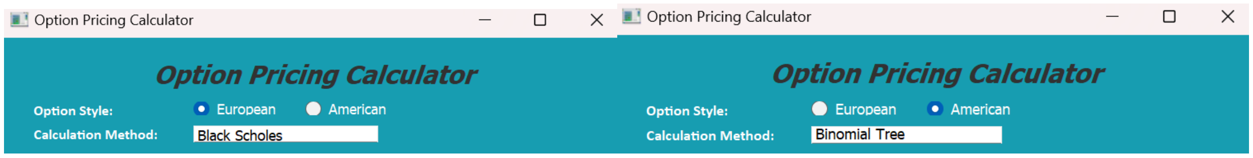

```
Desktop — python3 options_calc.py — 80×24
Last login: Sun Nov  5 17:29:43 on ttys000
(base) kaydon@Kaydons-MacBook-Pro ~ % cd desktop
(base) kaydon@Kaydons-MacBook-Pro desktop % python3 options_calc.py
```

Figure 59: Opening the GUI

6.2 Using the GUI

Firstly, you are required to select the Option Style, whether it is a European or American option. In our program, a European option will trigger the Black Scholes calculation method while an American option will trigger the Binomial Tree calculation method as seen in the display field.


Option Pricing Calculator

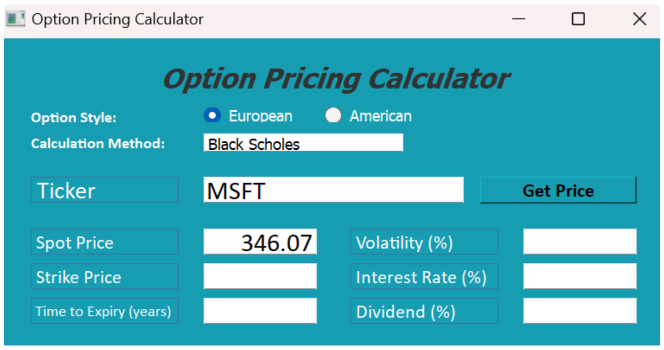
Option Style: [✓ European] American
Calculation Method: Black Scholes

Option Pricing Calculator

Option Style: European [✓ American]
Calculation Method: Binomial Tree

Figure 60: Using the GUI

Next, you can input the Ticker into the Ticker field to fetch the latest closing stock price from Yahoo Finance. The Spot Price will then be automatically filled as seen below.

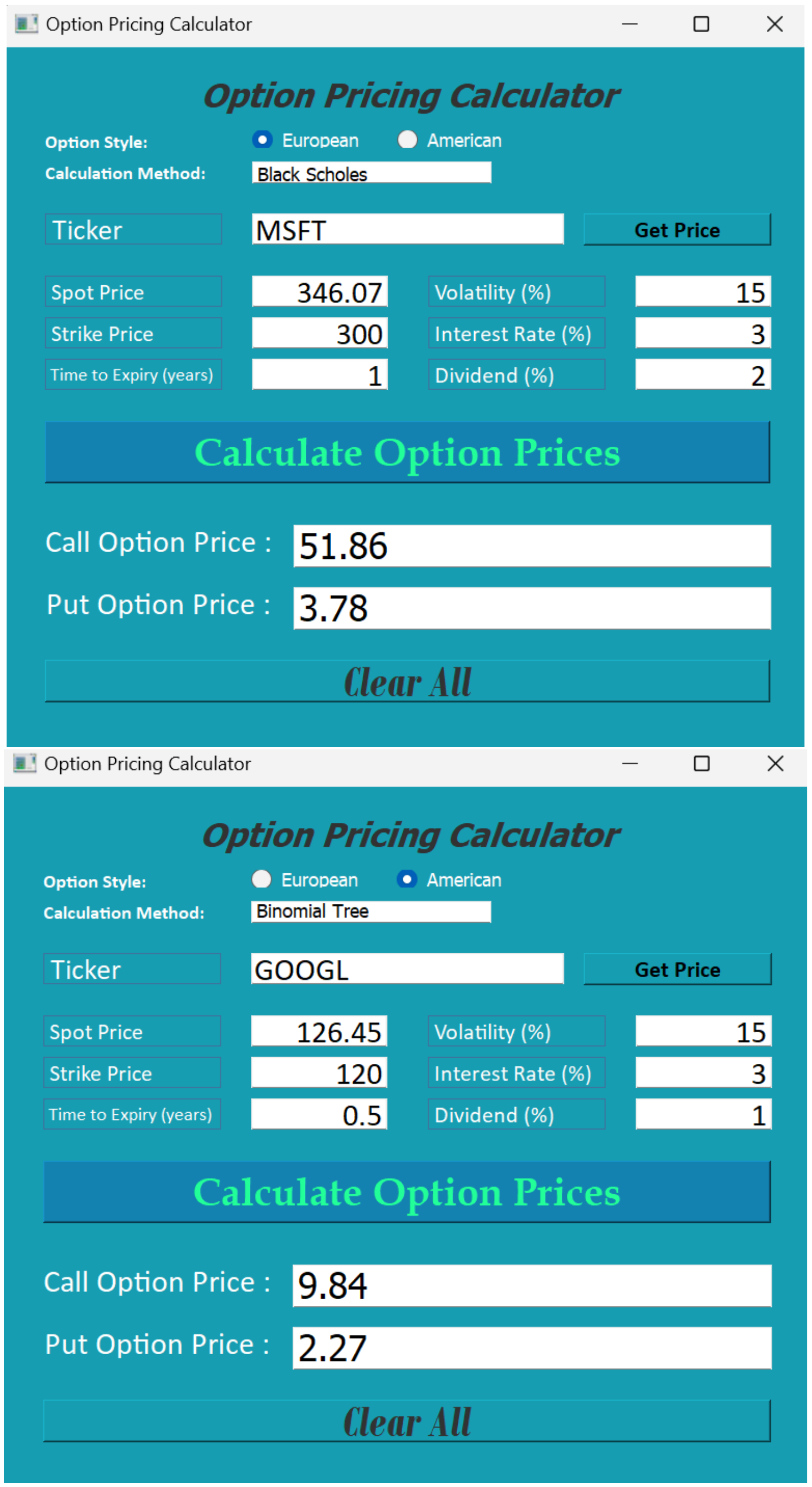

Option Pricing Calculator

Option Style: [✓ European] American
Calculation Method: Black Scholes
Ticker: MSFT

Spot Price: 346.07
Strike Price:
Time to Expiry (years):
Volatility (%):
Interest Rate (%):
Dividend (%):

Figure 61: Keying in the Inputs

You can then enter the different variables required to calculate the option price, which includes the Strike Price, Time to Expiry (years), Volatility (%), Interest Rate (%), Dividend (%). After which, press "Calculate Option Prices" to generate the call and put prices.


Option Pricing Calculator

Option Pricing Calculator

Option Style: [✓ European] American

Calculation Method: Black Scholes

Ticker: MSFT — Get Price

Spot Price: 346.07 — Volatility (%): 15

Strike Price: 300 — Interest Rate (%): 3

Time to Expiry (years): 1 — Dividend (%): 2

Calculate Option Prices

Call Option Price : 51.86

Put Option Price : 3.78

Clear All

Option Pricing Calculator

Option Pricing Calculator

Option Style: European [✓ American]

Calculation Method: Binomial Tree

Ticker: GOOGL — Get Price

Spot Price: 126.45 — Volatility (%): 15

Strike Price: 120 — Interest Rate (%): 3

Time to Expiry (years): 0.5 — Dividend (%): 1

Calculate Option Prices

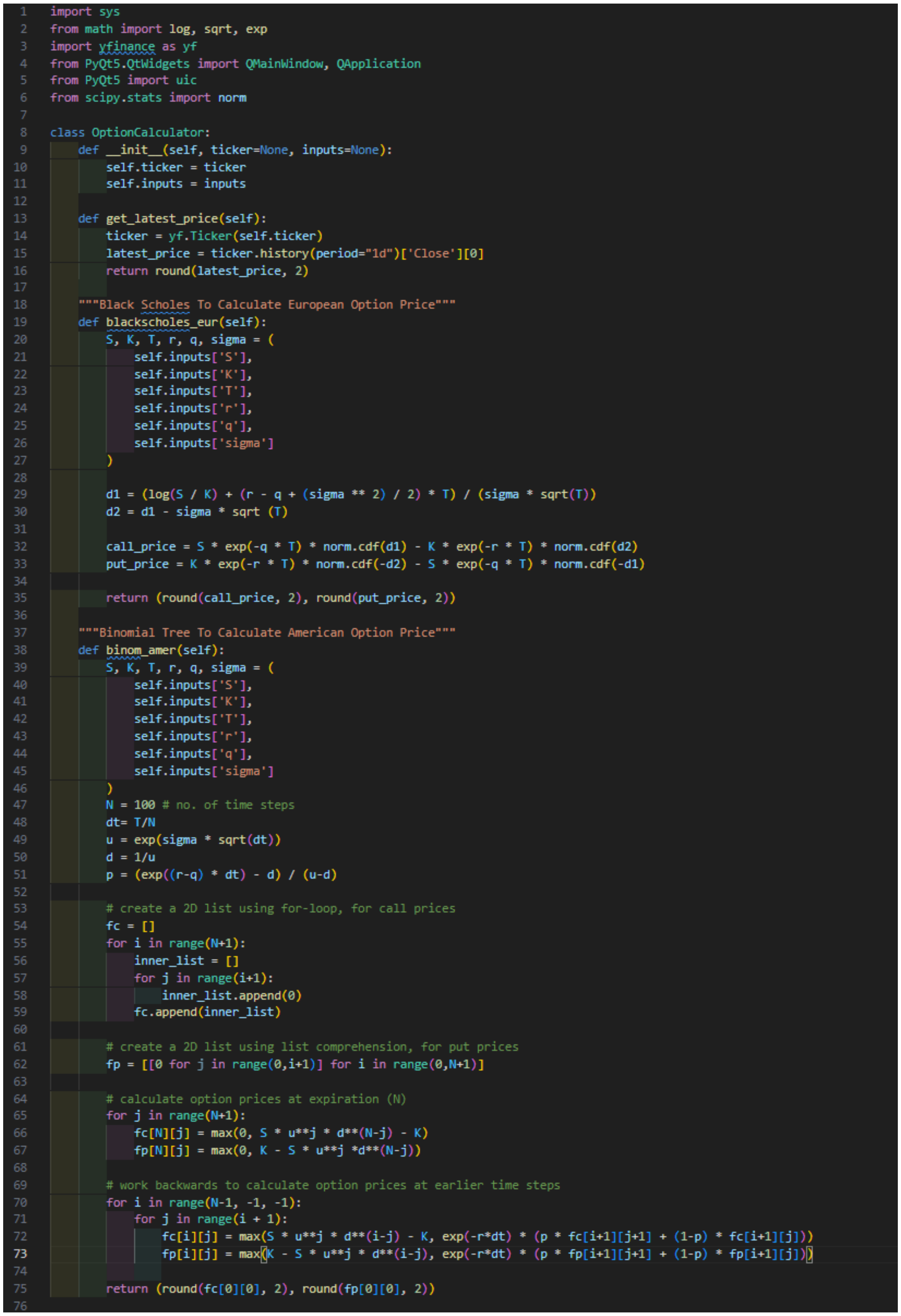
Call Option Price : 9.84

Put Option Price : 2.27

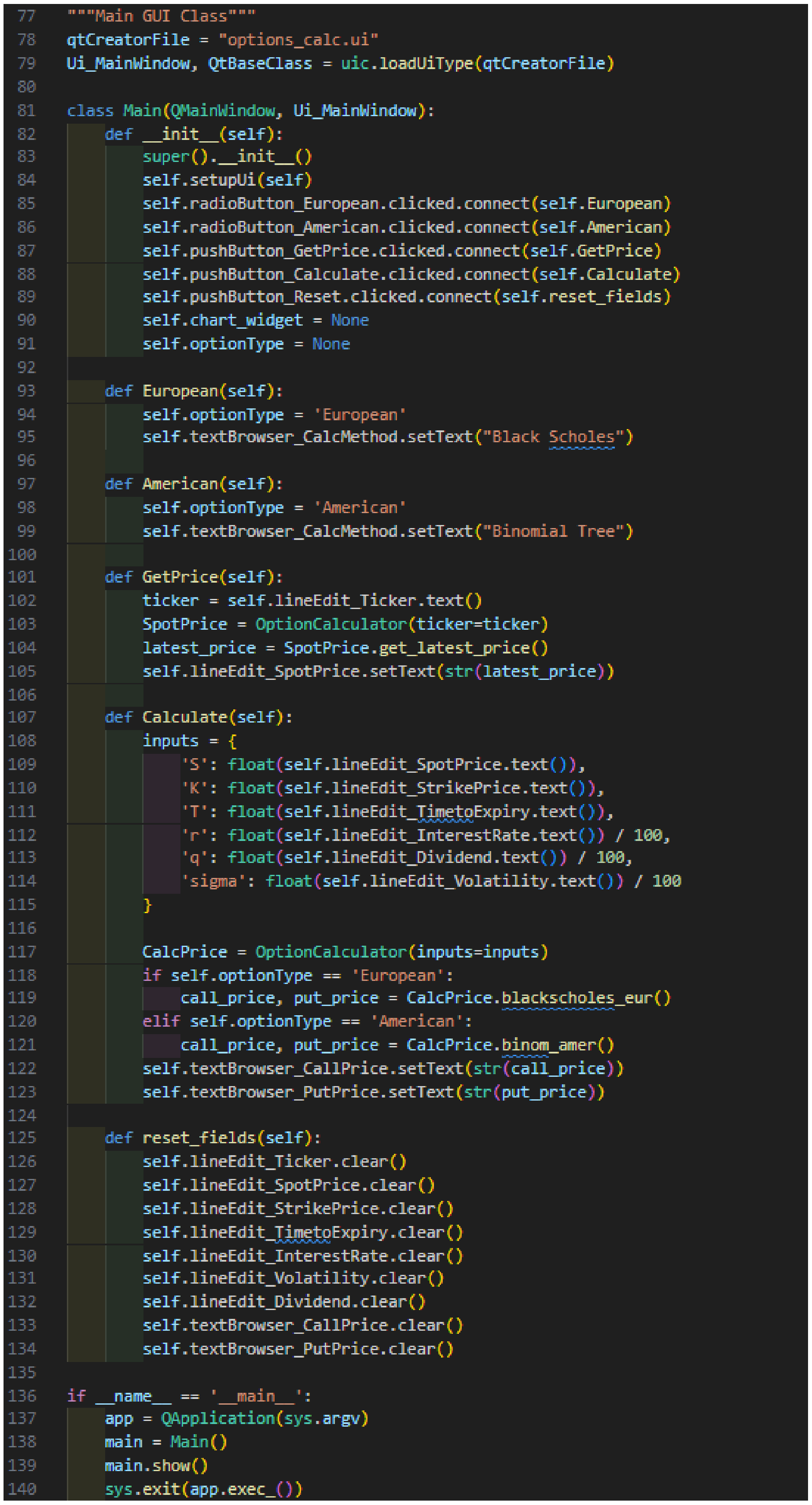
Clear All

Figure 62: Final Options Calculator Output

7. Appendix



```python
import sys
from math import log, sqrt, exp
import yfinance as yf
from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5 import uic
from scipy.stats import norm

class OptionCalculator:
    def __init__(self, ticker=None, inputs=None):
        self.ticker = ticker
        self.inputs = inputs

    def get_latest_price(self):
        ticker = yf.Ticker(self.ticker)
        latest_price = ticker.history(period="1d")['Close'][0]
        return round(latest_price, 2)

    """Black Scholes To Calculate European Option Price"""
    def blackscholes_eur(self):
        S, K, T, r, q, sigma = (
            self.inputs['S'],
            self.inputs['K'],
            self.inputs['T'],
            self.inputs['r'],
            self.inputs['q'],
            self.inputs['sigma']
        )

        d1 = (log(S / K) + (r - q + (sigma ** 2) / 2) * T) / (sigma * sqrt(T))
        d2 = d1 - sigma * sqrt (T)

        call_price = S * exp(-q * T) * norm.cdf(d1) - K * exp(-r * T) * norm.cdf(d2)
        put_price = K * exp(-r * T) * norm.cdf(-d2) - S * exp(-q * T) * norm.cdf(-d1)

        return (round(call_price, 2), round(put_price, 2))

    """Binomial Tree To Calculate American Option Price"""
    def binom_amer(self):
        S, K, T, r, q, sigma = (
            self.inputs['S'],
            self.inputs['K'],
            self.inputs['T'],
            self.inputs['r'],
            self.inputs['q'],
            self.inputs['sigma']
        )
        N = 100 # no. of time steps
        dt= T/N
        u = exp(sigma * sqrt(dt))
        d = 1/u
        p = (exp((r-q) * dt) - d) / (u-d)

        # create a 2D list using for-loop, for call prices
        fc = []
        for i in range(N+1):
            inner_list = []
            for j in range(i+1):
                inner_list.append(0)
            fc.append(inner_list)

        # create a 2D list using list comprehension, for put prices
        fp = [[0 for j in range(0,i+1)] for i in range(0,N+1)]

        # calculate option prices at expiration (N)
        for j in range(N+1):
            fc[N][j] = max(0, S * u**j * d**(N-j) - K)
            fp[N][j] = max(0, K - S * u**j *d**(N-j))

        # work backwards to calculate option prices at earlier time steps
        for i in range(N-1, -1, -1):
            for j in range(i + 1):
                fc[i][j] = max(S * u**j * d**(i-j) - K, exp(-r*dt) * (p * fc[i+1][j+1] + (1-p) * fc[i+1][j]))
                fp[i][j] = max(K - S * u**j * d**(i-j), exp(-r*dt) * (p * fp[i+1][j+1] + (1-p) * fp[i+1][j]))

        return (round(fc[0][0], 2), round(fp[0][0], 2))
```



```python
"""Main GUI Class"""
qtCreatorFile = "options_calc.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)

class Main(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.radioButton_European.clicked.connect(self.European)
        self.radioButton_American.clicked.connect(self.American)
        self.pushButton_GetPrice.clicked.connect(self.GetPrice)
        self.pushButton_Calculate.clicked.connect(self.Calculate)
        self.pushButton_Reset.clicked.connect(self.reset_fields)
        self.chart_widget = None
        self.optionType = None

    def European(self):
        self.optionType = 'European'
        self.textBrowser_CalcMethod.setText("Black Scholes")

    def American(self):
        self.optionType = 'American'
        self.textBrowser_CalcMethod.setText("Binomial Tree")

    def GetPrice(self):
        ticker = self.lineEdit_Ticker.text()
        SpotPrice = OptionCalculator(ticker=ticker)
        latest_price = SpotPrice.get_latest_price()
        self.lineEdit_SpotPrice.setText(str(latest_price))

    def Calculate(self):
        inputs = {
            'S': float(self.lineEdit_SpotPrice.text()),
            'K': float(self.lineEdit_StrikePrice.text()),
            'T': float(self.lineEdit_TimetoExpiry.text()),
            'r': float(self.lineEdit_InterestRate.text()) / 100,
            'q': float(self.lineEdit_Dividend.text()) / 100,
            'sigma': float(self.lineEdit_Volatility.text()) / 100
        }

        CalcPrice = OptionCalculator(inputs=inputs)
        if self.optionType == 'European':
            call_price, put_price = CalcPrice.blackscholes_eur()
        elif self.optionType == 'American':
            call_price, put_price = CalcPrice.binom_amer()
        self.textBrowser_CallPrice.setText(str(call_price))
        self.textBrowser_PutPrice.setText(str(put_price))

    def reset_fields(self):
        self.lineEdit_Ticker.clear()
        self.lineEdit_SpotPrice.clear()
        self.lineEdit_StrikePrice.clear()
        self.lineEdit_TimetoExpiry.clear()
        self.lineEdit_InterestRate.clear()
        self.lineEdit_Volatility.clear()
        self.lineEdit_Dividend.clear()
        self.textBrowser_CallPrice.clear()
        self.textBrowser_PutPrice.clear()

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