

TTP Table of Contents

Sr.No.	Content Name	Course details	Page No.
1	TTP_Foundation 1	Variables, Primitive Data Types, print() method, Expressions, Libraries & Packages	1
2	Assignments 1		7
3	TTP_Foundation 2	Variables, Data Types, print() method	11
4	Assignments 2		24
5	TTP_Foundation 3	Data Structures – Lists, Tuples, Sets, Dictionaries	28
6	Assignments 3		46
7	TTP_Foundation 4	Control Flow & Conditional Statements. (if, elif, else)	50
8	Assignments 4		72
9	TTP_Foundation 5	Loops (for, while) , In-built functions, Lambda functions	76
10	Assignments 5		89
11	TTP_Foundation 6	NumPy Arrays , Indexing, Slicing, Broadcasting	93
12	Assignments 6		105
13	TTP_Foundation 7	NumPy Functions , Matplotlib Plotting	109
14	Assignments 7		121
15	Assignments 8		126
16	TTP_Foundation 8	Matplotlib Plotting	130
17	Assignments 9		137
18	TTP_Foundation 9	Pandas Series & DataFrame, Market Data Download	146

19	Assignments 10		154
20	TPP_Foundation 10	Pandas DataFrame()	159
21	Assignments 11		171
22	TPP_Foundation 11	Pandas Statistical Functions	179
23	Assignments 12		196
24	TPP_Foundation 12	Pandas Indexing, Slicing, Groupby operations	202
25	Assignments 14		221
26	TPP_Foundation 13	Pandas groupby()	229
27	Assignments 13		238
28	Basic Statistics for Technical Analysts	Mean, Median, Variance, Standard Deviation, Returns & Volatility, Correlation	246
29	Evaluating Performance Metrics for Trading Strategies	Cumulative returns, CAGR, Sharpe Ratio, Drawdown, Win/Loss ratio	264
30	TPP_Statistics 1		275
31	TPP_MACross_Metrics		288
32	TPP_Foundation 14	Quantitative Trading Strategy Workflow MA, EMA :- Strategy structure, Moving Average crossover, EMA logic, Signal generation (+1 / -1)	293
33	TPP_Foundation 15	Mean reversion logic, Buy & Hold benchmark, Strategy comparison	305
34	Portfolio & Risk Management	Position sizing, Risk per trade, Portfolio diversification, Drawdown control	315
35	TPP_Foundation 16	Portfolio & Risk Management	333
36	Assignments 15_Mini_Project		341

Basic Keyboard Shortcuts in Jupyter Notebook

A Jupyter Notebook, at its core, is a collection of cells. The cell is simply the 'box' that we write in. We work with **two** types of cells: `Code` and `Markdown`.

- We use the `Code` cell to write in a programming language like Python.
- We use the `Markdown` cell to write in a human language like English. *Markdown* is a computer language in itself.

A Jupyter Notebook operates in two modes :-

Edit Mode

Here, you can type into any cell. You can enter this mode by pressing the *Enter* key or double-clicking the mouse when you hover over the cell.

- Run a cell and go to the next cell in Edit Mode: *Shift + Enter*
- Run a cell and stay in the same cell in Edit Mode: *Ctrl + Enter*

Command Mode

You can enter into it by pressing the *Esc* key. In Command mode, you can carry out commonly performed notebook and cell actions efficiently.

- Navigation: Up and Down arrow keys to view different cells in the Notebook.
- Create a cell above the current one: *a*
- Create a cell below the current one: *b*
- Delete a cell: *d* (press twice)
- Change cell type to Markdown: *m*
- Interrupt the kernel: *i* (press twice)
- Restart the kernel: *0* (press twice)
- Click *h* to open Keyboard shortcut menu help

The `print()` method

Using the `print()` function gives us more control over the appearance of the output. Here are a few different ways in which you can use `print()`.

```
In [45]: # The print() function displays text or variables on the screen.  
print("Hello, World!")
```

Hello, World!

```
In [46]: # You can print multiple items separated by commas  
name = "Sam"  
age = 44  
print("My name is", name, "and I am", age, "years old.")
```

My name is Sam and I am 44 years old.

```
In [47]: # sep defines how multiple items are separated
print("Apple","Banana","Cherry",sep=" | ")

# end defines what happens at the end of the line (default = '\n')
print("This will stay on", end=" ")
print("the same line!")
```

Apple | Banana | Cherry
This will stay on the same line!

```
In [48]: print("This will stay on")
print("the same line!")
```

This will stay on
the same line!

```
In [49]: # end defines what happens at the end of the line (default = '\n')
print("This will be")
print("on different line")
print("Another print")
```

This will be
on different line
Another print

```
In [50]: # You can print expressions directly
x = 5
y = 3
print("Sum =", x + y)
```

Sum = 8

```
In [51]: # f-strings allow easy formatting inside strings
name = "My class"
students = 50
print(f"{name} has {students} active learners.")
```

My class has 50 active learners.

```
In [52]: # Triple quotes let you print multiple lines easily, just have to use print
print("""Line 1
Line 2
Line 3
Line 4""")
```

Line 1
Line 2
Line 3
Line 4

```
In [53]: # Use \n for newline, \t for tab
print("First Line\nSecond Line")
```

First Line
Second Line

```
In [54]: print("Item\tPrice")
print("Apple\t₹100")
```

Item	Price
Apple	₹100

```
In [55]: ticker = 'JUSTDIAL'
firm_name = 'Just Dial Ltd'
industry = 'Technology'
last_traded_price = 754.80
date_of_price_recorded = '6th Sep 2025'
market_cap = 64.19 * (10 ** 9) # The value is originally in billions, hence
```

```
In [56]: outstanding_shares = market_cap / last_traded_price
```

```
In [57]: print("The market capitalization of", firm_name, 'at the close of', date_of_
```

The market capitalization of Just Dial Ltd at the close of 6th Sep 2025 is
Rs. 64190000000.0

Now we will use f-strings or formatted string literals.

In an f-string, you should put all variables inside {} and avoid commas or quotes mixing mid-string.

```
In [58]: print(f"The market capitalization of {ticker} at the close of {date_of_price}
```

The market capitalization of JUSTDIAL at the close of 6th Sep 2025 is Rs.
64190000000.0

```
In [59]: ticker = 'INFY'
firm_name = 'Infosys Ltd'
industry = 'IT'
last_traded_price = 1466.50
date_of_price_recorded = '6th Sep 2025'
market_cap = 6.08 * (10 ** 12) # Value in rupees (6.08 trillion)
```

```
In [60]: print(f"The market capitalization of {ticker} at the close of {date_of_price}
```

The market capitalization of INFY at the close of 6th Sep 2025 is Rs. 6080
000000000.0

In the `.format()` method, the numbers inside the curly braces `{ }` — like `{0}, {1}, {2}, {3}` — are positional placeholders that refer to the order of the arguments inside `.format()`.

```
In [61]: # Example output with .format() method
print("{0} ({1}) has a market capitalization of ₹{2:,.0f} as on {3}."
      .format(firm_name, ticker, market_cap, date_of_price_recorded))
```

Infosys Ltd (INFY) has a market capitalization of ₹6,080,000,000,000 as on 6th Sep 2025.

The `:s` format specifier explicitly tells Python: treat this value as a string.

But when `.format()` sees that the argument is already a string, it automatically uses string formatting by default — so `:s` is optional.

```
In [62]: # Example output with .format() method
print("{0} ({1}) has a market capitalization of ₹{2:,.0f} as on {3}."
      .format(firm_name, ticker, market_cap, date_of_price_recorded))
```

Infosys Ltd (INFY) has a market capitalization of ₹6,080,000,000,000 as on 6th Sep 2025.

```
In [63]: print("The market capitalization of {0:s} ({1:s}) on {3:s} is ₹{2:,.2f} billion"
      .format(ticker, firm_name, market_cap / 10 ** 9, date_of_price_recorded))
```

The market capitalization of INFY (Infosys Ltd) on 6th Sep 2025 is ₹6,080.00 billion.

It is used inside a `.format()` string, and it controls how a number is displayed — especially its formatting, commas, and decimal places.

{2:,.0f}

Part	Meaning
2	Refers to the third argument in <code>.format()</code> (since counting starts from 0).
:	Introduces the format specification .
,.0f	The actual formatting rule (explained below).
Component	Meaning
-----	-----
,	Add thousand separators (commas)
.0	Show 0 digits after the decimal point
f	Keeps it as a numeric value, not scientific notation

Expression	Meaning	Result
<code>10 ** 12</code>	10 raised to 12 (convert trillion to Rs.)	1,000,000,000,000
<code>x / (10 ** 9)</code>	divide x by 1 billion	converts to billions

	Expression	Meaning	Result
In [64]:	2 > 3		
Out[64]:	False		
In [65]:	x = 40		
In [66]:	print(type(40))		<class 'int'>
In [67]:	x = 50.50 print(type(x))		<class 'float'>
In [68]:	10 / 3		
Out[68]:	3.333333333333335		
In [69]:	10 // 3		
Out[69]:	3		
In [70]:	10 % 3		
Out[70]:	1		
In [71]:	20 // 6		
Out[71]:	3		
In [3]:	20 / 6		
Out[3]:	3.333333333333335		
In []:			
In []:			
In []:			

TTP Assignment 1

Question #1

Print the statement - Welcome to Trading Transformation Program, 2025

```
In [1]: print("Welcome to Trading Transformation Program, 2025")
```

```
Welcome to Trading Transformation Program, 2025
```

Question #2

Create a variable x, assign the value 100 to it. Then check its data type and print the value.

```
In [85]: x=100  
print (x)  
print(type (x))
```

```
100  
<class 'int'>
```

Question #3

create 3 variables name, age, city - assign your own details

print the statement calling the above variables

My name is ___ and i am ___ years old. I live in ___

```
In [3]: name = "Ramjilal"  
age = 47  
live = "Vasai"  
print("My name is", name, "and I am", age, "years old", "I live in", live)
```

```
My name is Ramjilal and I am 47 years old I live in Vasai
```

Question #4

Create two variables x and y with values 500 and 300 respectively.

Print their sum using a print statement that displays both text and the computed result together.

```
In [4]: x=500  
y=300  
  
print("sum of", x , "and", y, "is", x+y)
```

```
sum of 500 and 300 is 800
```

Question #5

create 2 variables k and f with values 200.15 and 999.50 respectively

print their ID's

In [12]:

```
k=200.15
f=999.50
print ("k", id(k))
print ("f", id(f))
```

```
k 2387420816560
f 2387420816368
```

Question #6

You are given monthly sales figures as follows: jan_sales = 1200 feb_sales = 950 march_sales = 1100 april_sales = 875

Create a variable named **total_sales** that adds up all four months' sales.

Then, print the statement:

Total Sales = _____

In [90]:

```
jan_sales=1200
feb_sales = 950
march_sales = 1100
april_sales = 875
total_sales= (jan_sales+ feb_sales+ march_sales+ april_sales)
print ("Total Sales:",total_sales)
```

```
Total Sales: 4125
```

Question #7

Using aliases, import the libraries **pandas** and **numpy**. Then, print their version numbers.

 Hint: Use the standard aliases `pd` for pandas and `np` for numpy.

Example Output:

```
pandas version : 2.2.2
numpy version : 1.26.4
```

In [17]:

```
import pandas as pd
import numpy as np
print("pandas version :", pd.__version__)
print("numpy version :" , np.__version__)
```

```
pandas version : 2.2.2
numpy version : 1.26.4
```

Question #8

Create two variables `x = 100` and `y = 500`. Use **logical operators** to check whether `x` is greater than `y` or smaller than `y`. Print the result accordingly.

 Hint: Use comparison operators (`>`, `<`) or direct print expressions.

In [69]:

```
x=100
y=500
print (x>y)
```

False

Question #9

Create variables for a small business scenario as follows: `product_price = 750` `quantity_sold = 5` `total_expenses = 1800`

- 1 Calculate **total_sales** by multiplying `product_price` with `quantity_sold`
- 2 Create a variable **profit** that stores the difference between `total_sales` and `total_expenses`.
- 3 Print both **total_sales** and **profit** with clear labels.

 Hint: Use multiple print statements

In [80]:

```
product_price = 750
quantity_sold = 5
total_expenses = 1800
print ("total_sales", product_price* quantity_sold)
print ("profit", product_price* quantity_sold - total_expenses)
print ("total_sales", product_price* quantity_sold , "profit", product_price*
```

total_sales 3750
profit 1950
total_sales 3750 profit 1950

Question #10

Execute the following two print statements and observe the difference in their results:

```
print(10 + 5 * 2)
print((10 + 5) * 2)
```

In [81]:

```
print(10 + 5 * 2)
```

20

In [83]:

```
print((10 + 5) * 2)
```

30

TTP_Foundation2

What is Programming?

Programming means giving instructions to a computer or any device so that it can perform a task. These tasks can be simple - like showing today's date - or complex - like collecting and analyzing weather data. Programming is also called coding or software development.

📦 Primitive Data Types in Python

Primitive data types are the **basic built-in types** that store simple, single values — not collections or complex objects.

Main Primitive Data Types:

Data Type	Example	Description
int	x = 10	Stores whole numbers (positive, negative, or zero).
float	x = 3.14	Stores numbers with decimal points (real numbers).
bool	x = True	Stores Boolean values — True or False .
str	x = "Hello"	Stores text or a sequence of characters.

How would you make Python print "Welcome to TTP" ?

In [74]: `print("Welcome to TTP")`

Welcome to TTP

Python can do normal math like a calculator:

In [75]: `2 + 3`

Out[75]: 5

In [76]: `10 - 4`

Out[76]: 6

In [77]: `5 * 8`

Out[77]: 40

In [78]: `20 / 3`

Out[78]: 6.666666666666667

```
In [79]: 20 // 3 #floor division doesnt show remainder
```

```
Out[79]: 6
```

```
In [80]: 20 % 3    # modulus only shows remainder
```

```
Out[80]: 2
```

```
In [81]: 2 ** 3      # exponent (2 raised to 3)
```

```
Out[81]: 8
```

```
In [82]: (5 + 3) * 2    # parentheses change priority
```

```
Out[82]: 16
```

A variable in Python is a name that stores a value in memory - you can think of it as a labeled box that holds data for your program to use later.

One thing to keep in mind, the equal '=' sign used while assigning a value to a variable. It should not be read as 'equal to'. It should be read or interpreted as "is set to".

```
In [83]: x = 9
print(x == 10) #checks if x equals 10 --> Output is True or False
```

```
False
```

```
In [84]: x = 100
y = 45
z = x - y
print(z)
```

```
55
```

Built-in math functions

```
In [85]: abs(-5)
```

```
Out[85]: 5
```

```
In [86]: abs(5)
```

```
Out[86]: 5
```

```
In [87]: round(3.456789, 3)
```

```
Out[87]: 3.457
```

```
In [88]: pow(2, 5) #2 to the power 5 = 2×2×2×2×2=32
```

```
Out[88]: 32
```

```
In [89]: 2 ** 5 # alternative
```

```
Out[89]: 32
```

```
In [90]: max(10, 20, 30)
```

```
Out[90]: 30
```

```
In [91]: min(10, 20, 30)
```

```
Out[91]: 10
```

```
In [92]: name = "SAM"  
age = 25  
score = 98.5
```

```
In [93]: type(name)
```

```
Out[93]: str
```

```
In [94]: type(age)
```

```
Out[94]: int
```

```
In [95]: type(score)
```

```
Out[95]: float
```

```
In [96]: print(name, age, score)
```

```
SAM 25 98.5
```

ID of an object

The keyword id () specifies the object's address in memory.

```
In [97]: x = 200
```

```
In [98]: id(x)
```

```
Out[98]: 140719538813960
```

```
In [99]: id(name)
```

```
Out[99]: 1993622614064
```

Data Type of an Object

Every piece of data in Python has a type - it tells Python what kind of value it is and what you can do with it. For example, numbers can be added or multiplied, while text (called a string) can be joined together. The data type decides which operations are allowed and what kind of values can be stored. You can use the type() function to check the data type of any value or variable.

```
In [100]: print(type(name))
print(type(age))
print(type(score))

<class 'str'>
<class 'int'>
<class 'float'>
```

```
In [101]: print(x)
```

```
200
```

```
In [102]: x = x + 200.15
print(x)      # This will print the new value of 'x' variable
type(x)       # This will print the most updated data type of 'x'
```

```
400.15
```

```
Out[102]: float
```

```
In [103]: id(x)
```

```
Out[103]: 1993764235760
```

Note this is different from the 'int x' ID. Python automatically takes care of the physical representation of different data types i.e. an integer value will be stored in a different memory location than a float or string.

There is no semicolon to indicate an end of the statement and therefore Python interprets the end of the line as the end of the statement

```
In [104]: monday_sales = 1200
tuesday_sales = 950
wednesday_sales = 1100
thursday_sales = 875

total_sales = monday_sales + tuesday_sales + wednesday_sales + thursday_sales
print("Total Sales:", total_sales)
```

```
Total Sales: 4125
```

If a line of code becomes too long, you can break it into multiple lines to make it easier to read. In such cases, use a backward slash \ as a line continuation character to tell Python that the statement continues on the next line.

```
In [105]: monday_sales = 2500
tuesday_sales = 1800
wednesday_sales = 3200
thursday_sales = 1500
friday_sales = 2750
saturday_sales = 2100
sunday_sales = 1950

total_sales = monday_sales + tuesday_sales + wednesday_sales + \
              thursday_sales + friday_sales + saturday_sales + sunday_sales

print("Total Weekly Sales:", total_sales)
```

Total Weekly Sales: 15800

Indentation

Python requires proper indentation — that means adding spaces at the beginning of lines to show which code belongs together. The number of spaces can vary, but all lines inside the same block must have the same indentation.

Indentation helps Python (and you!) understand which statements are part of the same structure, like inside an if statement or a loop.

```
In [106]: # Python Program to calculate the square of number
number = 8
num_sq = number ** 2
print (num_sq)
```

64

```
In [107]: number = 8
num_sq = number ** 2
print (num_sq)
```

64

```
In [108]: # Both print statements are indented equally, so Python knows they belong to
if 5 > 2:
    print("Five is greater than two!")
    print("This line is also inside the if block.")
```

Five is greater than two!
This line is also inside the if block.

Modules

Any file in Python which has a .py extension can be a module. A module can consist of arbitrary objects, classes, attributes or functions which can be imported by users.

Syntax for installing external modules

```
!pip install numpy
```

Built-in modules (like math, random, datetime)

Python comes with a large set of built-in (standard library) modules — these are included automatically when you install Python, so you don't need to install them using `pip`. You just import them:

```
In [109]: import math  
print(math.sqrt(16))
```

```
4.0
```

Aliasing

When we import a library, its name can sometimes be long to type repeatedly. To make it shorter, we can give it a **nickname**, called an **alias**.

For example, instead of writing `numpy` every time, we can use a shorter name like `np`. This is done using the `as` keyword while importing.

```
import numpy as np
```

Now we can use `np` wherever we want to use NumPy.

Although you can choose any alias, certain short forms are widely used in the Python community. For example:

- `numpy` → `np`
- `pandas` → `pd`
- `matplotlib.pyplot` → `plt`

Following these standard aliases makes your code easier for others to read and understand.

Math module

which consists of mathematical constants and functions like `math.pi`, `math.sine`, `math.cosine`, etc.

When working with different modules, Python issues warning in case of the user should be alerted of some condition in the program such as use of obsolete modules. These warinings can be supressed by importing the `warnings` module as below.

```
In [110]: import warnings  
warnings.filterwarnings('ignore')
```

```
In [111]: import math
```

```
In [112]: print(math.pi)
print(math.cos(2)) #returns cosine of an angle, angles must be radians, here i
print(math.sin(1)) #returns sine of an angle, angles must be radians, here i
```

```
3.141592653589793
-0.4161468365471424
0.8414709848078965
```

Python only displays the result of the last expression, not all of them. That's because the Python interpreter automatically prints only the final evaluated expression in a cell or block, unless you explicitly ask it to print the others.

The dir () function

The built-in function called `dir()` is used to find out what functions a module defines. It returns a sorted list of strings.

```
In [114]: #import pandas
#dir(pandas)
```

```
In [115]: #dir(math)
```

```
In [116]: import numpy as np
```

```
In [117]: # Will return the maximum number of the number set
np.max([4, 5, 6, 3, 4, 5, 9, 8, 7, 12])
```

```
Out[117]: 12
```

```
In [118]: np.min([4, 5, 6, 3, 4, 5, 9, 8, 7, 12])
```

```
Out[118]: 3
```

Check the version of the package

```
In [119]: np.__version__
```

```
Out[119]: '1.26.4'
```

```
In [120]: import pandas as pd
pd.__version__
```

```
Out[120]: '2.2.2'
```

```
In [121]: #math.__version__
```

The **version** attribute is not available for built-in (standard library) modules like `math`, `os`, or `sys`, because they're part of Python itself — they don't have a separate version number. They simply use the same version as your Python interpreter.

Definition

An **operator** in Python is a special symbol used to perform operations on variables and values — like addition, comparison, or logical testing.

Basic Operators in Python

Category	Operators	Description	Example	Result
Arithmetic	+ , - , * , / , // , % , **	Perform basic mathematical operations	5 + 2	7
Assignment	= , += , -= , *= , /= , %=	Assign or update values in variables	x = 5; x += 3	8
Comparison	== , != , > , < , >= , <=	Compare two values; returns True or False	5 > 3	True
Logical	and , or , not	Combine or invert Boolean expressions	True and False	False

Python Operator Precedence

Priority	Operator	Description	Example
1	()	Parentheses (grouping)	(2 + 3) * 4 → 20
2	**	Exponent / Power	2 ** 3 → 8
3	* , / , // , %	Multiplication, Division, Floor-division, Modulus	10 / 2 → 5.0
4	+ , -	Addition, Subtraction	5 + 3 → 8
5	< , > , <= , >= , == , !=	Comparison operators	5 > 3 → True
6	not , and , or	Logical operators	a and b , not x
7 (Lowest)	=	Assignment	x = 10

The **assignment operator** (=) has the **lowest precedence** in Python.

This means expressions on the right side are evaluated **first**, and the result is then assigned to the variable on the left.

```
In [122]: x = 10 + 5
          print(x)
```

15

```
In [123]: print(2 + 3 * 4)
          print((2 + 3) * 4) #Parentheses always win.
```

14
20

In [124]: `print(2 * 3 ** 2) # ** (power) has higher precedence than *`

18

In [125]: `print(10 + 2 * 3) #Multiplication happens before addition`

16

In [126]: `print(20 / 5 * 2) # / and * have equal precedence, so they run left to right`

8.0

In [127]: `print(10 + 2 * 3 > 15) #Comparison after Arithmetic`

True

In [128]: `# Both comparisons evaluated first, then logical and`

`x = 5`

`print(x > 3 and x < 10)`

True

In [129]: `# Both comparisons evaluated first, then logical and`

`# The and operator returns True only if both sides are True.`

`x = 5`

`print(x > 3 and x > 10)`

False

The **not** operator

not means “reverse the truth” or “give the opposite result.”

In [130]: `# comparisons evaluated first, then logical`

`x = 5`

`print(not (x > 3))`

False

Expressions

'Expressions' are generally a combination of numbers, variables, and operators.

Core Money-Value Formulas

Future value (FV)

Future Value Formula

$$FV = PV \times (1 + r)^n$$

Where:

- (FV): Future Value
- (PV): Present Value
- (r): Rate of return (per period)
- (n): Number of periods

What would be the FV, if I have 100000 with me now and I will be investing it for 5 year, at an annual return of 10%?

In [131]:

```
PV = 100000
r = 0.10
n = 5

FV = PV * ((1+r) ** n) # Formula for calculating Future Value

print (int(FV))
```

161051

Present value (PV)

Present Value Formula

$$PV = \frac{FV}{(1 + r)^n}$$

Where:

- (PV): Present Value
- (FV): Future Value
- (r): Rate of return (per period)
- (n): Number of periods

What would be the PV, if I have to discount 1050 at a 5% annual rate for a period of 1 year?

Simply say, how much that future ₹1050 is worth today, assuming money loses value over time at 5% per year

In [132]:

```
FV = 1050
r = 0.05
n = 1

PV = FV / ((1 + r) ** n) # Formula for calculating Present Value

print (PV)
```

1000.0

Annuity payments

Annuity Payment Formula (Annuity Due)

$$AP = \frac{FV \times r}{((1 + r)^n - 1) \times (1 + r)}$$

Where:

- (AP): Annuity Payment (periodic saving amount)
- (FV): Future Value (target amount)
- (r): Rate of return (per period)
- (n): Number of periods

What would be the annual periodic saving amount, if you want a lump sum of 100000 at the end of 5 years? The rate of return is 10%? (Assuming the first payment is made at the start of each year)

```
In [133]: r = 0.1
n = 5
PV = 0
FV = 100000

AP = (FV * r) / (((1 + r) ** n - 1)*(1+r)) # Formula for Annuity payments, g
print (int(AP))

14890
```

Compounding

Future Value Formula (with Compounding)

$$FV = PV \times \left(1 + \frac{r}{n}\right)^{n \times t}$$

Where:

- (FV): Future Value
- (PV): Present Value (initial investment)
- (r): Annual interest rate
- (n): Number of compounding periods per year
- (t): Time in years

Assume that the 5% annual interest rate bond makes semiannual payments. That is, for an investment of 1000, you will get 25, after the first 6 months and another 25 after 1 year. The annual rate of interest is 5%. What would be the FV, if I hold the bond for 1 year?

```
In [134]: PV = 100000
r = 0.05
n = 2 # number of periods = 2 since bond makes semiannual payments
t = 1 # number of years

FV = PV * ((1+(r/n)) ** (n*t)) # Formula for compounding

print(int(FV))
```

105062

Checking the Current working directory path

The `os` package in Python lets you interact with your computer's operating system — for tasks like working with files, directories, and environment variables

```
In [135]: import os
```

```
In [138]: #print(os.getcwd())
```

```
In [ ]:
```

TTP Assignment 2

Question #1

Create two variables `student_name = "Rahul"` and `student_age = 20`. Use an **f-string** to print the sentence: **My name is Rahul and I am 20 years old.**

```
In [140]: student_name = "Rahul"
          student_age = 20
          print(f"My name is {student_name} and i am {student_age} years old")
```

My name is Rahul and i am 20 years old

Question #2

Create variables `item = "Tablet"` and `price = 18500`. Use an ****f-string**** to print the message: ****The price of Tablet is ₹18500.**** Then, modify your code so that the price appears with a comma separator (**₹18,500**).

 Hint: Use `{price:,}` inside the f-string to format numbers with commas.

```
In [141]: item = "Tablet"
          price = 18500
          print(f"The price of {item} is {price : ,}")
```

The price of Tablet is 18,500

Question #3

Create a variable `radius = 7`. Calculate the **area of a circle** using the formula `3.14 * radius ** 2`. Print the result with a clear message such as:

Area of Circle = _____

 Hint: Use a single `print()` statement combining text and the computed value.

```
In [142]: radius = 7
          area = 3.14* radius ** 2
          print("Area of a Circle", area)
```

Area of a Circle 153.86

Question #4

Create two variables `fruit = "Mango"` and `quantity = 6`. Use the `.format()` method to print the sentence:

I bought 6 Mangoes from the market.

```
In [144]: fruit = "Mango"
quantity = 6
print("I bought", quantity, fruit, "from the market")
```

I bought 6 Mango from the market

Question #5

Create variables `ticker = "TCS"`, `market_cap = 120000` and `last_traded_price = 3825`. Use the `.format()` method to print the following message:

TCS has a market capitalization of ₹120000 Cr and the last traded price is ₹3825.

```
In [9]: ticker = "TCS"
market_cap = 120000
last_traded_price = 3825
print(f"{ticker} has a market capitalization of ₹{market_cap} Cr and the last
```

TCS has a market capitalization of ₹120000 Cr and the last traded price is ₹3825.

Question #6

Modify the previous program to display the numbers with comma separators.

Example output:

TCS has a market capitalization of ₹1,20,000 Cr and the last traded price is ₹3,825.

 Hint: Use `{:,}` inside `.format()` to format numbers with commas.

```
In [146]: ticker = "TCS"
market_cap = 120000
last_traded_price = 3825
print(f"{ticker} has a market capitalization of ₹{market_cap : ,} Cr and the ]
```

TCS has a market capitalization of ₹ 120,000 Cr and the last traded price is ₹ 3,825.

Question #7

Create four variables:

- a = 25 (integer)
- b = 12.5 (float)
- c = True (boolean)
- d = "Python" (string)

Print each variable's value and check its **data type** using the `type()` function.

```
In [147]: a = 25
b = 12.5
c = True
d = "Python"
print(type (a))
print(type (b))
print(type (c))
print(type (d))
```

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
```

Question #8

Create a program that demonstrates the use of the `sep` and `end` parameters in the `print()` function.

- 1** Print three city names — *Delhi, Mumbai, Chennai* — separated by “ | ” using the `sep` parameter.
- 2** Then, print two words — *Python* and *Programming* — on the **same line** using the `end` parameter.

```
In [16]: print ("Delhi", "Mumbai", "Chennai",sep= " | ")
print ("Python and", end=" ")
print ("Programming")
```

```
Delhi | Mumbai | Chennai
Python and Programming
```

Question #9

Use **triple quotes** (“““ “““) in a print statement to display multiple lines of text.

Print the following exactly as shown:

```
Welcome to Python
This is your second assignment
Keep practicing daily
```

```
In [131]: print("""Welcome to Python
This is your second assignment
Keep practicing daily""")
```

```
Welcome to Python
This is your second assignment
Keep practicing daily
```

Question #10

Write a program that uses **escape characters** to format output.

- 1** Use `\n` to print text on a **new line**.
- 2** Use `\t` to create a **tab space** between words.

Print the following output exactly as shown:

Name Subject

Riya Maths

Aman Science

In [137]: `print("Name Subject\n Riya Maths\n Aman Science")`

Name Subject

Riya Maths

Aman Science

In [138]: `print("Name Subject\t Riya Maths\t Aman Science")`

Name Subject Riya Maths Aman Science

In []:

TTP_Foundation3



Python Data Structures

Data structures are different ways of storing and organizing data so that it can be used efficiently.

Python provides several built-in data structures that make it easy to handle different types of information.

Each structure has its own features, strengths, and use cases - depending on whether you need to store ordered data, unique values, or key-value pairs.

The 4 main data structures in Python are:

List – an ordered, changeable collection of items.

Tuple – an ordered, unchangeable (immutable) collection.

Set – an unordered collection of unique items.

Dictionary – a collection of key–value pairs for quick lookups.

Ordered: Items have a fixed position (index) and always appear in the same sequence when you access or print them.

Mutable: You can change, add, or remove elements after the object is created.

Allow Duplicates: The same value can appear more than once inside the collection.

Lists []

A list in Python is used to store multiple pieces of data in one place. Unlike some programming languages, Python lists can hold different types of values together — numbers, strings, or even other lists.

Lists are mutable, which means you can change, add, or remove elements after creating them — without creating a new list.

Creating lists

List is enclosed by square brackets and elements should be separated by a comma.

```
In [127]: new_list = [] # Empty List  
type(new_list)
```

```
Out[127]: list
```

```
In [128]: new_list = [100, 200, 300, 4000] # A List of integers  
type(new_list)
```

```
Out[128]: list
```

```
In [129]: print(new_list)
```

```
[100, 200, 300, 4000]
```

```
In [130]: new_list = [10, 20.2, "thirty", 40] # A List of mixed data types  
type(new_list)
```

```
Out[130]: list
```

```
In [131]: print(new_list)
```

```
[10, 20.2, 'thirty', 40]
```

```
In [132]: new_list = [[10, 20, 30], [10.1, 20.2, 30.3], ["ten", "twenty", "thirty"]]  
# 3 Lists nested inside another list  
  
type(new_list)
```

```
Out[132]: list
```

```
In [133]: print(new_list)
```

```
[[10, 20, 30], [10.1, 20.2, 30.3], ['ten', 'twenty', 'thirty']]
```

Accessing The Lists Elements

```
In [134]: new_list1 = [100, 200, 300, 4000]
```

```
In [135]: new_list1[0]
```

```
Out[135]: 100
```

```
In [136]: new_list1[-1]
```

```
Out[136]: 4000
```

```
In [137]: new_list1[0:3]
```

```
Out[137]: [100, 200, 300]
```

```
In [138]: print(new_list)
```

```
[[10, 20, 30], [10.1, 20.2, 30.3], ['ten', 'twenty', 'thirty']]
```

```
In [139]: new_list[0]
```

```
Out[139]: [10, 20, 30]
```

```
In [140]: new_list[1]
```

```
Out[140]: [10.1, 20.2, 30.3]
```

```
In [141]: new_list[0][2]
```

```
Out[141]: 30
```

```
In [142]: new_list[2][2]
```

```
Out[142]: 'thirty'
```

```
In [143]: new_list[2][0]
```

```
Out[143]: 'ten'
```

Methods for list manipulation

```
In [144]: my_list = [100, 200, 300, 400, 500]
```

```
In [145]: print(my_list)
```

```
[100, 200, 300, 400, 500]
```

list.append (x)

Add an item to the end of the list.

```
In [146]: my_list.append(50)
```

```
print(my_list)
```

```
[100, 200, 300, 400, 500, 50]
```

list.extend (x)

Extend the list by appending all the items at the end of the list.

```
In [147]: my_list.extend([60, 70, 80, 90])
```

```
print(my_list)
```

```
[100, 200, 300, 400, 500, 50, 60, 70, 80, 90]
```

```
In [148]: my_list.extend([60])
```

```
In [149]: my_list
```

```
Out[149]: [100, 200, 300, 400, 500, 50, 60, 70, 80, 90, 60]
```

list.insert (i,x)

Insert an item at any given position within the list. The first argument 'i', is the index of the item before which you want to insert something. To insert something at the beginning of the list, you may type list.insert (0,x)

```
In [150]: # Inserting an item in the beginning  
my_list.insert(0, 0)
```

```
print(my_list)
```

```
[0, 100, 200, 300, 400, 500, 50, 60, 70, 80, 90, 60]
```

```
In [151]: my_list.insert(1, 5000)
```

```
print(my_list)
```

```
[0, 5000, 100, 200, 300, 400, 500, 50, 60, 70, 80, 90, 60]
```

```
In [152]: # Inserting an item at the end
```

```
my_list.insert(11, 1000)
```

```
print(my_list)
```

```
[0, 5000, 100, 200, 300, 400, 500, 50, 60, 70, 80, 1000, 90, 60]
```

```
In [153]: # Inserting an item at the 6th position in a list
```

```
my_list.insert(6, 55)
```

```
print(my_list)
```

```
[0, 5000, 100, 200, 300, 400, 55, 500, 50, 60, 70, 80, 1000, 90, 60]
```

list.remove (x)

Remove the first item from the list whose value is 'x'. It is an error if there is no such item.

Use .remove() when you know the value

```
In [154]: my_list.remove(100)
```

```
print(my_list)
```

```
[0, 5000, 200, 300, 400, 55, 500, 50, 60, 70, 80, 1000, 90, 60]
```

```
In [155]: fruits = ["apple", "banana", "apple", "cherry", "apple"]
```

```
print(fruits)
```

```
fruits.remove("apple")
```

```
print(fruits)
```

```
['apple', 'banana', 'apple', 'cherry', 'apple']
```

```
['banana', 'apple', 'cherry', 'apple']
```

If you want to remove all duplicates of "apple":

We can use a simple Loop:

```
In [156]: new_fruits = []
for f in fruits:
    if f != "apple":
        new_fruits.append(f)
print(new_fruits)
```

```
['banana', 'cherry']
```

`list.pop (i)`

Remove any item from any given position (index) in the list.

Use `.pop()` when you know the index or need (this removes it permanently)

```
In [157]: print(my_list)
```

```
[0, 5000, 200, 300, 400, 55, 500, 50, 60, 70, 80, 1000, 90, 60]
```

```
In [158]: my_list[5]
```

```
Out[158]: 55
```

```
In [159]: # Removes and returns the '5th' element from the List
my_list.pop(5)
```

```
Out[159]: 55
```

```
In [160]: print(my_list)
```

```
[0, 5000, 200, 300, 400, 500, 50, 60, 70, 80, 1000, 90, 60]
```

```
In [161]: # Removes and returns the Last element from the List
my_list.pop()
```

```
Out[161]: 60
```

```
In [162]: print(my_list)
```

```
[0, 5000, 200, 300, 400, 500, 50, 60, 70, 80, 1000, 90]
```

`list.index (x)`

It returns a zero-based index in the list of the first item whose value is x. Raises an error if there is no such item as 'x'.

```
In [163]: my_list.index(50)
```

```
Out[163]: 6
```

```
In [164]: my_list.index(0)
```

```
Out[164]: 0
```

list.count (x)

Returns the number of times 'x' appears in the list

```
In [165]: new_list = [100, 100, 100, 20, 30, 40, 50]
print(new_list)
new_list.count(100)
```

```
[100, 100, 100, 20, 30, 40, 50]
```

Out[165]: 3

list.reverse ()

It reverses the items of the list.

```
In [166]: print(my_list)
```

```
[0, 5000, 200, 300, 400, 500, 50, 60, 70, 80, 1000, 90]
```

```
In [167]: my_list.reverse()
```

```
print(my_list)
```

```
[90, 1000, 80, 70, 60, 50, 500, 400, 300, 200, 5000, 0]
```

list.sort ()

It sorts the items in the list in ascending order by default

```
In [168]: my_list.sort()
```

```
In [169]: print(my_list)
```

```
[0, 50, 60, 70, 80, 90, 200, 300, 400, 500, 1000, 5000]
```

```
In [170]: #For descending order
my_list.sort(reverse=True)
print(my_list)
```

```
[5000, 1000, 500, 400, 300, 200, 90, 80, 70, 60, 50, 0]
```

Dictionaries { }

A dictionary in Python is used to store data in pairs — called key-value pairs. Each key is unique and acts like a label that helps you quickly find its corresponding value. Dictionaries are unordered, changeable (mutable), and written using curly braces {}

A key in a dictionary is the unique identifier used to access a specific value

```
In [171]: new_dict = {} # Empty Dictionary
          type(new_dict)
```

Out[171]: dict

```
In [172]: student = {"name": "Amit", "age": 20, "course": "Python" }

          print(student["name"])      # Access value using key
```

Amit

```
In [173]: print(student)

{'name': 'Amit', 'age': 20, 'course': 'Python'}
```

```
In [174]: student["age"] = 21      # Update a value
          print(student)

{'name': 'Amit', 'age': 21, 'course': 'Python'}
```

```
In [175]: student["grade"] = "A"    # Add a new key-value pair
```

```
In [176]: print(student)

{'name': 'Amit', 'age': 21, 'course': 'Python', 'grade': 'A'}
```

Dictionary manipulations

Let us have a look at the few functions for accessing or manipulating dictionaries.

`len (x_dict)`

To know the number of key: value pairs in the dictionary.

```
In [177]: len(student)
```

Out[177]: 4

`x_dict.keys ()`

Returns all the 'keys' of dictionaries

```
In [178]: student.keys()
```

Out[178]: dict_keys(['name', 'age', 'course', 'grade'])

`x_dict.values ()`

Returns all the 'values' of dictionaries

In [179]: `student.values()`

Out[179]: `dict_values(['Amit', 21, 'Python', 'A'])`

The **del** statement

It is used for deleting any keys from the dictionary.

In [180]: `del student['age']`

`print(student)`

`{'name': 'Amit', 'course': 'Python', 'grade': 'A'}`

`x_dict.pop(key)`

It will delete the 'value' corresponding to the required key.

In [181]: `student.pop('grade')`

Out[181]: `'A'`

In [182]: `print(student)`

`{'name': 'Amit', 'course': 'Python'}`

`sorted(x_dict)`

The dictionary will get sorted by its values.

In [183]: `student_scores = {"Rahul": 85, "Amit": 92, "Neha": 78, "Sneha": 88}`

```
# Sort dictionary by keys (names)
sorted_names = sorted(student_scores)

print(sorted_names)
```

`['Amit', 'Neha', 'Rahul', 'Sneha']`

If you want to sort by the values, you can do this:

In [184]: `sorted_by_value = sorted(student_scores, key=student_scores.get)`
`print(sorted_by_value) #Keys sorted by Values`

`['Neha', 'Rahul', 'Sneha', 'Amit']`

In [185]: `# Sort by values (ascending) and print all key + value pairs`

```
for name in sorted(student_scores, key=student_scores.get):
    print(name, ":", student_scores[name])
```

`Neha : 78`

`Rahul : 85`

`Sneha : 88`

`Amit : 92`

```
In [186]: # Sort by values (ascending) and print all key + value pairs
for name in sorted(student_scores, key=student_scores.get, reverse=True):
    print(name, ":", student_scores[name])
```

Amit : 92
 Sneha : 88
 Rahul : 85
 Neha : 78

x_dict.clear()
 Clears all the content of the dictionary

```
In [187]: student_scores.clear()
print(new_dict)
{}
```

Tuples ()

Tuple is an immutable list. Similar to lists a tuple can contain a heterogeneous sequence of elements but it is not possible to append, edit or remove any individual elements within a tuple.

How to create tuples

Tuples are enclosed in parenthesis and the items within them are separated by commas.

There are no methods supported by Tuples that can help us manipulate a tuple once created.

```
In [188]: my_tup = () #this is an empty tuple
type(my_tup)
```

Out[188]: tuple

```
In [189]: my_tup = (5, 10, 300, 400) # A tuple of integers
print(my_tup)
```

(5, 10, 300, 400)

```
In [190]: type(my_tup)
```

Out[190]: tuple

```
In [191]: my_tup = (1000, 75.8, 'gaurav', 400) # A tuple of mixed data type
type(my_tup)
```

Out[191]: tuple

```
In [192]: print(my_tup)
```

(1000, 75.8, 'gaurav', 400)

```
In [193]: my_tup = ((2000, 3000, 400), (60.1, 70.2, 80.3),
              ("fifty", "sixty", "seventy")) # Tuples can be nested
type(my_tup)
```

Out[193]: tuple

```
In [194]: my_tup
```

Out[194]: ((2000, 3000, 400), (60.1, 70.2, 80.3), ('fifty', 'sixty', 'seventy'))

```
In [195]: my_tup = (2500, 3500, 4500, 5500, 6500)
```

```
In [196]: print(my_tup)
```

(2500, 3500, 4500, 5500, 6500)

```
In [197]: my_tup[0]
```

Out[197]: 2500

```
In [198]: my_tup[-1]
```

Out[198]: 6500

Access Tuples by its index position

```
In [199]: t = ('a', 'b', 'c', 'd')
```

```
print(t)
```

('a', 'b', 'c', 'd')

```
In [200]: print(t[2]) # Access by index → 'c'
```

c

```
In [201]: print(t[-1]) # Negative index → 'd'
```

d

Access Tuples Length

```
In [202]: print(len(t)) # Length → 4
```

4

Tuple Slicing

```
In [203]: t = (10, 20, 30, 40, 50)
print(t)
```

(10, 20, 30, 40, 50)

```
In [204]: print(t[0:4]) # element 1 to 3, doesnt includes 4th element here
(10, 20, 30, 40)
```

```
In [205]: print(t[:3]) # all till 3rd element
(10, 20, 30)
```

```
In [206]: print(t[-1])
50
```

```
In [207]: print(t[-3:]) # last 2 elements
(30, 40, 50)
```

```
In [208]: print(t[-3])
30
```

Sets

A set is an unordered collection with no duplicate elements. They are useful to create lists that hold only unique values and are also mutable. The elements of a set can be anything like numbers, strings or characters.

How to create Sets

Curly braces or the set () function can be used to create sets and the items within them are separated by commas.

```
In [209]: my_set = set() # this is an empty set
print(my_set)
type(my_set)

set()
```

```
Out[209]: set
```

```
In [210]: # We can use the set() function to create sets
set_a = set('TEMPERATUREEEEE') #contains 11 words
print(type(set_a))
print(set_a) #only unique values will b

<class 'set'>
{'U', 'E', 'P', 'M', 'R', 'T', 'A'}
```

set() function takes only one argument and not multiple strings

```
In [211]: #my_set = set('Infosys', 'TCS', 'Reliance', 'Auropharma', 'Grasim')
#This will throw a TypeError
```

Alternatively, we can use set() with a list

```
In [212]: my_set = set(['Infosys', 'TCS', 'Reliance', 'Auropharma', 'Grasim'])
```

```
In [213]: print(my_set)
```

```
{'Grasim', 'Reliance', 'TCS', 'Auropharma', 'Infosys'}
```

Simplest, to use only Curly Braces

```
In [214]: my_set2 = {'ABC', 'DFG', 'KJH', 'JHY'}
```

```
In [215]: my_set2
```

```
Out[215]: {'ABC', 'DFG', 'JHY', 'KJH'}
```

◆ Key idea - Sets are not ordered

Sets only care what elements are present, not the order in which they appear.

So, there's no concept of "first" or "last" element in a set.

Only Unique values are printed

```
In [216]: A = {10, 20, 30, 40}
B = {30, 10, 20}
print(A == B)    # Only contents are compared not positions
```

```
False
```

Set operations

Mathematical operations like set union, set intersection, set difference can be performed

```
In [217]: # Lets create 2 sets - set_1 and set_2
```

```
set_1 = set('LMKNOP')
```

```
set_2 = set('OPQRST')
```

```
print(set_1)
```

```
print(set_2)
```

```
{'P', 'K', 'N', 'M', 'O', 'L'}
{'P', 'S', 'R', 'T', 'O', 'Q'}
```

x.union(y)

This method returns all the unique items that are present in the two sets, as a new set.

In [218]: `set_1.union(set_2)`

Out[218]: `{'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T'}`

In [219]: `#Simpler alternate way of performing Union is by using the pipe '/' operator
set_1 | set_2`

Out[219]: `{'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T'}`

x.intersection(y)

This method returns the common items that are present in two sets, as a new set.

In [220]: `set_1.intersection(set_2)`

Out[220]: `{'O', 'P'}`

In [221]: `#Simpler alternate way of performing Intersection is by using ampersand '&' operator
set_1 & set_2`

Out[221]: `{'O', 'P'}`

x.difference(y)

This method returns the items of 'set 1' which are not common to the 'set 2', as a new set.

In [222]: `print(set_1)
print(set_2)`

`{'P', 'K', 'N', 'M', 'O', 'L'}`
`{'P', 'S', 'R', 'T', 'O', 'Q'}`

In [223]: `set_1.difference(set_2)`

Out[223]: `{'K', 'L', 'M', 'N'}`

In [224]: `#Simpler alternate way of performing Difference is by using minus '-' operator
set_1 - set_2`

Out[224]: `{'K', 'L', 'M', 'N'}`

difference_update ()

Removes all elements from the first set that are also present in the second set.

It modifies the first set - no new set is created.

In [225]: `set_1.difference_update(set_2)
print(set_1)
print(set_2)`

`{'K', 'N', 'M', 'L'}`
`{'P', 'S', 'R', 'T', 'O', 'Q'}`

x.isdisjoint(y)

The method checks whether two sets have no elements in common.

Returns:

True → if the sets have no common elements

False → if any element is shared between them

In [226]: `set_1.isdisjoint(set_2)`

Out[226]: True

In [227]: `set_3 = set('ABCD')`
`set_4 = set('CDEF')`
`print(set_3)`
`print(set_4)`

```
{'D', 'B', 'A', 'C'}
{'C', 'D', 'E', 'F'}
```

In [228]: `set_3.isdisjoint(set_4) #Here we have common elements`

Out[228]: False

y.issubset(x)

This method returns True for 'Set 2', if all the elements of 'Set 2' are present in 'Set 1'

Easy way to remember:

Subset → smaller inside bigger

Superset → bigger contains smaller

In [229]: `set_5 = set('ABCDEFG')`
`set_6 = set('EFG')`
`print(set_5)`
`print(set_6)`

```
{'D', 'B', 'E', 'G', 'A', 'C', 'F'}
{'G', 'E', 'F'}
```

In [230]: `set_6.issubset(set_5)`

Out[230]: True

Because set_5 has extra elements (A, B, C, D) that are not present in set_6. So, set_5 is not a subset of set_6.

In [231]: `set_5.issubset(set_6)`

Out[231]: False

x.issuperset(y)

This method returns True for 'Set 1' if all the elements of Set 2 are present in 'Set 1'.

In [232]: `#Lets first check if set_5 is a superset as we know it contains more elements
set_5 > set_6`

Out[232]: True

In [233]: `set_6 > set_5`

Out[233]: False

In [234]: `set_5.issuperset(set_6)`

Out[234]: True

In [235]: `set_6.issuperset(set_5)`

Out[235]: False

x.add(e)

It adds a single item to the set and updates the set.

In [236]: `set_5`

Out[236]: {'A', 'B', 'C', 'D', 'E', 'F', 'G'}

In [237]: `set_5.add('XYZ')`

In [238]: `set_5`

Out[238]: {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'XYZ'}

x.update(e)

It adds multiple items to the set and updates the set.

In [239]: `set_5.update(['X', 'Y', 'Z'])`

In [240]: `set_5`

Out[240]: {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'X', 'XYZ', 'Y', 'Z'}

x.discard(e)

It removes a single item from the set and updates it.

In [241]: `set_5.discard('XYZ')`

In [242]: `set_5`

Out[242]: {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'X', 'Y', 'Z'}

x.pop()

It pops and returns any arbitrary item from the set. The original set permanently loses that element.

```
In [243]: set_5.pop()
```

```
Out[243]: 'Z'
```

```
In [244]: set_5
```

```
Out[244]: {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'X', 'Y'}
```

x.clear()

It clears all the items of the set.

```
In [245]: set_7 = set('THE')
print(set_7)
```

```
{'H', 'T', 'E'}
```

```
In [246]: set_7.clear()
```

```
In [247]: print(set_7) #Output will be an empty set
```

```
set()
```

```
In [ ]:
```

TTP Assignment 3

Question #1

- A. Create a variable List_1 with the below data
102.5, 104.2, 103.8, 105.6, 106.3, 107.1, 106.5, 108.4, 109.2, 110.0
- B. print this list
- C. print the 2nd item which is 104.2
- D. Access the 1st item which is 102.5, do not print it
- E. Print the data type of List_1

```
In [1]: #A. Create a variable List_1 with the below data
List_1 = [102.5, 104.2, 103.8, 105.6, 106.3, 107.1, 106.5, 108.4, 109.2, 110.0]

#B. print this list
print(List_1)

#C. print the 2nd item which is 104.2
print(List_1[1])

#D. Access the 1st item which is 102.5, do not print it
first_item = List_1[0]

#E. Print the data type of List_1
print(type(List_1))
```

[102.5, 104.2, 103.8, 105.6, 106.3, 107.1, 106.5, 108.4, 109.2, 110.0]
104.2
<class 'list'>

Question #2

- A. Create a variable List_2 with the below data
102.5, 45, Apple, 99.9, Banana, 150, 87.3, Grape, 200, Mango
- B. Print the statement - This is my list: _____

```
In [3]: #A. Create a variable List_2 with the below data
List_2 = [102.5, 45, "Apple", 99.9, "Banana", 150, 87.3, "Grape", 200, "Mango"]

#B. Print the statement with List
print("This is my list:", List_2)
```

This is my list: [102.5, 45, 'Apple', 99.9, 'Banana', 150, 87.3, 'Grape', 200, 'Mango']

Question #3

- A. Add 5000 to the end of List_1
- B. print the updated list which is List_1
- C. Add 10000 to the of this updated list
- D. print the updated list

```
In [4]: # Original List_1
List_1 = [102.5, 104.2, 103.8, 105.6, 106.3, 107.1, 106.5, 108.4, 109.2, 110.0, 500]
#A. Add 5000 to the end of List_1
List_1.append(5000)

#B. print the updated list which is List_1
print(List_1)

#C. Add 10000 to the end of this updated list
List_1.append(10000)

#D. print the updated list
print(List_1)
```

[102.5, 104.2, 103.8, 105.6, 106.3, 107.1, 106.5, 108.4, 109.2, 110.0, 500]
[102.5, 104.2, 103.8, 105.6, 106.3, 107.1, 106.5, 108.4, 109.2, 110.0, 500, 10000]

Question #4

- A. Add 7000, 9000, 20000 to the end of List_1
 - B. print the updated list which is List_1
- Hint : should be added in 1 line of code

```
In [5]: #A. Add 7000, 9000, 20000 to the end of List_1
List_1.extend([7000, 9000, 20000])

#B. print the updated list which is List_1
print(List_1)
```

[102.5, 104.2, 103.8, 105.6, 106.3, 107.1, 106.5, 108.4, 109.2, 110.0, 500, 10000, 7000, 9000, 20000]

Question #5

- A. Create List_3 with the below data
20.5, 4.2, 3.8, 105.6, 200.3, 107.1, 300.5, 400.4, 500.2, 600.0
- B. print this list
- C. print by accessing the Last item which is 600
- D. insert 1000 at the end of List_3
- E. print List_3
- F. insert 0 at the start of List_3

```
In [6]: #A. Create List_3 with the below data
List_3 = [20.5, 4.2, 3.8, 105.6, 200.3, 107.1, 300.5, 400.4, 500.2, 600.0]

#B. print this list
print(List_3)

#C. print by accessing the Last item which is 600
print(List_3[-1])

#D. insert 1000 at the end of List_3
List_3.append(1000)

#E. print List_3
print(List_3)

#F. insert 0 at the start of List_3
List_3.insert(0, 0)

print(List_3)
```

```
[20.5, 4.2, 3.8, 105.6, 200.3, 107.1, 300.5, 400.4, 500.2, 600.0]
600.0
[20.5, 4.2, 3.8, 105.6, 200.3, 107.1, 300.5, 400.4, 500.2, 600.0, 1000]
[0, 20.5, 4.2, 3.8, 105.6, 200.3, 107.1, 300.5, 400.4, 500.2, 600.0, 1000]
```

Question #6

- A. Remove 1000 from List_3
 - B. print this list
- Hint: specify 1000 in your code

```
In [7]: #A. Remove 1000 from List_3
List_3.remove(1000)

#B. print this list
print(List_3)
```

```
[0, 20.5, 4.2, 3.8, 105.6, 200.3, 107.1, 300.5, 400.4, 500.2, 600.0]
```

Question #7

- A. Remove the 1st element from List_3
 - B. print this list
- Hint: specify the element position

```
In [11]: #A. Remove the 1st element from List_3
List_3.pop(0)

#B. print this list
print(List_3)
```

```
[20.5, 4.2, 3.8, 105.6, 200.3, 107.1, 300.5, 400.4, 500.2, 600.0]
```

Question #8

- A. Fetch the index position of 200.3

```
In [12]: #A. Fetch the index position of 200.3  
print(List_3.index(200.3))
```

4

Question #9

A. Create List_4 with the below data

[100, 100, 100, 20, 30, 40, 50]

B. print this list

C. Fetch or return the number of times 100 appears on the List_4

```
In [13]: #A. Create List_4 with the below data  
List_4 = [100, 100, 100, 20, 30, 40, 50]
```

```
#B. print this list  
print(List_4)
```

```
#C. Fetch or return the number of times 100 appears on the List_4  
print(List_4.count(100))
```

[100, 100, 100, 20, 30, 40, 50]

3

Question #10

A. Print List_3

B. Reverse the items of this list

C. Print List_3

D. Now, sort the List_3 in ascending order

E. Print the list

```
In [14]: #A. Print List_3  
print(List_3)
```

```
#B. Reverse the items of this list  
List_3.reverse()
```

```
#C. Print List_3  
print(List_3)
```

```
#D. Now, sort the List_3 in ascending order  
List_3.sort()
```

```
#E. Print the list  
print(List_3)
```

[20.5, 4.2, 3.8, 105.6, 200.3, 107.1, 300.5, 400.4, 500.2, 600.0]
[600.0, 500.2, 400.4, 300.5, 107.1, 200.3, 105.6, 3.8, 4.2, 20.5]
[3.8, 4.2, 20.5, 105.6, 107.1, 200.3, 300.5, 400.4, 500.2, 600.0]

In []:

```
# TTP Foundation 4
```



Control Flow



Loops and Conditional Statements



The syntax for an 'if' conditional statement is as follows:

```
if (condition_1):  
    statement_block_1  
  
elif (condition_2):  
    statement_block_2  
  
elif (condition_3):  
    statement_block_3
```

Control Flow - These decide what should run based on conditions.

if - run a block if condition is true, checks a condition

elif - extra conditions

else - fallback, doesn't check a condition, default action when all conditions fail

continue - skip to next loop iteration

break - exit the loop completely

Key rule:

One : if per chain

Zero or more : elif

Zero or one : else

Basic If–Elif–Else Conditions

Class Exercise 1

Write a Python program that checks the value of a variable score and prints:

"Outstanding" if the score is greater than 90

"Excellent" if the score is between 80 and 90

"Good" if the score is between 70 and 80

"You Need To Improve" for all other values, Assume score=88

```
In [1]: score = 88

if score > 90:
    print("Outstanding")
elif score > 80 and score < 90:
    print("Excellent")
elif score > 70 and score < 80:
    print("Good")
else:
    print("You Need To Improve")
```

Excellent

Class Exercise 2

Write a Python program that checks the current stock price stored in the variable stock_price and prints:

"We will buy 50 shares of ABC" if the price is less than 999

"We will buy 20 shares of ABC" if the price is exactly 1000

"We will buy 150 shares of ABC" if the price is greater than 1000

Assume stock_price = 1000.

```
In [2]: stock_price = 1500 # then...

if (stock_price < 999):
    print("We will buy 50 shares of ABC")

elif (stock_price == 1000):
    print("We will buy 20 shares of ABC")

elif (stock_price > 1000):
    print("We will buy 150 shares of ABC")
```

We will buy 150 shares of ABC

Class Exercise 3

Write a Python program that checks the price of a stock stored in the variable stock_price_ABC and prints:

"We will sell the stock and book the profit" if the price is greater than 250

Otherwise, print "We will keep buying the stock"

Assume stock_price_ABC = 200.

```
In [3]: stock_price_ABC = 200 # then...

if stock_price_ABC > 250: # if condition 1 is false then....
    print("We will sell the stock and book the profit")

else:
    # this block of code will be executed
    print(" We will keep buying the stock")
```

We will keep buying the stock

A list contains many values, so you use a for loop to check each price one by one with the same if-elif logic.

Loop Through Lists & Apply Conditions

A loop statement allows us to execute a statement or group of statements multiple times. The general syntax for a 'for' loop is :

```
for (variable) in sequence:
    block of statements
```

Here, the block of statements within the loop will get executed, until all 'sequence' elements get exhausted. Once all sequence elements are exhausted, the program will come out of the loop.

Class Exercise 4

Write a Python program that loops through a list of stock prices and prints a trading action for each price based on the following rules:

If the price is less than 999, print: "price → We will buy 50 shares of ABC"

If the price is exactly 1000, print: "price → We will buy 20 shares of ABC"

If the price is greater than 1000, print: "price → We will buy 150 shares of ABC"

Use the list: prices = [995, 1000, 1020, 980, 1000, 1010]

```
In [4]: prices = [995, 1000, 1020, 980, 1000, 1010]

for stock_price in prices:

    if stock_price < 999:
        print(stock_price, "-> We will buy 50 shares of ABC")

    elif stock_price == 1000:
        print(stock_price, "-> We will buy 20 shares of ABC")

    elif stock_price > 1000:
        print(stock_price, "-> We will buy 150 shares of ABC")
```

995 → We will buy 50 shares of ABC
 1000 → We will buy 20 shares of ABC
 1020 → We will buy 150 shares of ABC
 980 → We will buy 50 shares of ABC
 1000 → We will buy 20 shares of ABC
 1010 → We will buy 150 shares of ABC

all three (while, continue, break) belong to loops.

While Loop Example

while → A type of loop

It repeats as long as a condition stays true.

Class Exercise 5

Write a Python program using a while loop that starts a counter at 0 and keeps printing the counter value as long as it is less than or equal to 5. After printing each value, increase the counter by 1.

```
In [5]: import time

counter = 0

while counter <= 5:
    print(counter)
    counter += 1
    time.sleep(1)
```

0
 1
 2
 3
 4
 5

✓ **continue** → Skip the current iteration

It jumps to the next loop cycle.

Range Syntax : range(start, stop, step) - default start = 0, step = 1

Class Exercise 6

Write a Python program that uses a for loop to print numbers from 1 to 5, but skip the number 3 using the continue statement.

```
In [6]: for i in range(1, 6):
    if i == 3:
        continue # skip 3
    print(i)
```

1
2
4
5

✓ break → Stop the loop completely

It exits the loop immediately, even if items are remaining.

Class Exercise 7

Write a Python program that uses a for loop to print numbers from 1 onward, but stop the loop completely when the number becomes 3 using the break statement.

```
In [7]: for i in range(1, 6):
    if i == 3:
        break # stop at 3
    print(i)
```

1
2

✓ Summary:

while = loop that runs based on a condition

continue = skip one iteration

break = exit the loop fully

Example — Using while for Continuous Checking (simulation)

Class Exercise 8

Write a Python program that starts with a stock price of 95 and keeps increasing it by 2 inside a while loop.

As long as the price is less than the target price of 105, print the current price.

Once the loop finishes, print "Target reached".

```
In [8]: price = 95
target = 105

while price < target:
    print("Current price:", price)
    price += 2
print("Target reached")
```

```
Current price: 95
Current price: 97
Current price: 99
Current price: 101
Current price: 103
Target reached
```

Class Exercise 9

Given a list of closing prices for a stock, write a Python program that loops through each price and prints:

"We Buy" if the price is less than 1200

"No new positions" if the price is exactly 1200

"We Sell" if the price is greater than 1200

After the loop ends, print "We are now out of the loop".

Use the price list: Close_Price_Stock = [1000, 1010, 1200, 1250, 1200, 1300, 1397, 1400, 1425, 1410]

In [38]: # Closing prices of the stock over a period

```
Close_Price_Stock = [1000, 1010, 1200, 1250, 1200,
                     1300, 1397, 1400, 1425, 1410] # Our sequence

for i in Close_Price_Stock:

    if i < 1200:
        print("We Buy")

    elif i == 1200:
        print("No new positions")

    elif i > 1200:
        print("We Sell")

print("We are now out of the loop")
```

```
We Buy
We Buy
No new positions
We Sell
No new positions
We Sell
We Sell
We Sell
We Sell
We Sell
We Sell
We are now out of the loop
```

The variable ‘i’ first stores the value ‘1000’ in it and runs it through the loop to execute the statements.

Now, ‘i’ will run through the sequence and pick the second element of the sequence which is ‘1010’.

Similarly, it will keep executing all the elements of the loop. Observe the output.

Class Exercise 10

Write a Python program that uses a while loop to start with $x = 0$ and keep increasing x by 1 on each iteration.

As long as x is less than or equal to 15, print the value of x .

After the loop finishes, print "We are now out of the loop".

```
In [10]: x = 0
```

```
while x <=15:  
    x = x + 1  
    print(x)  
print("We are now out of the loop")
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
We are now out of the loop
```

Class Exercise 11

Write a Python program that uses a for loop with range(1, 16, 2) to iterate through odd numbers from 1 to 15.

Inside the loop, increase the value of x by 1 and print it.

After the loop finishes, print "We are out of loop".

```
In [11]: for x in range (1, 16, 2):
```

```
    x = x + 1  
    print(x)
```

```
print("We are out of loop")
```

```
2  
4  
6  
8  
10  
12  
14  
16  
We are out of loop
```

Append Values

Class Exercise 12

Write a Python program that loops through a list of stock prices and uses append() to add trading signals to a new list called signal based on these rules:

If the price is greater than 70, append "buy" to the list

If the price is less than 70, append "sell" to the list

If the price is exactly 70, append "hold" to the list

Use the price list: price = [70.2, 70, 80, 60.6]

Finally, print the signal list.

```
In [12]: # Price of a stock
price = [70.2, 70, 80, 60.6]
signal = []
```

```
In [13]: type(signal)
```

```
Out[13]: list
```

```
In [14]: # Pay attention to the indentation and the syntax
```

```
for i in range(len(price)):

    if price[i]>70 :
        signal.append('buy')

    elif price[i]< 70:
        signal.append('sell')

    else:
        signal.append('hold')
```

```
In [15]: print(signal)
```

```
['buy', 'hold', 'buy', 'sell']
```

Looping Over Lists, Strings & Dictionaries

Class Exercise 13

Create a list called top_gainers containing the stock names: ['BHARTIARTL', 'EICHERMOT', 'HCLTECH', 'BAJFINANCE', 'RELIANCE']

Write a Python program that loops through this list and prints each stock along with its index number. Use the .index() method inside the print() statement to display the index.

```
In [16]: top_gainers= ['BHARTIARTL', 'EICHERMOT', 'HCLTECH', 'BAJFINANCE', 'RELIANCE']
```

```
In [17]: for i in top_gainers:
    print(top_gainers.index(i), ' : ', i)
```

```
0 : BHARTIARTL
1 : EICHERMOT
2 : HCLTECH
3 : BAJFINANCE
4 : RELIANCE
```

.index() searches the list top_gainers
 It finds the position of the current value (gainer)
 And returns the index number (0, 1, 2, 3 ...)

Strings in Python are iterable objects. In other words, strings are a sequence of characters. Hence, we can use a string as a sequence object in the for loop.

```
In [18]: program = "PYTHON"

for x in program:
    print(x)
```

P
Y
T
H
O
N

Looping through dictionaries involves a different approach as compared to lists and strings. As dictionaries are not index based, we need to use its built-in items() method as below:

Class Exercise 14

Create a dictionary named dict that stores stock tickers as keys and their prices as values:

```
{'AAPL':193.53, 'HP':24.16, 'MSFT':108.29, 'GOOG':1061.49}
```

Write a Python program that loops through the dictionary using .items() and prints each stock's name and its price in the format:

Price of AAPL is 193.53

```
In [19]: dict = {'AAPL':193.53, 'HP':24.16, 'MSFT':108.29, 'GOOG':1061.49}

for key, value in dict.items():
    print(f'Price of {key} is {value}')
```

Price of AAPL is 193.53
 Price of HP is 24.16
 Price of MSFT is 108.29
 Price of GOOG is 1061.49

```
In [20]: dict = {'AAPL':193.53, 'HP':24.16, 'MSFT':108.29, 'GOOG':1061.49}

for x, y in dict.items():
    print(f'Price of {x} is {y}')
```

Price of AAPL is 193.53
 Price of HP is 24.16
 Price of MSFT is 108.29
 Price of GOOG is 1061.49

In [21]: `# If we execute dict.items() directly, it returns dict items in the form of tuples`

Out[21]: `dict_items([('AAPL', 193.53), ('HP', 24.16), ('MSFT', 108.29), ('GOOG', 1061.49)])`

List Comprehensions

SYNTAX [expression for item in iterable]

List comprehension is an elegant way to define and create a list in Python. It is used to create a new list from another sequence, just like a mathematical set notation in a single line. As we see in the Python code above, list comprehension starts and ends with square brackets to help us remember that the output will be a list.

Class Exercise 15

Write a Python program that creates an empty list called square_list.

Using a for loop with range(0, 10), compute the square of each number and use append() to add the result to the list.

After each append, print the updated square_list.

```
In [22]: square_list = []

for x in range(0, 10):
    square_list.append(x ** 2)
    print(square_list)

[0]
[0, 1]
[0, 1, 4]
[0, 1, 4, 9]
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16, 25]
[0, 1, 4, 9, 16, 25, 36]
[0, 1, 4, 9, 16, 25, 36, 49]
[0, 1, 4, 9, 16, 25, 36, 49, 64]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Type *Markdown* and *LaTeX*: α^2

Class Exercise 16

Rewrite the logic of generating squares of numbers from 0 to 9 using list comprehension instead of a for loop.

Create a list that contains x^2 for every x in range(0, 10).

```
In [23]: # Now using List Comprehension
[x ** 2 for x in range(0,10)]
```

```
Out[23]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Class Exercise 17

Filter positive numbers less than 20 that are divisible by both 2 and 3 using list comprehension instead of a for loop.

Use range(0, 20)

```
In [24]: # We can have multiple condition to filter the output
# Have a list of all positive numbers less than 50 which are divisible by 2 and 3

[x for x in range(0,20) if x%2==0 if x%3==0] # gives us remainder
```

```
Out[24]: [0, 6, 12, 18]
```

```
In [25]: 5 % 2
```

```
Out[25]: 1
```

```
In [26]: 6 % 2
```

```
Out[26]: 0
```

Example of IF + ELSE in List Comprehension

```
In [27]: labels = ["Even" if x % 2 == 0 else "Odd" for x in range(6)]
print(labels)
```

```
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']
```

Trading Signal Generator (BUY / SELL)

Class Exercise 18

Create two lists named short_ma and long_ma that store short-term and long-term moving average values:

```
short_ma = [100, 102, 104, 106, 105, 103, 101]
long_ma = [101, 101, 102, 103, 104, 104, 104]
```

Write a Python program that loops through both lists (starting from index 1) and prints:

“Buy signal” when short_ma crosses above long_ma

“Sell signal” when short_ma crosses below long_ma

“Hold” otherwise

```
In [28]: short_ma = [100, 102, 104, 106, 105, 103, 101]
long_ma = [101, 101, 102, 103, 104, 104, 104]

for i in range(1, len(short_ma)):
    # start from 1 to compare today's value
    if short_ma[i-1] < long_ma[i-1] and short_ma[i] > long_ma[i]:
        print(f"Day {i}: Buy signal")
    elif short_ma[i-1] > long_ma[i-1] and short_ma[i] < long_ma[i]:
        print(f"Day {i}: Sell signal")
    else:
        print("Hold")
```

Day 1: Buy signal
Hold
Hold
Hold
Day 5: Sell signal
Hold

zip() gives one pair at a time

A Loop takes one pair at a time

We only convert it to a list, when we want to see all values together

The zip() function - Iterable

```
In [29]: company_names = ['Apple', 'Microsoft', 'Tesla']
tickers = ['AAPL', 'MSFT', 'TSLA']
z = zip(company_names, tickers)
print(z)
```

<zip object at 0x0000017AC6FA7E80>

```
In [30]: z_list = list(z)
print(z_list)
```

[('Apple', 'AAPL'), ('Microsoft', 'MSFT'), ('Tesla', 'TSLA')]

Class Exercise 19

Create three lists named price, ma_value, and rsi_value that store price, moving average, and RSI values for 20 periods:

```
price = [102.5, 104.1, 103.8, 105.7, 107.2, 106.4, 108.9, 109.3, 110.1, 111.8, 112.4, 113.2, 114.0, 115.6, 116.8, 118.1, 117.5, 119.3, 120.0, 121.4]
```

```
ma_value = [101.8, 102.3, 102.9, 103.5, 104.2, 105.0, 105.8, 106.6, 107.4, 108.1, 108.9, 109.7, 110.3, 111.0, 111.8, 112.6, 113.2, 113.9, 114.7, 122.5]
```

```
rsi_value = [42, 45, 48, 52, 55, 58, 60, 64, 67, 70, 73, 69, 65, 62, 58, 55, 52, 49, 46, 43]
```

Write a Python program that loops through these lists using zip() and prints:

BUY when price > MA and RSI > 40

SELL when price < MA

HOLD in all other cases

Add logic to ensure:

Before the first BUY ever happens → output should be only BUY or None

After the first BUY → use normal BUY / SELL / HOLD behaviour

Store all signals inside a list named signals.

```
In [31]: price = [100, 104.1, 103.8, 105.7, 107.2, 106.4, 108.9, 109.3, 110.1, 111.8,
               112.4, 113.2, 114.0, 115.6, 116.8, 118.1, 117.5, 119.3, 120.0, 121.4]
ma_value = [101.8, 102.3, 102.9, 103.5, 104.2, 105.0, 105.8, 106.6, 107.4, 108.
            108.9, 109.7, 110.3, 111.0, 111.8, 112.6, 113.2, 113.9, 114.7, 122
rsi_value = [42, 45, 48, 52, 55, 58, 60, 64, 67, 70,
              73, 69, 65, 62, 58, 55, 52, 49, 46, 43]
```

In [32]: `import time`

```

signals = [] # Where signals will be stored
in_position = False          # Means we have NOT bought yet, Initial State
first_buy_done = False        # Means NO BUY has happened ever, Initial State

for p, ma, rsi in zip(price, ma_value, rsi_value):

    buy_condition = (p > ma) and (rsi > 40)
    sell_condition = (p < ma)

    # CASE 1: If we have NEVER done any BUY yet
    if first_buy_done == False:

        # Check if BUY should happen for the first time
        if buy_condition == True:
            position = "BUY"
            in_position = True
            first_buy_done = True    # First BUY has now happened

        else:
            position = None         # No HOLD allowed before first BUY

    # CASE 2: Normal BUY/SELL/HOLD logic after FIRST BUY is done
    else:

        # If BUY condition is true AND we are NOT already in a position
        if buy_condition == True and in_position == False:
            position = "BUY"
            in_position = True

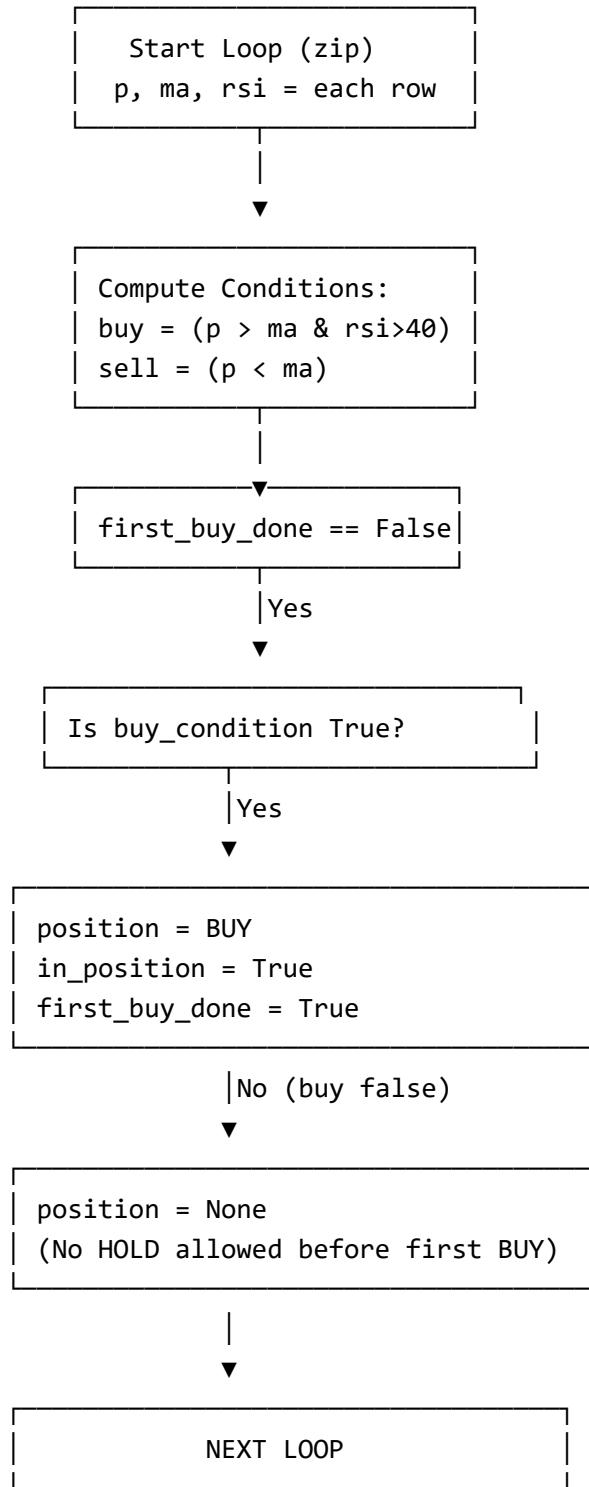
        # If SELL condition is true AND we ARE in a position
        elif sell_condition == True and in_position == True:
            position = "SELL"
            in_position = False

        # In all other cases → HOLD
        else:
            position = "HOLD"

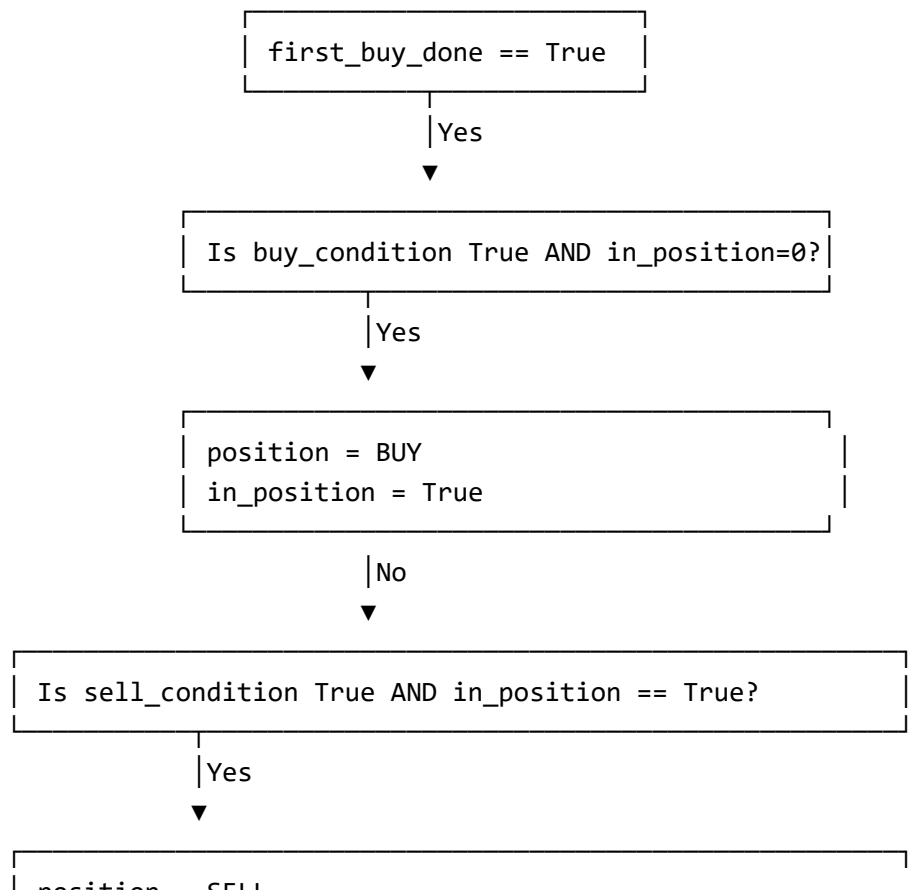
    print(f"Price:{p} MA:{ma} RSI:{rsi} -> {position}")
    signals.append(position)
    time.sleep(1)

```

```
Price:100  MA:101.8  RSI:42  ->  None
Price:104.1  MA:102.3  RSI:45  ->  BUY
Price:103.8  MA:102.9  RSI:48  ->  HOLD
Price:105.7  MA:103.5  RSI:52  ->  HOLD
Price:107.2  MA:104.2  RSI:55  ->  HOLD
Price:106.4  MA:105.0  RSI:58  ->  HOLD
Price:108.9  MA:105.8  RSI:60  ->  HOLD
Price:109.3  MA:106.6  RSI:64  ->  HOLD
Price:110.1  MA:107.4  RSI:67  ->  HOLD
Price:111.8  MA:108.1  RSI:70  ->  HOLD
Price:112.4  MA:108.9  RSI:73  ->  HOLD
Price:113.2  MA:109.7  RSI:69  ->  HOLD
Price:114.0  MA:110.3  RSI:65  ->  HOLD
Price:115.6  MA:111.0  RSI:62  ->  HOLD
Price:116.8  MA:111.8  RSI:58  ->  HOLD
Price:118.1  MA:112.6  RSI:55  ->  HOLD
Price:117.5  MA:113.2  RSI:52  ->  HOLD
Price:119.3  MA:113.9  RSI:49  ->  HOLD
Price:120.0  MA:114.7  RSI:46  ->  HOLD
Price:121.4  MA:122.5  RSI:43  ->  SELL
```



AFTER FIRST BUY DONE



Class Exercise 20

Create two lists named `short_ma` and `long_ma` that store short-term and long-term moving average values:

`short_ma = [100, 102, 104, 106, 105, 103, 101]`

`long_ma = [101, 101, 102, 103, 104, 104, 104]`

Write a Python program that loops through both lists starting from index 1 and prints:

“Buy signal” when `short_ma` crosses above `long_ma`

“Sell signal” when `short_ma` crosses below `long_ma`

“Hold” in all other cases

Use yesterday’s values (`i-1`) and today’s values (`i`) to detect the crossover.

```
In [36]: short_ma = [100, 102, 104, 106, 105, 103, 101]
long_ma = [101, 101, 102, 103, 104, 104, 104]

for i in range(1, len(short_ma)):      # start from 1 to compare today's value
    if short_ma[i-1] < long_ma[i-1] and short_ma[i] > long_ma[i]:
        print(f"Day {i}: Buy signal")
    elif short_ma[i-1] > long_ma[i-1] and short_ma[i] < long_ma[i]:
        print(f"Day {i}: Sell signal")
    else:
        print(f"Day {i}: Hold")
```

```
Day 1: Buy signal
Day 2: Hold
Day 3: Hold
Day 4: Hold
Day 5: Sell signal
Day 6: Hold
```

```
In [ ]:
```

TTP Assignment 4

Question #1

- A. create a new empty dictionary, name it misc_dict
- B. check and print data type of misc_dict

```
In [1]: #A. create a new empty dictionary, name it misc_dict
misc_dict = {}

#B. check and print data type of misc_dict
print(type(misc_dict))

<class 'dict'>
```

Question #2

- A. Update the misc_dict with the below data
name - virat, sport - cricket, age - 39, category - batsman
- B. check and print misc_dict

```
In [2]: #A. Update the misc_dict with the below data
misc_dict.update({"name" : "virat",
                  "sport": "cricket",
                  "age" : 39,
                  "category" : "batsman"
})

#B. check and print misc_dict
print(misc_dict)

{'name': 'virat', 'sport': 'cricket', 'age': 39, 'category': 'batsman'}
```

Question #3

- A. Update misc_dict category with all-rounder
- B. print misc_dict
- C. create a new key-value pair and name it rating - A+ D. print misc_dict

```
In [4]: #A. Update misc_dict category with all-rounder
misc_dict ["category"] = " all-rounder"

#B. print misc_dict
print(misc_dict)

#C. create a new key-value pair and name it rating - A+ D. print misc_dict
misc_dict["rating"] = "A+D"

print(misc_dict)

{'name': 'virat', 'sport': 'cricket', 'age': 39, 'category': ' all-rounder'}
{'name': 'virat', 'sport': 'cricket', 'age': 39, 'category': ' all-rounder',
 'rating': 'A+D'}
```

Question #4

- A. check length of misc_dict
- B. Return all keys
- C. Return all values
- D. Delete rating E. print misc_dict

```
In [7]: #A. check length of misc_dict
print(len(misc_dict))

#B. Return all keys
print(misc_dict.keys())

#C. Return all values
print(misc_dict.values())

#D. Delete rating
del misc_dict["rating"]

# E. print misc_dict
print(misc_dict)
```

```
5
dict_keys(['name', 'sport', 'age', 'category', 'rating'])
dict_values(['virat', 'cricket', 39, 'all-round', 'A+D'])
{'name': 'virat', 'sport': 'cricket', 'age': 39, 'category': 'all-round'}
```

Question #5

- A. create the new dictionary and name it misc_dict1 with below data
Arjun: 47, Meera: 65, Kabir: 76, Isha: 89, Vikram: 95, Tara: 82, Nikhil: 88
- B. print misc_dict1
- C. create a variable sorted_names and sort by keys
- D. print the variable sorted_names, it should come in ascending order

```
In [8]: #A. create the new dictionary and name it misc_dict1 with below data
misc_dict1 = {"Arjun": 47,
              "Meera": 65,
              "Kabir": 76,
              "Isha": 89,
              "Vikram": 95,
              "Tara": 82,
              "Nikhil": 88}

#B. print misc_dict1
print(misc_dict1)

#C. create a variable sorted_names and sort by keys
sorted_names = sorted(misc_dict1.keys())

#D. print the variable sorted_names, it should come in ascending order
print(sorted_names)
```

```
{'Arjun': 47, 'Meera': 65, 'Kabir': 76, 'Isha': 89, 'Vikram': 95, 'Tara': 82, 'Nikhil': 88}
['Arjun', 'Isha', 'Kabir', 'Meera', 'Nikhil', 'Tara', 'Vikram']
```

Question #6

- A. Now, create a variable sorted_scores and sort by values for misc_dict1
- B. print the variable sorted_scores, this will print names sorted by scores(in ascending order)
- C. clear all contents of misc_dict1
- D. print misc_dict1

```
In [9]: #A. Now, create a variable sorted_scores and sort by values for misc_dict1
sorted_scores = sorted(misc_dict1, key = misc_dict1.get)

#B. print the variable sorted_scores, this will print names sorted by scores(i
print(sorted_scores )

#C. clear all contents of misc_dict1
misc_dict1.clear()

#D. print misc_dict1
print(misc_dict1)

['Arjun', 'Meera', 'Kabir', 'Tara', 'Nikhil', 'Isha', 'Vikram']
{}
```

Question #7

- A. create TUPLE ttp_tp with below data
35, 65, 75, 85, 95, 105, 115
- B. check data type of ttp_tp
- C. print the length of ttp_tp

```
In [11]: #A. create TUPLE ttp_tp with below data
ttp_tp = (35, 65, 75, 85, 95, 105, 115)

#B. check data type of ttp_tp
print(type(ttp_tp))

#C. print the Length of ttp_tp
print(len(ttp_tp))

<class 'tuple'>
7
```

Question #8

- A. Access the 1st value in ttp_tp
- B. Access the last value
- C. Access 75

```
In [14]: #A. Access the 1st value in ttp_tp
print(ttp_tp[0])

#B. Access the last value
print(ttp_tp[-1])

#C. Access 75
print(ttp_tp[2])
```

35
115
75

Question #9

- A. Using Tuple Slicing, Access element 0 to 5
- B. Access last 3 elements
- F. Access all elements till 3rd element
- G. Access all elements but not 35

```
In [12]: #A. Using Tuple Slicing, Access element 0 to 5
print(ttp_tp[0:6])

#B. Access last 3 elements
print(ttp_tp[-3:])

#F. Access all elements till 3rd element
print(ttp_tp[ :3])

#G. Access all elements but not 35
print(ttp_tp[1: ])
```

(35, 65, 75, 85, 95, 105)
(95, 105, 115)
(35, 65, 75)
(65, 75, 85, 95, 105, 115)

Question #10

- A. Access all elements till 3rd element
- B. Access all elements but not 35

```
In [13]: #A. Access all elements till 3rd element
print(ttp_tp[ :3])

#B. Access all elements but not 35
print(ttp_tp[1:])
```

(35, 65, 75)
(65, 75, 85, 95, 105, 115)

In []:

```
# TTP Foundation 5
```

Functions in Python

A function in Python is a block of reusable code that performs a specific task.

Instead of writing the same code multiple times, you can define it once as a function and call it whenever needed.

A simple user-defined function

The syntax for constructing a function is:

```
def function_name (parameter-list):
    Statements, i.e function body
    return a value, if required
```

Class Exercise 1

Write a Python function named greet() which, when called, prints the message "Welcome To Coding".

```
In [1]: def greet():
           print("Welcome To Coding")
```

```
In [2]: greet()
           Welcome To Coding
```

```
In [3]: def greet_user(name):
           print("Hello, ", name)

greet_user("My name is ViveK")
```

```
Hello, My name is ViveK
```

```
In [4]: greet_user("Welcome to TTP")
           Hello, Welcome to TTP
```

Class Exercise 2

Define a function square(x) that prints the square of the number passed to it. Call the function with the value 5.

```
In [5]: def square(x):
    print("This is a square", x**2)

square(3)
```

This is a square 9

Function with Return Value

Functions can also return results instead of just printing them.

```
In [6]: def square(x):
    return x*x
```

```
In [7]: result = square(5)
print(result)
```

25

```
In [8]: result2 = square(10)
print(result2)
```

100

```
In [9]: # Function with multiple variables
def add(a, b):
    return a + b

add(100, 200)
```

Out[9]: 300

Class Exercise 3

Define a function power(num, exp=2) that returns num raised to the power of exp. Call the function with the value 5 using the default exponent.

```
In [10]: # We have used default parameter exp = 2
def power(num, exp=4):
    return num**exp

power(5)
```

Out[10]: 625

Function with Conditions Inside

Class Exercise 4

Define a function check_trend(price1, price2) that compares two prices and returns "Uptrend" if price2 is greater than price1, otherwise returns "Downtrend".

```
In [11]: def check_trend(price1, price2):
    if price2 > price1:
        return "Uptrend"
    else:
        return "Downtrend"
```

```
In [12]: check_trend(100, 200)
```

```
Out[12]: 'Uptrend'
```

Class Exercise 5

Define a function compare(num1, num2) that compares two numbers and prints whether the first number is greater than, equal to, or less than the second number.

```
In [13]: def compare(num1, num2):
    if num1 > num2:
        print("Number 1 is greater than number 2.")
    elif num1 == num2:
        print("Number 1 is equal to number 2.")
    else:
        print("Number 1 is less than number 2.)")
```

```
In [14]: compare(5, 5)
```

Number 1 is equal to number 2.

Function Returning Multiple Value

Class Exercise 6

Define a function get_stats(a, b) that returns two values:

the sum of a and b

the difference of a and b

Call the function with a = 5 and b = 7 and store the result in variable x.

```
In [15]: def get_stats(a, b):
    return a+b, a-b

x = get_stats(5, 7)
```

```
In [16]: print(x)
```

(12, -2)

```
In [17]: type(x)
```

```
Out[17]: tuple
```

We can do Tuple Unpacking into separate Variables

```
In [18]: add_value, diff_value = get_stats(5, 7)
print(add_value, diff_value)
```

12 -2

```
In [19]: add_value
```

Out[19]: 12

```
In [20]: diff_value
```

Out[20]: -2

Class Exercise 7

Define a function ohlc_range(high, low) that calculates the range of a candle (high - low) and its midpoint ((high + low)/2).

Return both values from the function.

```
In [21]: def ohlc_range(high, low):
    rng = high - low
    mid = (high + low) / 2
    return rng, mid
```

```
In [22]: rng, mid = ohlc_range(2820, 2770) # (50, 2795.0)
```

```
In [23]: print(rng, mid)
```

50 2795.0

Class Exercise 8

Define a function signals_stats(close_today, sma_value) that returns:

the distance between the closing price and its SMA

a boolean value indicating whether the close is above the SMA

Call the function with close_today = 100 and sma_value = 98.

```
In [24]: def signals_stats(close_today, sma_value):
    return (close_today - sma_value), (close_today > sma_value)

distance, bullish = signals_stats(100, 98)
print(distance)
print(bullish)
```

2

True

Class Exercise 9

Define a function `sl_tp(price, sl_pct=0.02, tp_pct=0.04)` that returns the Stop-Loss (SL) and Take-Profit (TP) levels for a given price.

SL is calculated as price * (1 - sl_pct) and TP as price * (1 + tp_pct).

```
In [25]: def sl_tp(price, sl_pct=0.02, tp_pct=0.04):
    return price*(1-sl_pct), price*(1+tp_pct)
```

```
In [26]: sl_tp(100)
```

```
Out[26]: (98.0, 104.0)
```

Class Exercise 10

Write a function `functname(listname)` that iterates through a list, prints the running total at each step, and finally returns the total sum.

Use the list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
In [27]: list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def functname(listname):
    sum1 = 0
    for number in listname:
        sum1 = sum1 + number
        print("Running total:", sum1) # shows each addition
    return(sum1)

functname(list1)
```

```
Running total: 1
Running total: 3
Running total: 6
Running total: 10
Running total: 15
Running total: 21
Running total: 28
Running total: 36
Running total: 45
Running total: 55
```

```
Out[27]: 55
```

Class Exercise 11

Define a function `functname(listname)` that computes the running total of a list and returns a list containing each intermediate sum.

Call the function using `list1` and print the result.

```
In [28]: list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
In [29]: def functname(listname):
    sum1 = 0
    steps = []
    for number in listname:
        sum1 = sum1 + number
        steps.append(sum1)
    return steps

print(functname(list1))
```

```
[1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
```

Class Exercise 12

Define a function candle_type(price_open, price_close) that returns:
 "Bullish Candle" if the closing price is greater than the opening price
 "Bearish Candle" if the closing price is lower than the opening price
 "Doji Candle" if both prices are equal

```
In [30]: open_price = [
    100, 102, 101, 103, 105,
    104, 106, 108, 107, 110,
    112, 111, 113, 115, 114,
    116, 118, 117, 119, 120]

close_price = [
    101, 100, 103, 104, 106,
    103, 107, 109, 108, 111,
    110, 112, 114, 113, 115,
    117, 116, 118, 121, 119]
```

```
In [31]: def candle_type(price_open, price_close):
    if price_close > price_open:
        return "Bullish Candle"
    elif price_close < price_open:
        return "Bearish Candle"
    else:
        return "Doji Candle"
```

```
In [32]: candle_type(open_price, close_price)
```

```
Out[32]: 'Bullish Candle'
```

```
In [33]: candle_status = []

for x, y in zip(open_price, close_price):
    candle_status.append(candle_type(x, y))

print(candle_status)
```

```
['Bullish Candle', 'Bearish Candle', 'Bullish Candle', 'Bullish Candle', 'Bullish Candle',
 'Bearish Candle', 'Bullish Candle', 'Bullish Candle', 'Bullish Candle', 'Bullish Candle',
 'Bullish Candle', 'Bullish Candle', 'Bearish Candle', 'Bullish Candle', 'Bullish Candle',
 'Bullish Candle', 'Bearish Candle', 'Bullish Candle', 'Bullish Candle', 'Bullish Candle',
 'Bullish Candle', 'Bullish Candle', 'Bearish Candle', 'Bullish Candle', 'Bullish Candle']
```

Class Exercise 13

Write a program that checks whether two lists open_price and close_price have the same length.

If not, print an error message.

If yes, loop through the lists and append each candle's type (bullish/bearish/doji) into a list called candle_status.

Finally, print the candle_status list.

```
In [34]: candle_status = []

if len(open_price) != len(close_price):
    print("Error: open and close prices do not match in length!")
else:
    for i in range(len(open_price)):
        candle_status.append(candle_type(open_price[i], close_price[i]))

print(candle_status)

['Bullish Candle', 'Bearish Candle', 'Bullish Candle', 'Bullish Candle', 'Bullish Candle', 'Bearish Candle', 'Bullish Candle', 'Bullish Candle', 'Bullish Candle', 'Bullish Candle', 'Bullish Candle', 'Bearish Candle', 'Bullish Candle', 'Bearish Candle', 'Bullish Candle', 'Bullish Candle', 'Bullish Candle', 'Bearish Candle', 'Bullish Candle', 'Bullish Candle', 'Bullish Candle', 'Bearish Candle', 'Bullish Candle', 'Bullish Candle', 'Bearish Candle']
```

Function to calculate daily % change

Class Exercise 14

Define a function pct_change(price_list) that calculates the percentage change between each consecutive pair of prices.

The first value should be None (because no previous price exists).

Each subsequent value should be the rounded percentage change.

Write the complete function.

```
In [35]: prices = [102, 104, 103, 105, 108, 110, 109, 111, 113, 112,
               115, 117, 118, 116, 119, 121, 120, 122, 124, 123]

In [36]: def pct_change(price_list):
    pct_values = []

    for i in range(len(price_list)):
        if i == 0:
            pct_values.append(None) # No previous value for first element
        else:
            prev = price_list[i-1]
            curr = price_list[i]
            pct = ((curr - prev) / prev) * 100
            pct_values.append(round(pct, 2))

    return pct_values
```

```
In [37]: pct_vals = pct_change(prices)
pct_vals
```

```
Out[37]: [None,
 1.96,
 -0.96,
 1.94,
 2.86,
 1.85,
 -0.91,
 1.83,
 1.8,
 -0.88,
 2.68,
 1.74,
 0.85,
 -1.69,
 2.59,
 1.68,
 -0.83,
 1.67,
 1.64,
 -0.81]
```

Understanding Lambda

The lambda operator is a way to create small **anonymous functions** i.e. functions without a name.

They are temporary functions i.e. they are needed only where they have been created.

Can take any number of arguments, but only one expression.

Commonly used with functions like **map()**, **filter()**, and **sorted()**.

Automatically returns the result of the expression (no need to write return).

A simple lambda example

The general syntax for Lambda is as follows:

```
lambda argument_list: expression
```

```
In [38]: add = lambda x, y: x + y
print(add(2, 3))
```

5

Class Exercise 15

Create a lambda function to find the cube of a number.

```
In [39]: cube = lambda x: x**3
cube(3) # Output: 27
```

Out[39]: 27

map ()

One of the advantages of using a lambda is the map() function.

```
map (lambda, sequence of lists)
```

map() applies the lambda function to all elements within the sequence. These elements are generally lists.

Class Exercise 16

Using the map() function, create a lambda expression that adds the corresponding elements of three lists (list_1, list_2, list_3) and returns a new list containing the sums.

```
In [40]: # The Lists have to be of same Length to apply the map () function in Lambda.  
list_1 = [1, 2, 3, 4]  
list_2 = [10, 20, 30, 40]  
list_3 = [100, 200, 300, 400]
```

```
In [41]: list(map(lambda x, y, z: (x + y + z), list_1, list_2, list_3))
```

```
Out[41]: [111, 222, 333, 444]
```

```
In [42]: list(map(lambda x, y, z: print(x + y + z), list_1, list_2, list_3))
```

```
111  
222  
333  
444
```

```
Out[42]: [None, None, None, None]
```

filter ()

Another advantage of using a lambda is the filter() function.

```
filter (lambda, list)
```

It is an elegant way to filter out the required elements from a list.

```
In [43]: list_age = [1, 2, 3, 4, 3, 5, 8, 13, 21, 34, 55, 65, 70, 80, 90] # This is a
```

```
In [44]: list(filter(lambda x: x > 13, list_age))
```

```
Out[44]: [21, 34, 55, 65, 70, 80, 90]
```

Class Exercise 17

Using the filter() function, create a lambda expression that selects only the elements equal to "Buy" from the list signals. Write the one-line code to return a list containing only the "Buy" signals.

```
In [45]: signals = ['Buy', 'Sell', 'Sell', 'Buy',
                 'Buy', 'Sell', 'Buy'] # This is a list
```

```
In [46]: list(filter(lambda x: x == 'Sell', signals))
```

```
Out[46]: ['Sell', 'Sell', 'Sell']
```

Sort list of tuples by second value

sorted ()

Another advantage of using a lambda is the sorted() function. It is an elegant way to sort out the required elements from a list, tuple or sequence.

```
sorted(data, key=lambda x: expression)
```

Class Exercise 18

Use the sorted() function with a lambda as the key to sort the list based on the entire tuple. Write the code to store the sorted result in sorted_data and print it.

```
In [47]: data = [(1, 5), (3, 2), (4, 9), (2, 1)]
sorted_data = sorted(data, key=lambda x: x)
print(sorted_data)
```

```
[(1, 5), (2, 1), (3, 2), (4, 9)]
```

lambda x: x[1] tells Python to sort by the second element of each tuple.

Class Exercise 19

Use the sorted() function with a lambda as the key to sort the list based on the 2nd element of the tuple.

Write the code to store the sorted result in sorted_data and print it.

```
In [48]: data = [(1, 5), (3, 2), (4, 9), (2, 1)]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)
```

```
[(2, 1), (3, 2), (1, 5), (4, 9)]
```

Class Exercise 20

Define a function `order(symbol, qty, price)` that prints the order details.

Then call this function using:

positional arguments

keyword arguments

a mix of positional and keyword arguments

Write all three function calls.

```
In [49]: def order(symbol, qty, price):
    print(symbol, qty, price)

order("RELIANCE", 10, 2500)          # positional
order(symbol="RELIANCE", qty=10, price=2500) # keyword
order("RELIANCE", price=2500, qty=10)      # mixed

RELIANCE 10 2500
RELIANCE 10 2500
RELIANCE 10 2500
```

Class Exercise 21

Define a function `generate_signal(close, sma)` that returns "Buy" if the closing price is greater than the SMA value, otherwise returns "Sell".

```
In [50]: def generate_signal(close, sma):
    if close > sma:
        return "Buy"
    return "Sell"
```

```
In [51]: generate_signal(99, 100)
```

```
Out[51]: 'Sell'
```

Class Exercise 22

You are given the following trading dataset:

```
prices = [100, 104, 103, 108, 107, 112, 110, 115]
opens = [98, 102, 101, 105, 106, 109, 108, 112]
highs = [105, 106, 104, 110, 108, 115, 114, 118]
lows = [97, 100, 99, 103, 104, 108, 107, 110]
volumes = [1200, 900, 4000, 650, 3200, 5000, 800, 4200]
```

1. Using **filter()** and **lambda**, extract all high-volume values, where volume > 3000.
2. Compute daily **volatility = high – low** using **map()** and **lambda**.
3. Find the Mean of prices
4. Find the Mean of opens
5. Find the Mean of high

```
In [52]: prices = [100, 104, 103, 108, 107, 112, 110, 115]
opens = [98, 102, 101, 105, 106, 109, 108, 112]
highs = [105, 106, 104, 110, 108, 115, 114, 118]
lows = [97, 100, 99, 103, 104, 108, 107, 110]
volumes = [1200, 900, 4000, 650, 3200, 5000, 800, 4200]
```

```
In [53]: volume_filter = list(filter(lambda x: x>3000, volumes))
print(volume_filter)

[4000, 3200, 5000, 4200]
```

```
In [54]: volatility = list ( map (lambda h, l: h - l, highs, lows) )
print(volatility)

[8, 6, 5, 7, 4, 7, 7, 8]
```

```
In [55]: mean_price = (lambda p: sum(p) / len(p))(prices)
print(mean_price)

107.375
```

```
In [56]: mean_opens = (lambda p: sum(p) / len(p))(opens)
print(mean_opens)

105.125
```

```
In [57]: mean_highs = (lambda p: sum(p) / len(p))(highs)
print(mean_highs)

110.0
```

```
In [ ]:
```

```
In [ ]:
```

TTP Assignment 5

Question #1

- A. create a new empty set and name it set_1
- B. check and print data type of set_1
- C. print set_1

```
In [1]: set_1 = set()  
print(set_1)  
type(set_1)
```

set()

Out[1]: set

Question #2

- A. create a new set and name it set_2, it should contain ARGUMENT
- B. check and print data type of set_2

```
In [2]: set_2 = set ("ARGUMENT")  
print(set_2)  
type(set_2)
```

{'R', 'E', 'M', 'N', 'U', 'A', 'G', 'T'}

Out[2]: set

Question #3

- A. create a new set and name it set_3, it should contain LUSTROUS
- B. check and print data type of set_3
- ** check the ouput carefully

```
In [3]: set_3 = set("LUSTROUS")  
print(set_3)  
type(set_3)
```

{'S', 'R', 'L', 'O', 'U', 'T'}

Out[3]: set

Question #4

- A. Using == compare set_3 and set_4

```
In [4]: set_2 == set_3
```

Out[4]: False

Question #5

- A. create a new set and name it set_5, it should contain KLMNOPQ
- B. create another new set and name it set_6, it should contain PQRSTUV
- C. using union output a new set
- D. print set_5 E. print set_6 F. using == compare both these sets

```
In [5]: set_5 = set("KLMNOPQ")
set_6 = set("PQRSTUV")
print(set_5 | set_6)

print(set_5)
print(set_6)

set_5 == set_6

{'P', 'S', 'R', 'V', 'M', 'Q', 'L', 'O', 'N', 'U', 'T', 'K'}
{'P', 'M', 'Q', 'O', 'L', 'N', 'K'}
{'S', 'P', 'R', 'V', 'Q', 'U', 'T'}
```

Out[5]: False

Question #6

- A. find the common items that are present in set_5 and set_6
- B. Return the items of 'set_5' which are not common to the 'set_6'

```
In [6]: print(set_5.intersection(set_6))
print(set_5.difference(set_6))

{'P', 'Q'}
{'M', 'O', 'N', 'L', 'K'}
```

Question #7

- A. Removes all elements from set_5 that are also present in set_6.'

```
In [7]: set_5.difference_update(set_6)
print(set_5)
print(set_6)

{'M', 'O', 'L', 'N', 'K'}
{'S', 'P', 'R', 'V', 'Q', 'U', 'T'}
```

Question #8

- A. Check if set_5 and set_6 have some common elements'
- B. Create a new set_7 which contains KITE
- C. Create a new set_8 which contains COLD
- D. Check of set_7 and set_8 have some common elements

```
In [8]: print(set_5.isdisjoint(set_6))

set_7 = set("KITE")
set_8 = set("COLD")
print(set_7.isdisjoint(set_8))
```

True
True

Question #9

- A. print set_6
- B. Add SKY and update set_6
- C. print set_6
- D. Add 1 2 3 4 5 (which are 5 unique elements) and update set_6
- E. print set_6

```
In [13]: print(set_6)
set_6.add("SKY")
print(set_6)
```

```
{'S', 'P', 'R', '2', 'V', 'Q', '3', '4', 'U', 'T', '5', 'K', '1', 'Y'}
{'S', 'P', 'R', '2', 'V', 'SKY', 'Q', '3', '4', 'U', 'T', '5', 'K', '1',
 'Y'}
```

```
In [14]: set_6.update("12345")
print(set_6)
```

```
{'S', 'P', 'R', '2', 'V', 'SKY', 'Q', '3', '4', 'U', 'T', '5', 'K', '1',
 'Y'}
```

Question #10

- A. Remove/Discard SKY from set_6
- B. Remove arbitrary item from set_6
- C. Clear contents of set_6
- D. print set_6

```
In [15]: set_6.discard("SKY")
print(set_6)
```

```
{'S', 'P', 'R', '2', 'V', 'Q', '3', '4', 'U', 'T', '5', 'K', '1', 'Y'}
```

```
In [16]: set_5.pop()
```

```
Out[16]: 'M'
```

```
In [17]: print(set_6.clear())
```

```
print(set_6)
```

None
set()

```
In [ ]:
```

```
# TTP Foundation 6
```

NumPy

NumPy stands for "**Numerical Python**". It includes mathematical functions, random number generation, linear algebra routines, Fourier transforms, and more. NumPy is the computational engine driving many Python programs. That's because many popular data science libraries like pandas are built on top of it. The data is stored as an array called a `numpy.ndarray`. These arrays are somewhat like native Python lists, except that the data is homogeneous (all elements of the same type). NumPy arrays are the backbone of pandas and that's why pandas inherits the same speed advantage over raw lists.

Python Lists vs ⚡ NumPy Arrays

Why lists are slow:

- Dynamic typing: Each element in a list is a full Python object with type info, reference count, etc.
- Pointer chasing: Lists store references, not raw data. Accessing values means following pointers.
- No vectorization: Operations like loop through elements in Python space, which is slow.
- C-level implementation: NumPy uses optimized C/Fortran routines under the hood.
- NumPy Vectorization: Operations are applied to entire arrays at once, avoiding Python loops.

Typically, NumPy runs 10 - 100x faster for numerical operations because it avoids Python-level loops.

NumPy arrays are meant for fast numerical computation, so they expect all elements to be of the same type. If you add strings, NumPy is forced to convert the whole array into text, breaking all mathematical operations.

- ✓ NumPy is optimized only for numeric + boolean types.

In []: `import numpy as np`

Creating Numpy Arrays - Faster, compact, Vectorized lists

0D array/ Scalar: A single number, No Rows, No Columns

1D array / Vector: single axis, accessed with one index

2D array/ Matrix: two axes (rows x columns), accessed with two indices

s

Syntax to create NumPy Array: arr = np.array([1, 2, 3]), dtype=float)

```
In [3]: import numpy as np

scalar = np.array(5)
print(type(scalar))
print("Dimensions:", scalar.ndim)

<class 'numpy.ndarray'>
Dimensions: 0
```

```
In [9]: open_price = np.array([100, 102, 101, 103, 105])
close_price = np.array([101, 100, 103, 104, 106])
```

```
In [10]: type(open_price)
```

```
Out[10]: numpy.ndarray
```

```
In [11]: print("Dimensions:", open_price.ndim) #1D array
```

```
Dimensions: 1
```

```
In [12]: open_price
```

```
Out[12]: array([100, 102, 101, 103, 105])
```

```
In [13]: print(close_price)
```

```
[101 100 103 104 106]
```

Basic Array Operations

```
In [14]: Price_change = (close_price - open_price) / (open_price) * 100
```

```
In [15]: print(Price_change)
```

```
[ 1.          -1.96078431  1.98019802  0.97087379  0.95238095]
```

```
In [17]: a = Price_change
```

NumPy Indexing Convention

- NumPy uses zero-based indexing:
- → first row, first column 0,0
- → first row, second column 0,1
- → second row, first column 1,0
- **Rows are indexed first, then columns**

Access: Only one index is needed in 1D Arrays.

In [20]: `print(a)`

```
[ 1.           -1.96078431  1.98019802  0.97087379  0.95238095]
```

In [18]: `a[1]`

Out[18]: -1.9607843137254901

In [21]: `a[-2]`

Out[21]: 0.9708737864077669

In [22]: `a[-5]`

Out[22]: 1.0

`arr.ndim` -- number of dimensions

`arr.shape` -- rows, columns

`arr.size` -- total elements

In [23]: `# Array attributes (1D)`
`print("Dimensions:", a.ndim)`
`print("Shape:", a.shape)`
`print("Size:", a.size)`

Dimensions: 1

Shape: (5,)

Size: 5

2D - Two-dimensional Array

In [25]: `arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])`

In [26]: `# Array attributes (2D)`
`print(arr2d)`
`print("Shape:", arr2d.shape)`
`print("Dimensions:", arr2d.ndim) #1 Row and 1 Column Structure, its a 2D`
`print("Size:", arr2d.size)`

`[[1 2 3]`

`[4 5 6]`

`[7 8 9]]`

Shape: (3, 3)

Dimensions: 2

Size: 9

Indexing Rules

A 2-Dimensional Array consists of rows and columns, so you need to specify both rows and columns, to locate an element.

Use : for selecting “everything”

stop index is always excluded

a[0:2]

Meaning:

start at row 0

take up to row 2 (excluded)

so you get rows 0 and 1

In [29]: `print(arr2d)`

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

In [30]: *#Access first row, first column element*
`arr2d[0, 0]`

Out[30]: 1

In [31]: *#Access third row, second column element*
`arr2d[2, 1]`

Out[31]: 8

In [32]: `arr2d[1,1]`

Out[32]: 5

In [33]: `arr2d[:, :]`

Out[33]: `array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])`

Slicing

When you want to select a certain section of an array, then you slice it. It could be a bunch of elements in a one-dimensional array and/or entire rows and columns in a two-dimensional array.

Slicing syntax 1D: `array[start : stop : step]`

Slicing syntax 2D : `array[row_slice, column_slice]`

Use colon (:) to select a range - `array[start:stop]`

To select all rows and columns - `array[:, :]`

To skip by 2 - array[::2]

```
array[1:5:2, 0:6:3] rows 1 - 4 with step 2, columns 0 - 5 with step 3
```

Slicing a 1D array

You can slice a one-dimensional array in various ways:

- Print first few elements
- Print last few elements
- Print middle elements
- Print elements after certain step.

In [34]: `print(open_price)`

```
[100 102 101 103 105]
```

In [35]: `#start at index 0, stop before index 3.
print(open_price[0:3]) # [10 20 30]`

```
[100 102 101]
```

In [38]: `# start 3 elements from the end, go to the end.
print(open_price[2:5])`

```
[101 103 105]
```

In [39]: `# Indexing Last 3 elements
print(open_price[-3:])`

```
[101 103 105]
```

In [40]: `#take every 2nd element (step = 2).
print(open_price[::2])`

```
[100 101 105]
```

In [41]: `#start at index 1, then take every 2nd element.
print(open_price[1::2])`

```
[102 103]
```

Slicing a 2D array

In [42]: `print(arr2d)`

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

In [43]: `arr2d[1:,:]` #Slicing from row 2 and all columns

Out[43]: `array([[4, 5, 6],
 [7, 8, 9]])`

```
In [44]: arr2d[0,:2] #Slicing row 1 and upto column 2
```

```
Out[44]: array([1, 2])
```

```
In [46]: arr2d[0][:2] #Another Way To Index, But a 2 step process and slower than arr2d[0,:2]
```

```
Out[46]: array([1, 2])
```

```
In [47]: x = np.array([
    [0, 1, 2, 3, 4],
    [100, 110, 12, 130, 140],
    [200, 210, 22, 230, 240],
    [300, 310, 32, 330, 340],
    [400, 410, 420, 430, 404]
])

print(x)
print("Shape:", x.shape)
```

```
[[ 0   1   2   3   4]
 [100 110 12  130 140]
 [200 210 22  230 240]
 [300 310 32  330 340]
 [400 410 420 430 404]]
Shape: (5, 5)
```

```
In [48]: x[:, 0]
```

```
Out[48]: array([ 0, 100, 200, 300, 400])
```

Class Exercise 1 - 10

1. Index the 2nd and 4th rows.
2. Index the first three rows.
3. Index the last two rows.
4. Index the first column only.
5. Index the last column only.
6. Index the middle row (row 3).
7. Index the middle column (column 3).
8. Index the top-left 3x3 block.
9. Index the bottom-right 2x2 block.
10. Index every alternate row (rows 0, 2, 4).

```
In [49]: print(x)
```

```
[[ 0   1   2   3   4]
 [100 110 12  130 140]
 [200 210 22  230 240]
 [300 310 32  330 340]
 [400 410 420 430 404]]
```

```
In [54]: #Index the 2nd and 4th rows.  
x[[1,3], :]
```

```
Out[54]: array([[100, 110, 12, 130, 140],  
                 [300, 310, 32, 330, 340]])
```

```
In [55]: print(x)
```

```
[[ 0   1   2   3   4]  
 [100 110 12 130 140]  
 [200 210 22 230 240]  
 [300 310 32 330 340]  
 [400 410 420 430 404]]
```

```
In [56]: #Index the first three rows.  
x[0:3, :]
```

```
Out[56]: array([[ 0, 1, 2, 3, 4],  
                 [100, 110, 12, 130, 140],  
                 [200, 210, 22, 230, 240]])
```

```
In [57]: #Index the first three rows.  
x[[0,1,2], :]
```

```
Out[57]: array([[ 0, 1, 2, 3, 4],  
                 [100, 110, 12, 130, 140],  
                 [200, 210, 22, 230, 240]])
```

```
In [58]: #Index the last two rows.  
x[-2:, :]
```

```
Out[58]: array([[300, 310, 32, 330, 340],  
                 [400, 410, 420, 430, 404]])
```

```
In [59]: #Index the last two rows.  
x[[-1,-2], :]
```

```
Out[59]: array([[400, 410, 420, 430, 404],  
                 [300, 310, 32, 330, 340]])
```

```
In [60]: #Index the first column only.  
x[:, 0]
```

```
Out[60]: array([ 0, 100, 200, 300, 400])
```

```
In [61]: #Index the last column only.  
x[:, -1]
```

```
Out[61]: array([-4, 140, 240, 340, 404])
```

```
In [62]: #Index the middle row (row 3).  
x[2, :]
```

```
Out[62]: array([200, 210, 22, 230, 240])
```

```
In [63]: #Index the middle column (column 3).
x[2, :]
```

```
Out[63]: array([200, 210, 22, 230, 240])
```

```
In [64]: #Index the middle row (row 3).
x[:, 2]
```

```
Out[64]: array([ 2, 12, 22, 32, 420])
```

```
In [65]: #Index the top-left 3x3 block.
x[:3, :3]
```

```
Out[65]: array([[ 0, 1, 2],
 [100, 110, 12],
 [200, 210, 22]])
```

```
In [66]: #Index the bottom-right 2x2 block.
x[-2:, -2:]
```

```
Out[66]: array([[330, 340],
 [430, 404]])
```

```
In [67]: #Index every alternate row (rows 0, 2, 4).
x[::2, :]
```

```
Out[67]: array([[ 0, 1, 2, 3, 4],
 [200, 210, 22, 230, 240],
 [400, 410, 420, 430, 404]])
```

```
In [68]: # Boolean masking
prices = np.array([100, 102, 98, 105, 97])
mask = prices > 100
print("Mask:", mask)
print("Filtered prices:", prices[mask])
```

```
Mask: [False True False True False]
Filtered prices: [102 105]
```

Vectorization

Vectorization of code helps us write complex codes in a compact way and execute them faster.

It allows to **operate** or apply a function on a complex object, like an array, "at once" rather than iterating over the individual elements. NumPy supports vectorization in an efficient way.

Array operations with a scalar

Every element of the array is added/multiplied/operated with the given scalar. We will discuss:

- Addition
- Subtraction
- Multiplication

```
In [69]: new_list = [100.5, 201.5, 301.5, 450.5, 50.5, 60.6, 70.12, 80.45]
AB = np.array(new_list) # Creating a 1D array or vector
print(AB)

[100.5 201.5 301.5 450.5 50.5 60.6 70.12 80.45]
```

Vectorization using scalars - addition, subtraction, multiplication

```
In [70]: #Scalar Addition
AB_add = AB + 2.4
print(AB_add)

[102.9 203.9 303.9 452.9 52.9 63. 72.52 82.85]
```

```
In [71]: #Scalar Subtraction
BC = AB_add - 2.5
print(BC)

[100.4 201.4 301.4 450.4 50.4 60.5 70.02 80.35]
```

```
In [72]: #Scalar Multiplication
BC = AB_add * 2
print(BC)

[205.8 407.8 607.8 905.8 105.8 126. 145.04 165.7 ]
```

```
In [75]: CD = AB / 2
print(CD)

[ 50.25 100.75 150.75 225.25 25.25 30.3 35.06 40.225]
```

2D Array operations with another 2D array

This is only possible when the shape of the two arrays is same. For example, a (3,3) array can be operated with another (3,3) array.

```
In [76]: A = np.array([[1, 2, 3], [11, 22, 33], [111, 222, 333]]) # Array of shape 3,3
B = np.ones((3, 3)) # Array of shape 3,3
print(A)
print("-----")
print(B)

[[ 1  2  3]
 [11 22 33]
 [111 222 333]]
-----
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

```
In [77]: # Addition of 2 arrays of same dimensions (3, 3)
print("Adding of the arrays is element wise: ")
print(A + B)
```

Adding of the arrays is element wise:
[[2. 3. 4.]
[12. 23. 34.]
[112. 223. 334.]]

```
In [78]: # Subtraction of 2 arrays
print("Subtracting array B from A is element wise: ")
print(A - B)
```

Subtracting array B from A is element wise:
[[0. 1. 2.]
[10. 21. 32.]
[110. 221. 332.]]

```
In [79]: # Multiplication of 2 arrays
```

```
A1 = np.array([[1, 2, 3], [4, 5, 6]]) # Array of shape 2,3
A2 = np.array([[2, 0, -1], [0, 4, -1]]) # Array of shape 2,3

print("Array 1", A1)
print("-----")
print("Array 2", A2)
print("-----")
print("Multiplying two arrays: ", A1 * A2)
print("As you can see above, the multiplication happens element by element.")
```

Array 1 [[1 2 3]
[4 5 6]]

Array 2 [[2 0 -1]
[0 4 -1]]

Multiplying two arrays: [[2 0 -3]
[0 20 -6]]
As you can see above, the multiplication happens element by element.

Broadcasting

allows 2D Array operations with a 1D array or vector

NumPy supports broadcasting. Broadcasting allows us to combine objects of **different shapes** within a single operation. But, to perform this operation one of the matrices needs to be a vector with its length equal to one of the dimensions of the other matrix. **The vector must fit perfectly across either the rows or the columns of the matrix.**

In [86]: `import numpy as np`

```
A = np.array([[1, 2, 3], [11, 22, 33], [111, 222, 333]])
B = np.array([1, 2, 3])
C = np.array([[1], [2], [3]] )

print(A)
print("_____")
print(B)
print(C)
```

```
[[ 1   2   3]
 [ 11  22  33]
 [111 222 333]]
```

```
_____
[1 2 3]
[[1]
 [2]
 [3]]
```

In [87]: `print(A * C)`

```
[[ 1   2   3]
 [ 22  44  66]
 [333 666 999]]
```

In [83]: `print("Multiplication with broadcasting: ")`

```
print(A * B)
```

Multiplication with broadcasting:

```
[[ 1   4   9]
 [ 11  44  99]
 [111 444 999]]
```

In [88]: `import numpy as np`

```
A = np.array([[1, 2, 3], [11, 22, 33], [111, 222, 333]])
B = np.array([1])

print(A * B)
```

```
[[ 1   2   3]
 [ 11  22  33]
 [111 222 333]]
```

In [92]: `#A = np.array([[1, 2, 3], [11, 22, 33], [111, 222, 333]])`
`#B = np.array([1, 2])`

```
#print(A * B) # ValueError: operands could not be broadcast together with shape
```

- Comparison operators: Comparing arrays and the elements of two similar shaped arrays
- Logical operators: AND/OR operands

Comparison Operators

```
In [93]: A = np.array([[11, 12, 13], [21, 22, 23], [31, 32, 33]])
B = np.array([[11, 102, 13], [201, 22, 203], [31, 32, 303]])
```

```
# It will compare all the elements of the array with each other
print (A == B)
```

```
[[ True False  True]
 [False  True False]
 [ True  True False]]
```

```
In [94]: # Will return 'True' only if each and every element is same in both the arrays
print(np.array_equal(A, B))
```

```
False
```

Logical Operators

```
In [98]: # Boolean Arrays
```

```
a = np.array([[True, True], [False, False]])
b = np.array([[True, False], [True, False]])
```

```
print(a)
print("-----")
print(b)
```

```
[[ True  True]
 [False False]]
-----
[[ True False]
 [ True False]]
```

```
In [99]: #minimum - either should be True? then output will be --> True
print(np.logical_or(a, b))
```

```
[[ True  True]
 [ True False]]
```

```
In [102]: #Both should be True --> then output will be --> True
print(np.logical_and(a, b))
```

```
[[ True False]
 [False False]]
```

Using Axis defines direction of operation.

axis = 0 → go down → operate column-wise

axis = 1 → go right → operate row-wise

```
In [103]: print(A)
```

```
[[11 12 13]
 [21 22 23]
 [31 32 33]]
```

```
In [104]: x = A.sum(axis = 0)
print(x)
```

```
[63 66 69]
```

```
In [105]: y = A.sum(axis = 1)
print(y)
```

```
[36 66 96]
```

```
In [ ]:
```

```
In [ ]:
```

```
# TTP Foundation 7
```

NumPy Function to create arrays quickly

Arange method np.arange()

- **Purpose:** Creates evenly spaced values within a given interval.
- **Parameters:**
 - start → beginning of interval (default = 0 if omitted).
 - stop → end of interval (exclusive).
 - step → spacing between values (default = 1).
- **Returns:** A 1D NumPy array of numbers.

```
In [99]: import numpy as np
x = np.arange(12) #similar to range that we used in our earlier classes
print(x)
```

[0 1 2 3 4 5 6 7 8 9 10 11]

```
In [100]: y = np.arange(1, 10)
print(y)
```

[1 2 3 4 5 6 7 8 9]

```
In [101]: k = np.arange(1.4, 15.5, step=2)
```

```
In [102]: print(k)
```

[1.4 3.4 5.4 7.4 9.4 11.4 13.4 15.4]

np.reshape()

reshape only rearranges, it never adds or removes elements.

New dimensions must multiply to the same total count as the old array.

```
In [103]: z = np.arange(40)
```

```
In [105]: print(z)
```

[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39]

In [106]: `z.reshape(10, 4)`

Out[106]: `array([[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11],
 [12, 13, 14, 15],
 [16, 17, 18, 19],
 [20, 21, 22, 23],
 [24, 25, 26, 27],
 [28, 29, 30, 31],
 [32, 33, 34, 35],
 [36, 37, 38, 39]])`

`np.linspace()`

Numpy.linspace also returns an evenly spaced array but needs the 'number of array elements' as an input from the user and creates the distance automatically.

Syntax:

`linspace(start, stop, num=50, endpoint=True, retstep=False)`

The 'start' and the 'stop' determines the range of the array. 'num' determines the number of elements in the array. By default the 'endpoint' is True, it will include the stop value and if it is false, the array will exclude the stop value.

If the optional parameter '`retstep`' is set to True, the function will return the value of the spacing between adjacent values.

By default, since the 'num' is not given, it will divide the range into 50 individual array elements

By default, it even includes the 'endpoint' of the range, since it is set to True by default

In [107]: `lin_1 = np.linspace(1, 10, num = 40, retstep=False)
print(lin_1)`

```
[ 1.          1.23076923  1.46153846  1.69230769  1.92307692  2.15384615  
 2.38461538  2.61538462  2.84615385  3.07692308  3.30769231  3.53846154  
 3.76923077  4.          4.23076923  4.46153846  4.69230769  4.92307692  
 5.15384615  5.38461538  5.61538462  5.84615385  6.07692308  6.30769231  
 6.53846154  6.76923077  7.          7.23076923  7.46153846  7.69230769  
 7.92307692  8.15384615  8.38461538  8.61538462  8.84615385  9.07692308  
 9.30769231  9.53846154  9.76923077 10.         ]]
```

Lets arrange numbers 1 - 8 to be divided into 20 individual array elements

```
In [108]: lin_2 = np.linspace(1, 8, 20, retstep=True)
print(lin_2)

(array([1.          , 1.36842105, 1.73684211, 2.10526316, 2.47368421,
       2.84210526, 3.21052632, 3.57894737, 3.94736842, 4.31578947,
       4.68421053, 5.05263158, 5.42105263, 5.78947368, 6.15789474,
       6.52631579, 6.89473684, 7.26315789, 7.63157895, 8.          ]),
     0.3684210526315789)
```

```
In [109]: # Simple Example

e = np.linspace(1, 10, 10, True, True)
print(e)

(array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.]), 1.0)
```

Create Array of ones and zeros using np.ones() and np.zeros()

```
In [110]: z = np.ones((12, 3))
```

```
In [111]: print(z)

[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

```
In [112]: m = np.zeros((6, 6), dtype = int)
```

```
print(m)

[[0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]
```

np.full()

```
In [113]: km = np.full((12,3), 5)
print(km)
```

```
[[5 5 5]
 [5 5 5]
 [5 5 5]
 [5 5 5]
 [5 5 5]
 [5 5 5]
 [5 5 5]
 [5 5 5]
 [5 5 5]
 [5 5 5]
 [5 5 5]
 [5 5 5]]
```

Identity function np.identity()

An identity array has equal number of rows and columns. It is a square array so that the diagonal elements are all 'ones'.

```
In [114]: W = np.identity(10, dtype = int)
print(W)
```

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

NumPy Random Number Generators

- **np.random.seed()** → Set the seed for reproducibility.
Example: `np.random.seed(42)`
- **np.random.rand(n)** → Uniform random numbers between 0 and 1.
Example: `np.random.rand(5) → [0.37, 0.95, 0.73, ...]`
- **np.random.randn(n)** → Standard normal distribution (mean=0, std=1).
Example: `np.random.randn(5) → [-0.14, 0.65, 1.52, ...]`
- **np.random.randint(low, high, size)** → Random integers in a range.
Example: `np.random.randint(95, 105, size=10) → open prices`
- **np.random.uniform(low, high, size)** → Uniform distribution over [low, high).
Example: `np.random.uniform(90, 110, size=10) → close prices`
- **np.random.normal(loc, scale, size)** → Normal distribution with mean (loc) and std (scale).
Example: `np.random.normal(loc=0.001, scale=0.02, size=10) → daily returns`

In [115]: `np.random.seed(100)`

In [116]: `x = np.random.rand(10)`
`print(x)`
`#x.mean() #Actual mean will be close to 0`

```
[0.54340494 0.27836939 0.42451759 0.84477613 0.00471886 0.12156912
 0.67074908 0.82585276 0.13670659 0.57509333]
```

In [117]: `x.std() #Actual Standard Deviation will be close to 1`

Out[117]: 0.2835606816389971

In [118]: `#The theoretical mean (expected value) of a uniform distribution is: low + h`
`#so in this case 90+120/2 = 105 (so close to this)`
`y = np.random.uniform(90, 120, size=10000)`

In [119]: `print(y)`

```
[116.73965863 96.27606366 95.55984659 ... 103.29872402 93.4750451
 94.97813708]
```

In [120]: `y.mean()`

Out[120]: 104.93292861268512

In [121]: `# Set seed for reproducibility`
`np.random.seed(50)`

`#Generate 20 numbers where average daily return = 0.1 percent, volatility = 2`
`daily_returns = np.random.normal(loc=0.001, scale=0.05, size=10000)`
`print(daily_returns)`

```
[-0.07701761 -0.00054888 -0.03004642 ... 0.01930034 0.0530678
 -0.0042766 ]
```

In [122]: `daily_returns.mean()`

Out[122]: 0.0017439574903867878

In [123]: `# Set seed for reproducibility`
`np.random.seed(50)`

`# Generate synthetic open and close prices for 10 days`
`open_price = np.random.randint(95, 105, size=10) # open prices between 95-`
`close_price = np.random.randint(90, 110, size=10) # close prices between 90`

`print("Open Prices:", open_price)`
`print("Close Prices:", close_price)`

```
Open Prices: [ 95  95  96  99 101 100 101 101 100  97]
Close Prices: [ 97 105  94 104  93  96 101 107 100  99]
```

NumPy Statistical Functions

- `np.mean(open_price)` → Average of all open prices.
Example: $(103+97+100+95+104+96+99+102+98+101)/10 = 99.5$
- `np.std(open_price)` → Spread/volatility of open prices.
Example: ≈ 2.87 , showing prices vary ~3 units around the mean.
- `np.var(open_price)` → Variance of open prices (square of std).
Example: ≈ 8.25 , representing the average squared deviation from the mean.
- `np.median(open_price)` → Middle value when open prices are sorted.
Example: 99.5 (average of 99 and 100 in sorted list).
- `np.max(close_price)` → Highest closing price.
Example: 109
- `np.min(close_price)` → Lowest closing price.
Example: 90
- `np.argmax(close_price)` → Index of the highest closing price.
Example: 0 (first day had the max close of 109).
- `np.argmin(close_price)` → Index of the lowest closing price.
Example: 4 (fifth day had the min close of 90).

```
In [124]: # Mean of open prices
mean_open = np.mean(open_price)

# Standard deviation of open prices
std_open = np.round(np.std(open_price),2)
#std_open = np.std(open_price)

# Maximum close price
max_close = np.max(close_price)

# Minimum close price
min_close = np.min(close_price)

# Index of maximum close price
argmax_close = np.argmax(close_price)

# Index of minimum close price
argmin_close = np.argmin(close_price)

print("Mean of Open Prices:", mean_open)
print("Std Dev of Open Prices:", std_open)
print("Max of Close Prices:", max_close)
print("Min of Close Prices:", min_close)
print("Index of Max Close Price:", argmax_close)
print("Index of Min Close Price:", argmin_close)

Mean of Open Prices: 98.5
Std Dev of Open Prices: 2.38
Max of Close Prices: 107
Min of Close Prices: 93
Index of Max Close Price: 7
Index of Min Close Price: 4
```

 [NumPy Official Documentation: NumPy Reference](https://numpy.org/doc/stable/reference/)
[\(https://numpy.org/doc/stable/reference/\)](https://numpy.org/doc/stable/reference/)

You can refer to the above official link for other NumPy functions/routines.

Boolean masking (Filtering without loops)

```
In [125]: prices = np.array([100, 102, 98, 105, 97])
mask = prices > 200
print("Mask:", mask)
print("Filtered prices:", prices[mask])
```

```
Mask: [False False False False False]
Filtered prices: []
```

Vectorized Signal Example

`np.where()`

`np.where()` is a NumPy function used to apply conditions and return results based on those conditions. It lets you apply conditions on entire arrays at once.

Think of it like:

Python's `if...else`, but vectorized for entire arrays.

It checks each element and chooses one of the given outputs.

Basic Syntax

```
np.where(condition, value_if_true, value_if_false)
```

```
In [126]: price = np.array([95, 102, 88, 110])

signal = np.where(price > 100, "BUY", "IGNORE")
print(signal)

['IGNORE' 'BUY' 'IGNORE' 'BUY']
```

```
In [127]: new_price = np.arange(10, 50)
print(new_price)

[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49]
```

```
In [128]: signal1 = np.where(new_price > 30, "BUY", "SELL")
print(signal1)

['SELL' 'SELL' 'SELL' 'SELL' 'SELL' 'SELL' 'SELL' 'SELL' 'SELL'
 'SELL' 'SELL' 'SELL' 'SELL' 'SELL' 'SELL' 'SELL' 'SELL' 'SELL'
 'SELL' 'BUY' 'BUY' 'BUY' 'BUY' 'BUY' 'BUY' 'BUY' 'BUY' 'BUY'
 'BUY' 'BUY' 'BUY' 'BUY' 'BUY' 'BUY' 'BUY' 'BUY' 'BUY']
```

```
In [129]: arr = np.array([1, 0, 3, 0, 5])
new_arr = np.where(arr == 0, np.nan, arr)
print(new_arr)

[ 1. nan  3. nan  5.]
```

```
In [130]: print(price)

[ 95 102  88 110]
```

```
In [131]: idx = np.where(price > 101) #will fetch the array location
print(idx)

(array([1, 3], dtype=int64),)
```

To get the actual values, we must subset the original array

```
In [132]: filtered = price[np.where(price > 101)]
print(filtered)

[102 110]
```

Signal Generation

```
In [133]: open_price = np.array([100, 102, 100])
close_price = np.array([101, 101, 100])

candle = np.where(close_price > open_price, "Bullish",
                  np.where(close_price < open_price, "Bearish",
                           "Doji"))

print(candle)

['Bullish' 'Bearish' 'Doji']
```

```
In [134]: print(open_price)

[100 102 100]
```

```
In [135]: print(close_price)

[101 101 100]
```

```
In [136]: #np.where() returns a tuple, so we extract the first element using [0]
index_max_close = np.where(close_price == np.max(close_price))[0]
print("All index's of max open:", index_max_close)

All index's of max open: [0 1]
```

```
In [137]: index_max_close = np.where(close_price == np.max(close_price))
print("All index's of max open:", index_max_close)

All index's of max open: (array([0, 1], dtype=int64),)
```

```
In [138]: x = np.array([2, 3, 1, 7, 2, 8, 5])
```

```
In [139]: np.where((x<6) & (x>3), 'buy', 'sell')
```

```
Out[139]: array(['sell', 'sell', 'sell', 'sell', 'sell', 'sell', 'buy'], dtype='|U4')
```

```
In [140]: print(x[np.where(x>3)])
```

```
[7 8 5]
```

NumPy provides two powerful functions for cumulative calculations

`np.cumsum()`

`np.cumsum()` adds numbers one by one and keeps a running total (cumulative sum).

`np.cumprod()`

multiplies numbers one by one and keeps the running product.

`np.cumprod()` multiplies numbers one by one and keeps the running product (cumulative product).

```
In [141]: arr = np.array([1, 2, 3, 4])
result = np.cumsum(arr)
print(result)
```

```
[ 1  3  6 10]
```

```
In [142]: arr1 = np.array([2, 3, 4, 5, 6, 7])
result1 = np.cumprod(arr1)
print(result1)
```

```
[ 2    6   24  120  720 5040]
```

Generating signals for multiple conditions strategy using `np.where()`

```
In [143]: np.random.seed(10)
```

```
price = np.linspace(100, 115, 12, retstep=True)
ma_short = np.linspace(100, 112, 12)
ma_long = np.linspace(100, 110, 12)
rsi = np.linspace(25, 40, 12)
```

```
In [144]: print(price)
print("-----")
print(ma_short)
print("-----")
print(ma_long)
print("-----")
print(rsi)

(array([100.          , 101.36363636, 102.72727273, 104.09090909,
       105.45454545, 106.81818182, 108.18181818, 109.54545455,
       110.90909091, 112.27272727, 113.63636364, 115.          ],
      ), 1.363636363635)
-----
[100.          101.09090909 102.18181818 103.27272727 104.36363636
 105.45454545 106.54545455 107.63636364 108.72727273 109.81818182
 110.90909091 112.          ]
-----
[100.          100.90909091 101.81818182 102.72727273 103.63636364
 104.54545455 105.45454545 106.36363636 107.27272727 108.18181818
 109.09090909 110.          ]
-----
[25.          26.36363636 27.72727273 29.09090909 30.45454545 31.81818182
 33.18181818 34.54545455 35.90909091 37.27272727 38.63636364 40.          ]
```

```
In [145]: signals = np.where(
    (ma_short > ma_long) & (rsi < 30), "BUY",
    np.where((ma_short < ma_long) & (rsi > 70), "SELL", "HOLD"))
```

```
In [147]: print(signals)

['HOLD' 'BUY' 'BUY' 'BUY' 'HOLD' 'HOLD' 'HOLD' 'HOLD' 'HOLD' 'HOLD'
 'HOLD']
```

Exercise 1: Create an array from 0 to 20 using step of 2.

```
In [148]: arr1 = np.arange(0, 21, 2)
print(arr1)
```

```
[ 0  2  4  6  8 10 12 14 16 18 20]
```

Exercise 2: Create an array of numbers from 10 to 1 (reverse order).

```
In [149]: arr2 = np.arange(10, 0, -1)
print(arr2)
```

```
[10  9  8  7  6  5  4  3  2  1]
```

Exercise 3: Generate 5 numbers between 0 and 1 (both inclusive).

```
In [150]: arr3 = np.linspace(0, 1, 5, retstep=True)
print(arr3)
```

```
(array([0.  , 0.25, 0.5 , 0.75, 1.  ]), 0.25)
```

Exercise 4: Generate 10 evenly spaced numbers between 50 and 100.

```
In [158]: arr5 = np.linspace(50, 100, 10)
print(arr5)

[ 50.          55.55555556  61.11111111  66.66666667  72.22222222
 77.77777778  83.33333333  88.88888889  94.44444444 100.        ]
```

Exercise 5: Create a 3×3 identity matrix.

```
In [159]: arr10 = np.identity(5)
print(arr10)

[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

```
In [151]: arr4 = np.identity(10)
print(arr4)

[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

Exercise 6: Create a 5×5 identity matrix and print the diagonal values only.

```
In [160]: I5 = np.identity(5)
print("Identity Matrix:\n", I5)

# Print diagonal values
print("Diagonal Values:", I5.diagonal())

Identity Matrix:
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
Diagonal Values: [1. 1. 1. 1. 1.]
```

Exercise 7: Generate 5 random integers between 1 and 50.

```
In [161]: arr15 = np.random.randint(1, 51, size=5)
print(arr15)

[43 41 37 17 37]
```

Exercise 8: Generate a 3×3 matrix of random integers between 10 and 99.

In [152]: `xyz = np.random.randint(10, 100, size=(3,3))
print(xyz)`

```
[[19 25 74]
 [38 99 39]
 [18 83 10]]
```

Exercise 9: Given array `a = np.array([3, 8, 1, 7, 9])`, find the index of the maximum value.

In [162]: `a_max = np.array([3, 8, 1, 7, 9])
print(np.argmax(a_max))`

```
4
```

Exercise 10: Given array `b = np.array([10, 4, 6, 2, 15])`, find the index of the minimum value.

In [163]: `a_min = np.array([3, 8, 1, 7, 9])
print(np.argmin(a_min))`

```
2
```

In [164]: `abc = np.arange(30)`

In [165]: `print(abc)`

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29]
```

In [166]: `abc.reshape(3, 10)`

Out[166]: `array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
 [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])`

In [167]: `abc.reshape(10, 3)`

Out[167]: `array([[0, 1, 2],
 [3, 4, 5],
 [6, 7, 8],
 [9, 10, 11],
 [12, 13, 14],
 [15, 16, 17],
 [18, 19, 20],
 [21, 22, 23],
 [24, 25, 26],
 [27, 28, 29]])`

In [168]: `print(abc)`

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29]
```

In []:

TTP Assignment 7

Functions

Question #1 - Candle Type Identification

Using the lists below:

```
open_price = [220, 225, 223, 230, 228, 232, 235, 240]
```

```
close_price = [224, 220, 226, 231, 227, 236, 233, 242]
```

Write a function `candle_type(open_p, close_p)` that returns:

"Bullish" / "Bearish" / "Doji"

Loop using `zip()` and print the full list of candle types.

```
In [6]: open_price = [220, 225, 223, 230, 228, 232, 235, 240]
close_price = [224, 220, 226, 231, 227, 236, 233, 242]
def candle_type(open_p, close_p):
    if open_p > close_p:
        return "Bullish"
    elif open_p < close_p:
        return "Bearish"
    else:
        return "Doji"

candle_list = []

for x, y in zip(open_price, close_price):
    candle_list.append(candle_type(x, y))

print(candle_list)
```

```
['Bearish', 'Bullish', 'Bearish', 'Bearish', 'Bullish', 'Bearish', 'Bullish', 'Bearish']
```

Question #2 - Percentage Change List

Given:

```
prices = [150, 152, 149, 155, 160, 158, 162, 165]
```

Write a function `pct_change(listname)` that returns a list of percentage changes.

First value should be `None`.

```
In [5]: prices = [150, 152, 149, 155, 160, 158, 162, 165]

def pct_change(price_list):
    pct_values = []

    for i in range(len(price_list)):
        if i == 0:
            pct_values.append(None)
        else:
            prev = price_list[i-1]
            curr = price_list[i]
            pct = ((curr - prev) / prev) * 100
            pct_values.append(round(pct, 2))

    return pct_values

pct_vals = pct_change(prices)
print(pct_vals)
```

[None, 1.33, -1.97, 4.03, 3.23, -1.25, 2.53, 1.85]

Question #3 - OHLC Range & Mid Price

Given:

highs = [305, 310, 308, 315, 320]

lows = [300, 305, 304, 310, 315]

Write a function **range_mid(high, low)** that returns:

- Range (high - low)
- Mid price $((\text{high} + \text{low}) / 2)$

Store all results in two lists: **range_list** and **mid_list**.

```
In [12]: highs = [305, 310, 308, 315, 320]
lows = [300, 305, 304, 310, 315]

def range_mid(high, low):
    range_list = high - low
    mid_list = (high + low) / 2
    return range_list, mid_list

range_list = []
mid_list = []

for h, l in zip(highs, lows):
    r, m = range_mid(h, l)
    range_list.append(r)
    mid_list.append(m)

print("Range List:", range_list)
print("Mid List:", mid_list)
```

Range List: [5, 5, 4, 5, 5]
 Mid List: [302.5, 307.5, 306.0, 312.5, 317.5]

Question #4 - Filter High Volume

Given a list of volumes:

```
volumes = [1200, 300, 4500, 8000, 1500, 9000, 200]
```

Use **filter()** + **lambda** to extract all values greater than 5000

```
In [13]: volumes = [1200, 300, 4500, 8000, 1500, 9000, 200]
          list(filter(lambda x:x > 5000, volumes))
```

```
Out[13]: [8000, 9000]
```

Question #5 - Generate Buy/Sell Signals

Given:

```
close_list = [105, 102, 108, 110, 107, 112]
```

```
sma_list = [103, 104, 105, 109, 108, 111]
```

Write a function **generate_signal(close, sma)** that returns:

"Buy" if close > sma, otherwise "Sell".

Loop and print all signals.

```
In [14]: close_list = [105, 102, 108, 110, 107, 112]
          sma_list = [103, 104, 105, 109, 108, 111]

def generate_signal(close,sma):
    if close > sma :
        return "Buy"
    else:
        return "Sell"

signals = []

for close, sma in zip(close_list, sma_list):
    signals.append(generate_signal(close, sma))

print(signals)
```

```
['Buy', 'Sell', 'Buy', 'Buy', 'Sell', 'Buy']
```

Question #6 - Calculate Spreads Using map()

Given:

```
bid = [100, 102, 101, 105]
```

```
ask = [101, 103, 102, 106]
```

Use **map()** + **lambda** to compute the spread:

ask - bid

Return the list of spreads.

```
In [15]: bid = [100, 102, 101, 105]
ask = [101, 103, 102, 106]

spreads = list(map(lambda b, a: a - b, bid, ask))

print(spreads)
```

[1, 1, 1, 1]

Question #7 - Sort a List of Tuples

Given stock data:

```
data = [('AAPL', 235.10), ('MSFT', 402.30), ('NVDA', 950.12), ('TSLA', 185.40)]
```

Sort the list based on the price (2nd value) using:

sorted() and **lambda**.

```
In [16]: data = [('AAPL', 235.10), ('MSFT', 402.30), ('NVDA', 950.12), ('TSLA', 185.40)]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)

[('TSLA', 185.4), ('AAPL', 235.1), ('MSFT', 402.3), ('NVDA', 950.12)]
```

Question #8 - SL & TP Function (Default Parameters)

Write a function:

```
def sl_tp(price, sl=0.03, tp=0.06):
```

Return the SL and TP levels.

Given:

```
entry_prices = [1000, 1020, 980, 1100]
```

Loop and print SL & TP for each price.

```
In [17]: def sl_tp(price, sl=0.03, tp=0.06):
    stop_loss = price * (1 - sl)
    take_profit = price * (1 + tp)
    return stop_loss, take_profit

entry_prices = [1000, 1020, 980, 1100]

for price in entry_prices:
    sl, tp = sl_tp(price)
    print(f"Entry: {price} -> SL: {sl:.2f}, TP: {tp:.2f}")
```

Entry: 1000 -> SL: 970.00, TP: 1060.00
 Entry: 1020 -> SL: 989.40, TP: 1081.20
 Entry: 980 -> SL: 950.60, TP: 1038.80
 Entry: 1100 -> SL: 1067.00, TP: 1166.00

Question #9 - Trend Checker Using zip()

Given:

```
day1 = [140, 141, 143, 142, 145]
```

```
day2 = [142, 140, 144, 146, 147]
```

Write a function **trend(a, b)** that returns:

"Uptrend" if b > a, else "Downtrend".

Use `zip()` to print the full trend list.

```
In [18]: day1 = [140, 141, 143, 142, 145]
day2 = [142, 140, 144, 146, 147]

def trend(a, b):
    if b > a:
        return "Uptrend"
    else:
        return "Downtrend"

trend_list = []

for d1, d2 in zip(day1, day2):
    trend_list.append(trend(d1, d2))

print(trend_list)
```

```
[ 'Uptrend', 'Downtrend', 'Uptrend', 'Uptrend', 'Uptrend' ]
```

Question #10 - Count Buy & Sell Using filter()

Given:

```
signals = ["Buy", "Sell", "Buy", "Buy", "Sell", "Sell", "Buy"]
```

Use **filter()** to:

- Extract only all "Buy"
- Extract only all "Sell"

Print how many Buy and Sell signals are present.

```
In [19]: signals = ["Buy", "Sell", "Buy", "Buy", "Sell", "Sell", "Buy"]

buy_signals = list(filter(lambda s: s == "Buy", signals))
sell_signals = list(filter(lambda s: s == "Sell", signals))

print("Buy Signals:", buy_signals)
print("Sell Signals:", sell_signals)
print("Total Buy:", len(buy_signals))
print("Total Sell:", len(sell_signals))
```

```
Buy Signals: ['Buy', 'Buy', 'Buy', 'Buy']
```

```
Sell Signals: ['Sell', 'Sell', 'Sell']
```

```
Total Buy: 4
```

```
Total Sell: 3
```

In []:

TTP Assignment 8

NumPy Arrays

Question #1A - Print Array Shape

Using the array **A** below:

```
A =  
[[10, 20, 30, 40, 50],  
 [15, 25, 35, 45, 55],  
 [12, 22, 32, 42, 52],  
 [17, 27, 37, 47, 57],  
 [19, 29, 39, 49, 59]]
```

Print the **shape** of the array.

```
In [1]: import numpy as np
```

```
In [2]: A = np.array([[10, 20, 30, 40, 50],  
 [15, 25, 35, 45, 55],  
 [12, 22, 32, 42, 52],  
 [17, 27, 37, 47, 57],  
 [19, 29, 39, 49, 59]]  
  
print(A)  
print("Shape:", A.shape)
```

```
[[10 20 30 40 50]  
 [15 25 35 45 55]  
 [12 22 32 42 52]  
 [17 27 37 47 57]  
 [19 29 39 49 59]]  
Shape: (5, 5)
```

Hint: import numpy as np before creating the array

Question #1B - Print Number of Dimensions

Using the same array **A**, print the **ndim** (number of dimensions).

```
In [3]: print("Dimensions:", A.ndim)
```

```
Dimensions: 2
```

Question #1C - Print Data Type

Using array **A**, print the **dtype**.

```
In [4]: print(type(A))
```

```
<class 'numpy.ndarray'>
```

Question #1D - Print Total Elements

Using array **A**, print the **size** (total number of elements).

```
In [5]: print("Size:", A.size)
```

```
Size: 25
```

Question #2 - Index 2nd and 4th Rows

From array **A**, index rows **1** and **3** (2nd and 4th rows).

```
In [6]: A[[1,3], :]
```

```
Out[6]: array([[15, 25, 35, 45, 55],
 [17, 27, 37, 47, 57]])
```

Question #3 - Index the First Three Rows

From array **A**, extract rows **0, 1, 2**.

```
In [7]: A[[0,1,2], :]
```

```
Out[7]: array([[10, 20, 30, 40, 50],
 [15, 25, 35, 45, 55],
 [12, 22, 32, 42, 52]])
```

Question #4 - Index the Last Two Rows

From array **A**, extract rows **3** and **4**.

```
In [8]: A[-2:, :]
```

```
Out[8]: array([[17, 27, 37, 47, 57],
 [19, 29, 39, 49, 59]])
```

Question #5 - Index the First Column Only

From array **A**, extract column **0** only.

```
In [9]: A[:, 0]
```

```
Out[9]: array([10, 15, 12, 17, 19])
```

Question #6 - Index the Last Column Only

From array **A**, extract column **4** only

In [10]: `A[:, -1]`

Out[10]: `array([50, 55, 52, 57, 59])`

Question #7 - Index the Middle Row (Row 3)

From array **A**, extract row **2** (middle row).

In [11]: `A[2, :]`

Out[11]: `array([12, 22, 32, 42, 52])`

Question #8 - Index the Middle Column (Column 3)

From array **A**, extract column **2** (middle column).

In [12]: `A[2, :]`

Out[12]: `array([12, 22, 32, 42, 52])`

Question #9 - Top-Left 3x3 Block

From array **A**, extract the block of rows **0 to 2** and columns **0 to 2**.

In [13]: `A[:3, :3]`

Out[13]: `array([[10, 20, 30],
[15, 25, 35],
[12, 22, 32]])`

Question #10 - Index Every Alternate Row

From array **A**, extract rows **0, 2, 4** using step slicing.

Array Operations Section (New Dataset)

Dataset **B**:

`[[5, 8, 12],
[3, 15, 20],
[7, 10, 25]]`

Question #11 - Add 10 to Every Element

Using array **B**, add **10** to every element and print the updated array.

```
In [14]: B = [[5, 8, 12],[3, 15, 20],[7, 10, 25]]  
arrB = np.array(B)  
  
arrB_add = arrB + 10  
print(arrB_add)
```

```
[[15 18 22]  
 [13 25 30]  
 [17 20 35]]
```

Question #12 - Multiply All Elements by 2

Using array **B**, multiply every element by **2** and print the result.

```
In [15]: BC = arrB_add * 2  
print(BC)
```

```
[[30 36 44]  
 [26 50 60]  
 [34 40 70]]
```

Question #13 - Column-Wise Sum

Using array **B**, compute the **sum of each column**.

```
In [23]: B = arrB.sum(axis = 0)  
print(B)
```

```
[15 33 57]
```

Question #14 - Row-Wise Maximum

Using array **B**, print the **maximum value from each row**.

```
In [24]: B = arrB.sum(axis = 1)  
print(B)
```

```
[25 38 42]
```

Question #15 - Comparison: Elements Greater Than 10

Using array **B**, check which elements are **greater than 10** and print the Boolean output.

```
In [25]: mask = arrB > 10  
print("Mask:", mask)  
print("Filtered prices:", arrB[mask])
```

```
Mask: [[False False  True]  
 [False  True  True]  
 [False False  True]]  
Filtered prices: [12 15 20 25]
```

```
# TTP Foundation 8
```

Matplotlib

`matplotlib` is a popular charting library used to produce publication-quality graphs and figures. We'll primarily use it as a visualization tool. Visualization is an important step in data analysis. It helps you get a 'feel' for your dataset and draw inferences that would otherwise not be possible. It is usually done at a preliminary stage. Matplotlib can be used with Python Lists, NumPy Arrays, Pandas Series & Dataframes

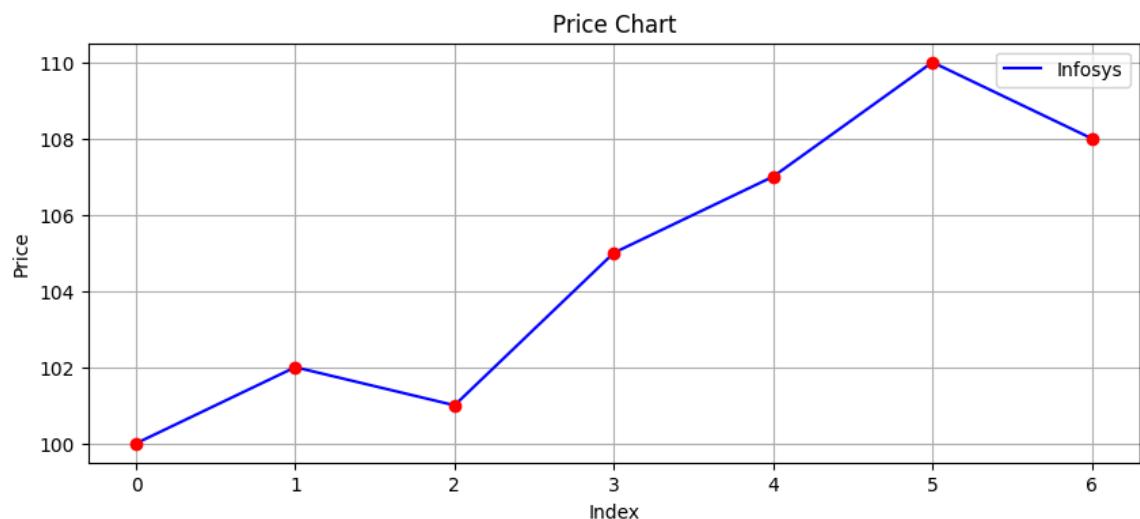
□ **Matplotlib Official Documentation:** [Matplotlib Reference](https://matplotlib.org/stable/users/index.html)
[\(https://matplotlib.org/stable/users/index.html\)](https://matplotlib.org/stable/users/index.html).

You can refer to the above official link for other Matplotlib functions/routines.

```
In [1]: import numpy as np  
import matplotlib.pyplot as plt
```

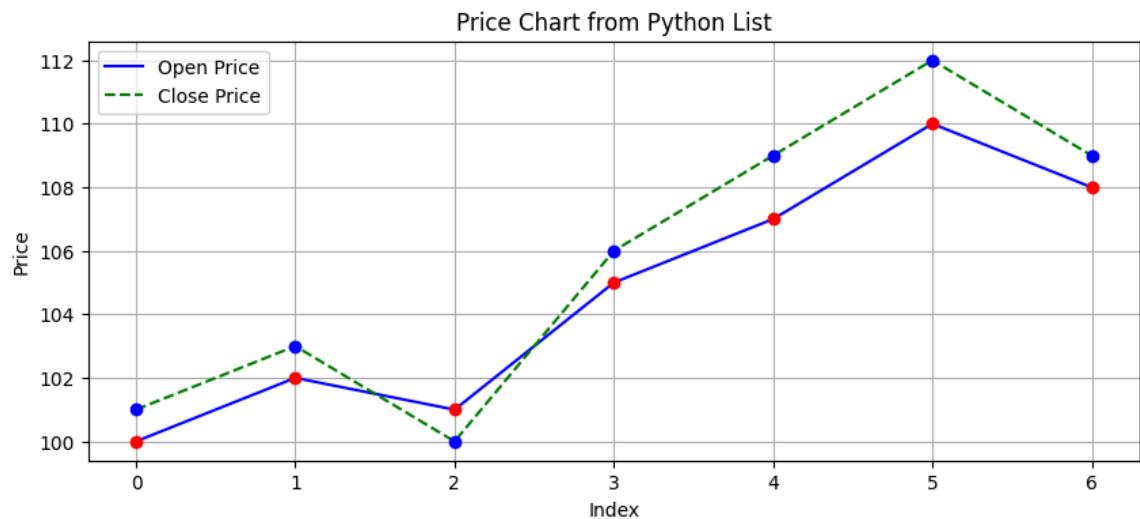
Plotting Line Charts

```
In [5]: price = [100, 102, 101, 105, 107, 110, 108]  
  
plt.figure(figsize=(10,4))  
plt.plot(price, 'b', label="Infosys")      # 'b' for line color  
plt.plot(price, 'ro')                      # red dot marker  
plt.grid(True) # Make the grid visible  
plt.title("Price Chart")  
plt.xlabel("Index")  
plt.ylabel("Price")  
plt.legend()  
plt.show()
```



```
In [7]: open_price = [100, 102, 101, 105, 107, 110, 108]
close_price = [101, 103, 100, 106, 109, 112, 109]

plt.figure(figsize=(10,4))
plt.plot(open_price, 'b', label="Open Price")      # 'b' for line color
plt.plot(open_price, 'ro')                         # 'ro' for red dot marker
plt.plot(close_price, '--g', label="Close Price")   # '--' for dashed line
plt.plot(close_price, 'bo')
plt.grid(True) # Make the grid visible
plt.title("Price Chart from Python List")
plt.xlabel("Index")
plt.ylabel("Price")
plt.legend()
plt.show()
```



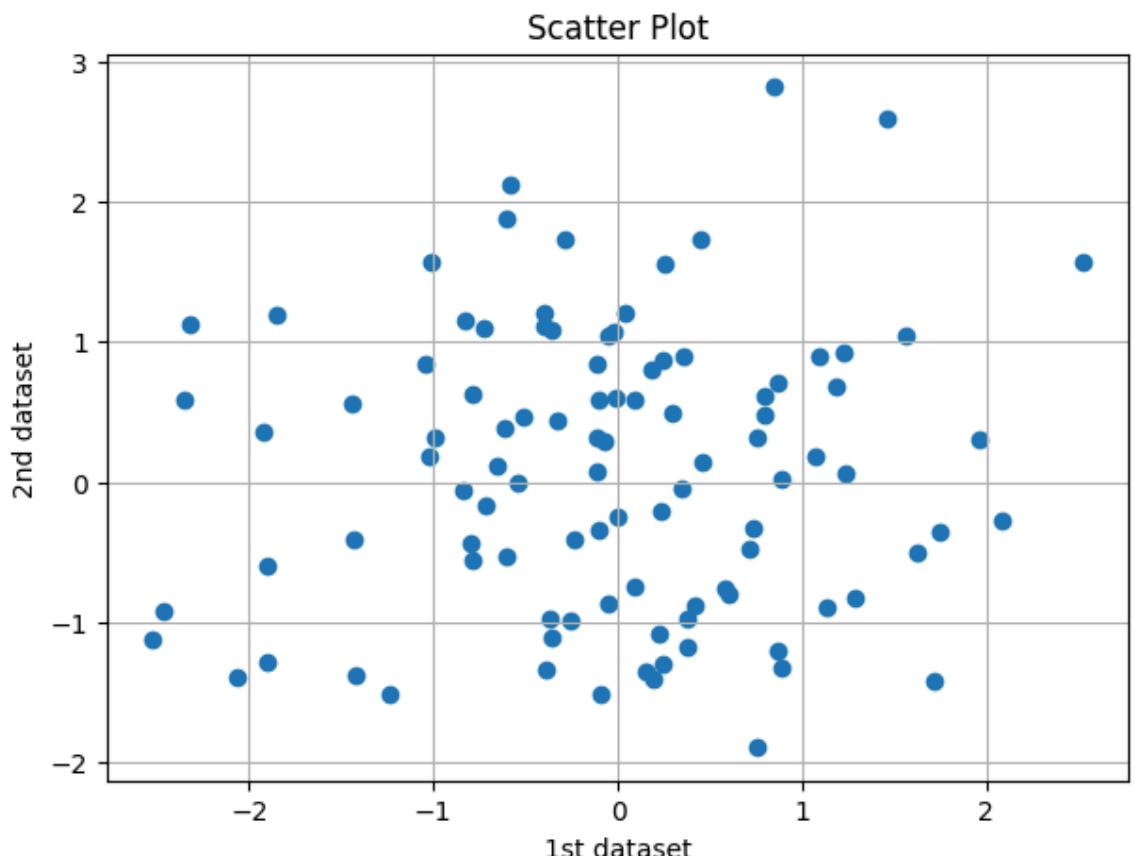
Plotting Scatter Plots

```
In [6]: y = np.random.standard_normal((100, 2)) # np.random accesses numpy's random
#Generates random numbers that follow a standard normal distribution, i.e. m
#The shape of the array here - it means 100 rows and 2 columns.

plt.figure(figsize=(7, 5))

# The function 'scatter' is called to our 'plt' object
plt.scatter(y[:, 0], y[:, 1], marker='o')
# All rows of column 0 vs all rows of column 1 for matching points.
# This is typical when plotting 2D data where each row is a coordinate pair

plt.grid(True)
plt.xlabel('1st dataset')
plt.ylabel('2nd dataset')
plt.title('Scatter Plot')
plt.show()
```



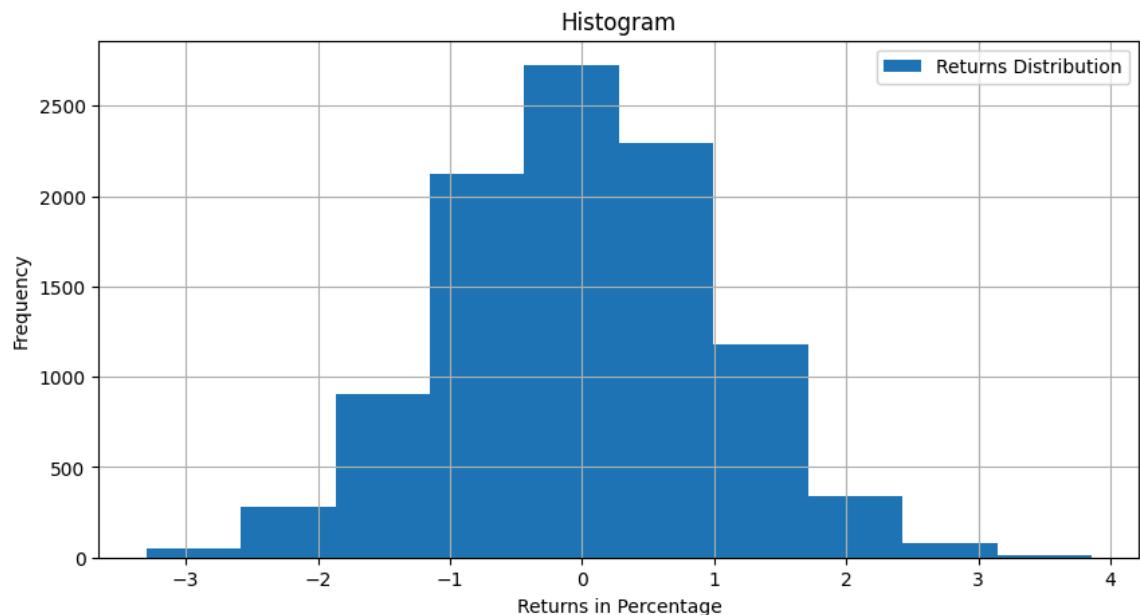
Plotting Histogram

```
In [15]: # Random data created
# When you use random functions, results change every time you run the program
# Setting a seed value ensures you always get the same random sequence
np.random.seed(100)
y = np.random.standard_normal(size=10000)

plt.figure(figsize=(10, 5))

# The function 'hist' is called to our 'plt' object
plt.hist(y, label=['Returns Distribution'])

plt.grid(True)
plt.legend(loc=0)
plt.ylabel('Frequency')
plt.xlabel('Returns in Percentage')
plt.title('Histogram')
plt.show()
```



Plotting Bar Charts

```
In [10]: import matplotlib.pyplot as plt

# Synthetic monthly closing prices of a stock
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
close_price = [150, 158, 155, 165, 160, 170]

# Convert to monthly returns (%)
monthly_return = []

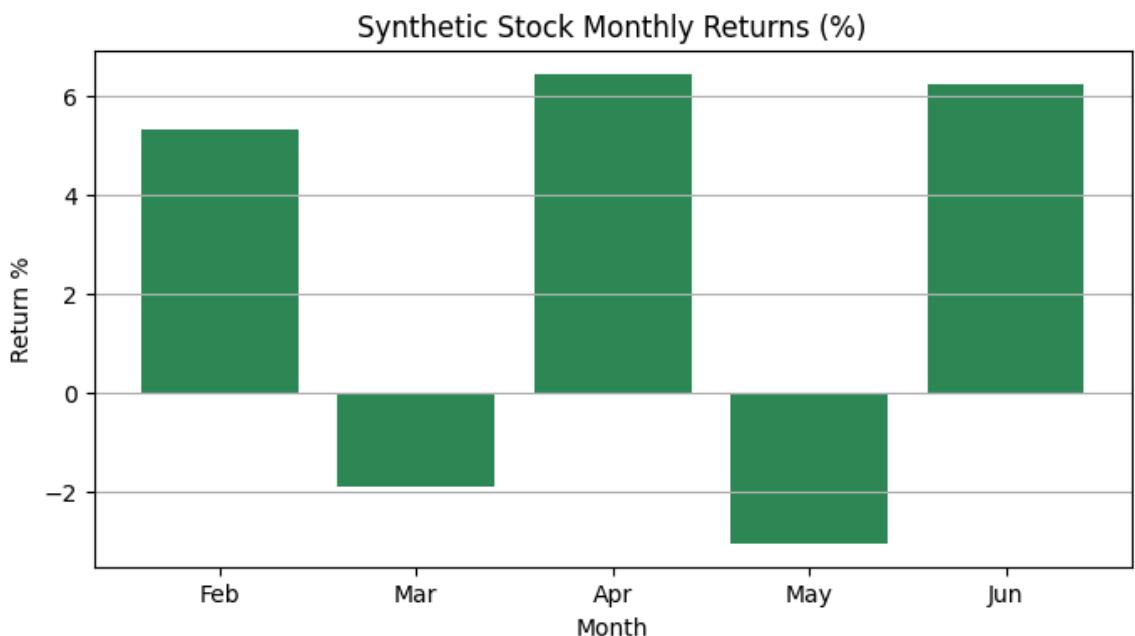
for i in range(1, len(close_price)):
    ret = ((close_price[i] - close_price[i-1]) / close_price[i-1]) * 100
    monthly_return.append(round(ret, 2))

# Bars will be from Feb to Jun (because Jan has no return)
plot_months = months[1:]

# Plot bar chart
plt.figure(figsize=(8,4))
plt.bar(plot_months, monthly_return, color='seagreen')

plt.grid(True, axis='y')
plt.title("Synthetic Stock Monthly Returns (%)")
plt.xlabel("Month")
plt.ylabel("Return %")

plt.show()
```



Plotting Pie Charts

```
In [11]: import matplotlib.pyplot as plt

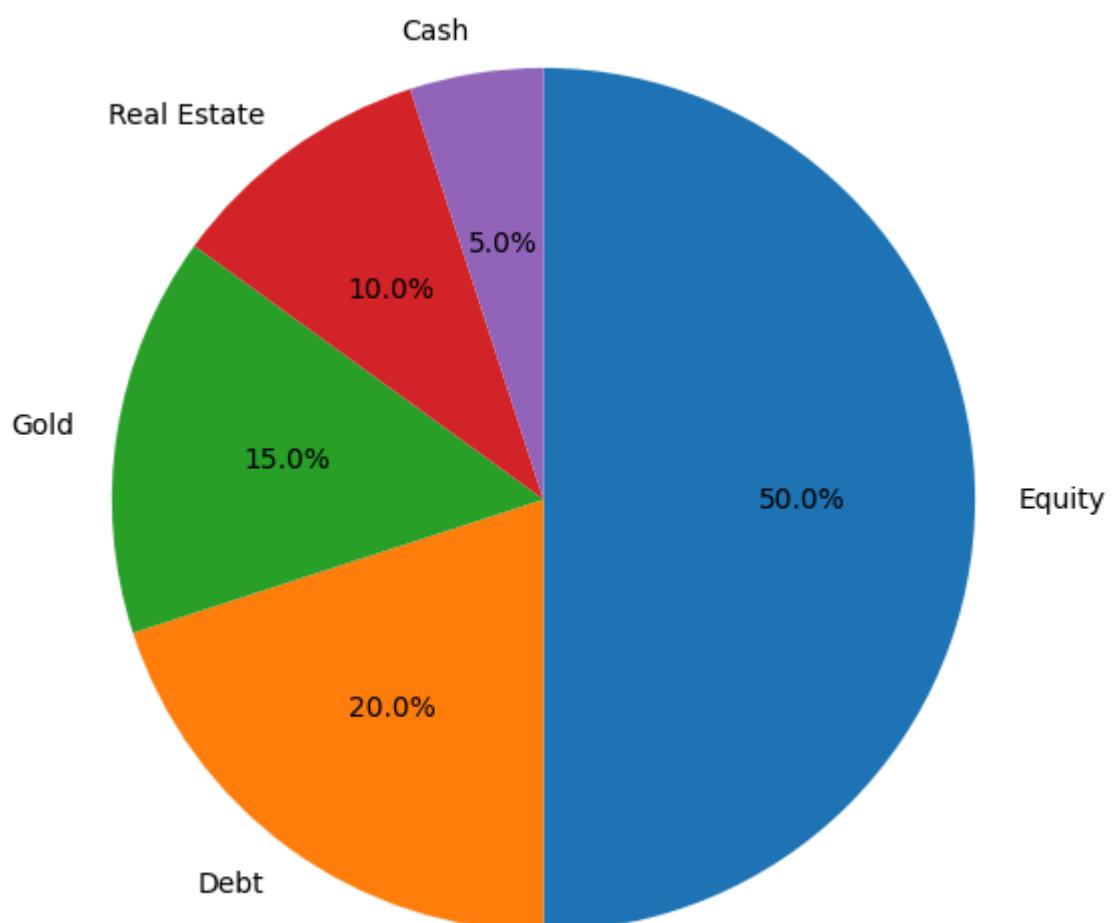
assets = ['Equity', 'Debt', 'Gold', 'Real Estate', 'Cash']
weights = [50, 20, 15, 10, 5]

plt.figure(figsize=(7,7))

plt.pie(weights, labels=assets, autopct='%1.1f%%',
        startangle=90, counterclock=False)

plt.title("Portfolio Distribution (Synthetic)")
plt.show()
```

Portfolio Distribution (Synthetic)



Optional Read (Candlestick Generation from Lists)

```
In [12]: open_price = [100, 102, 101, 105, 107, 106, 108, 110, 109, 111]
close_price = [102, 101, 104, 106, 108, 105, 109, 112, 110, 113]
high_price = [103, 103, 105, 107, 109, 108, 110, 113, 111, 114]
low_price = [98, 100, 100, 103, 105, 104, 107, 108, 108, 110]
```

```
In [13]: import matplotlib.pyplot as plt

plt.figure(figsize=(6,4))

for i in range(len(open_price)):

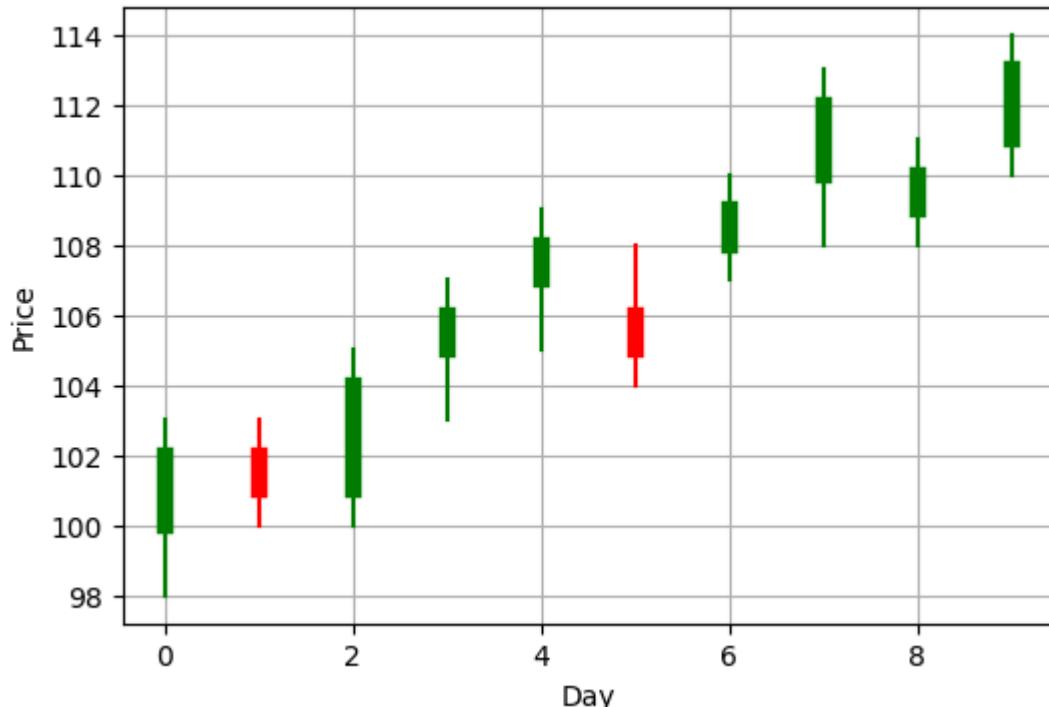
    # Choose color: green if close >= open else red
    color = 'green' if close_price[i] >= open_price[i] else 'red'

    # Wick: High to Low
    plt.plot([i, i], [low_price[i], high_price[i]], color=color)

    # Candle Body: Open to Close (thick line)
    plt.plot([i, i], [open_price[i], close_price[i]],
             color=color, linewidth=6)

plt.grid(True)
plt.title("Simple 10-Day Candlestick Chart")
plt.xlabel("Day")
plt.ylabel("Price")
plt.show()
```

Simple 10-Day Candlestick Chart



```
In [ ]:
```

TTP Assignment 9

NumPy Functions & Matplotlib Plotting

Question #1 – Create and Print Basic NumPy Array

Create an array **A** using the following syntax:

```
A = np.array([12, 24, 36, 48, 60, 72])
```

Print:

- The array
- The number of elements in the array using `A.size`
- The data type using `A.dtype`

```
In [2]: import numpy as np
A = np.array([12, 24, 36, 48, 60, 72])
print(A)
print("Size:", A.size)
print(type(A))
```

```
[12 24 36 48 60 72]
Size: 6
<class 'numpy.ndarray'>
```

Question #2 – Create an Array Using `np.arange()`

Create an array **x** using:

```
x = np.arange(15, 45, 3)
```

Print the array and its length.

```
In [4]: x = np.arange(15, 45, 3)
print("array:",x)
print("length:",len(x))
```

```
array: [15 18 21 24 27 30 33 36 39 42]
length: 10
```

Question #3 – Reshape the Array

Given the array:

```
y = np.arange(16)
```

Reshape it into a **4 × 4** matrix and print the result.

```
In [5]: y = np.arange(16)
print(y)
y.reshape(4, 4)

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
```

```
Out[5]: array([[ 0,  1,  2,  3],
   [ 4,  5,  6,  7],
   [ 8,  9, 10, 11],
   [12, 13, 14, 15]])
```

Question #4 – Generate Evenly Spaced Values

Using:

```
lin = np.linspace(5, 25, 7)
```

Print:

- The array
- The step difference between consecutive values

```
In [6]: lin = np.linspace(5, 25, 7)
print(lin)
step = lin[1] - lin[0]
print("Step difference:", step)

[ 5.          8.33333333 11.66666667 15.          18.33333333 21.66666667
 25.        ]
Step difference: 3.33333333333334
```

Question #5 – Create Ones and Zeros Matrices

Create the following using the given syntax:

- np.ones((6, 6))
- np.zeros((3, 7), dtype=int)

Print both matrices.

```
In [7]: ones_matrix = np.ones((6,6))
print("Ones Matrix:",ones_matrix)
zeros_matrix = np.zeros((3, 7), dtype=int)
print("zeros Matrix:",zeros_matrix)

Ones Matrix: [[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
zeros Matrix: [[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

Question #6 – Identity Matrix

Create a 7×7 identity matrix using:

```
np.identity(7)
```

Print the resulting matrix.

```
In [8]: id_matrix = np.identity(7)
print(id_matrix)
```

```
[[1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1.]]
```

Question #7 – Synthetic Open/Close Prices

Generate prices using:

```
np.random.seed(20)
open_p = np.random.randint(90, 110, size=12)
close_p = np.random.randint(85, 115, size=12)
```

Using NumPy functions, print:

- Mean of open prices
- Standard deviation of open prices
- Maximum and minimum close prices
- Indices of max and min close prices

```
In [9]: np.random.seed(20)
open_p = np.random.randint(90, 110, size=12)
close_p = np.random.randint(85, 115, size=12)

print("open prices:", open_p)
print("close prices:", close_p)

mean_open = np.mean(open_p)
std_open = np.std(open_p)
max_close = np.max(close_p)
min_close = np.min(close_p)
idx_max_close = np.argmax(close_p)
idx_min_close = np.argmin(close_p)

print("Mean of open prices:", mean_open)
print("Standard deviation of open prices:", std_open)
print("Maximum close prices:", max_close)
print("Minimum close prices:", min_close)
print("Indices of max close prices:", idx_max_close)
print("Indices of min close prices:", idx_min_close)
```

```
open prices: [ 93 105  99 101  97  92  90  98 109 106  96  96]
close prices: [101  94  90  92  90  87  91 111  98  96 111 110]
Mean of open prices: 98.5
Standard deviation of open prices: 5.5901699437494745
Maximum close prices: 111
Minimum close prices: 87
Indices of max close prices: 7
Indices of min close prices: 5
```

Question #8 – Boolean Masking

Given:

```
prices = np.array([120, 95, 102, 88, 140, 99])
```

Create a boolean mask using:

```
mask = prices > 100
```

Print the mask and the filtered output: `prices[mask]`

```
In [10]: prices = np.array([120, 95, 102, 88, 140, 99])
mask = prices > 100
print("Mask:", mask)
print("Filtered prices:", prices[mask])
```

```
Mask: [ True False  True False  True False]
```

```
Filtered prices: [120 102 140]
```

Question #9 – Generate BUY/HOLD Signals

Given:

```
p = np.array([98, 105, 112, 99, 120])
```

Generate signals using:

```
np.where(p > 110, "BUY", "HOLD")
```

Print the resulting array.

```
In [12]: p = np.array([98, 105, 112, 99, 120])  
signal = np.where(p > 100, "BUY", "HOLD")  
print(signal)  
['HOLD' 'BUY' 'BUY' 'HOLD' 'BUY']
```

Question #10 – Cumulative Sum and Product

Given:

```
arr = np.array([3, 1, 4, 2, 5])
```

Compute:

- np.cumsum(arr)
- np.cumprod(arr)

Print both results.

```
In [13]: arr = np.array([3, 1, 4, 2, 5])  
result = np.cumsum(arr)  
print(result)  
  
result1 = np.cumprod(arr)  
print(result1)  
[ 3  4  8 10 15]  
[ 3    3   12  24 120]
```

Question #11 – Plot a Simple Line Chart

Given the list:

```
p = [120, 122, 121, 125, 128, 130, 129]
```

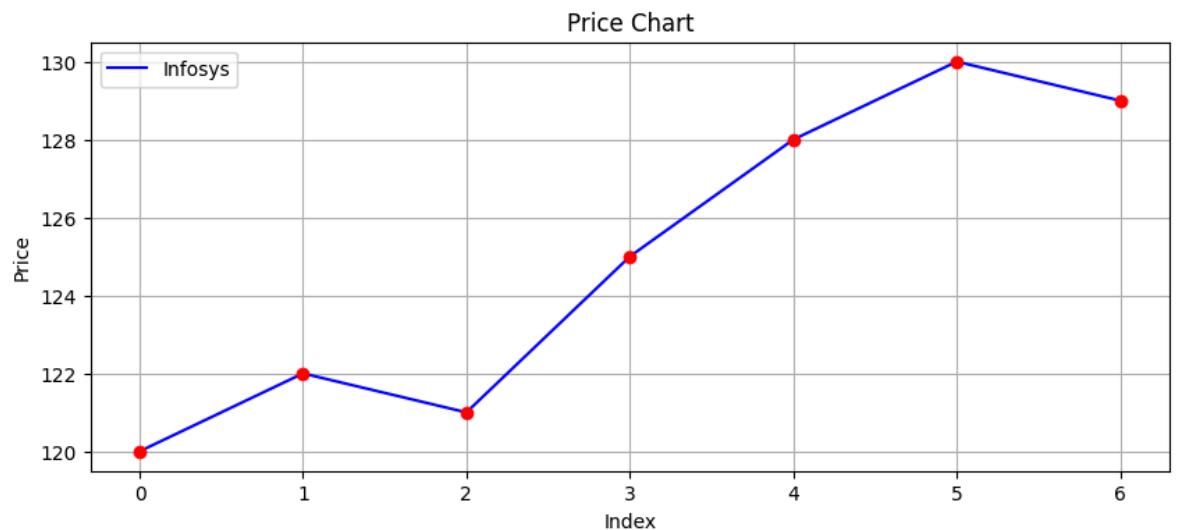
Plot a line chart of these prices using plt.plot() and add:

- Title • X and Y labels • Grid

```
In [15]: import numpy as np
import matplotlib.pyplot as plt

p = [120, 122, 121, 125, 128, 130, 129]

plt.figure(figsize=(10,4))
plt.plot(p, 'b', label="Infosys")
plt.plot(p, 'ro')
plt.grid(True)
plt.title("Price Chart")
plt.xlabel("Index")
plt.ylabel("Price")
plt.legend()
plt.show()
```



Question #12 – Plot High, Low, Open, Close (HLOC)

Given synthetic lists:

```
open_p = [100, 101, 103, 105, 107]
close_p = [102, 100, 104, 106, 108]
high_p = [103, 103, 105, 107, 109]
low_p = [98, 99, 100, 103, 104]
```

Plot all four series on the same chart with different colors and markers. Add a legend.

```
In [1]: import matplotlib.pyplot as plt

# Given data
open_p = [100, 101, 103, 105, 107]
close_p = [102, 100, 104, 106, 108]
high_p = [103, 103, 105, 107, 109]
low_p = [98, 99, 100, 103, 104]

# X-axis (time index)
days = range(1, len(open_p) + 1)

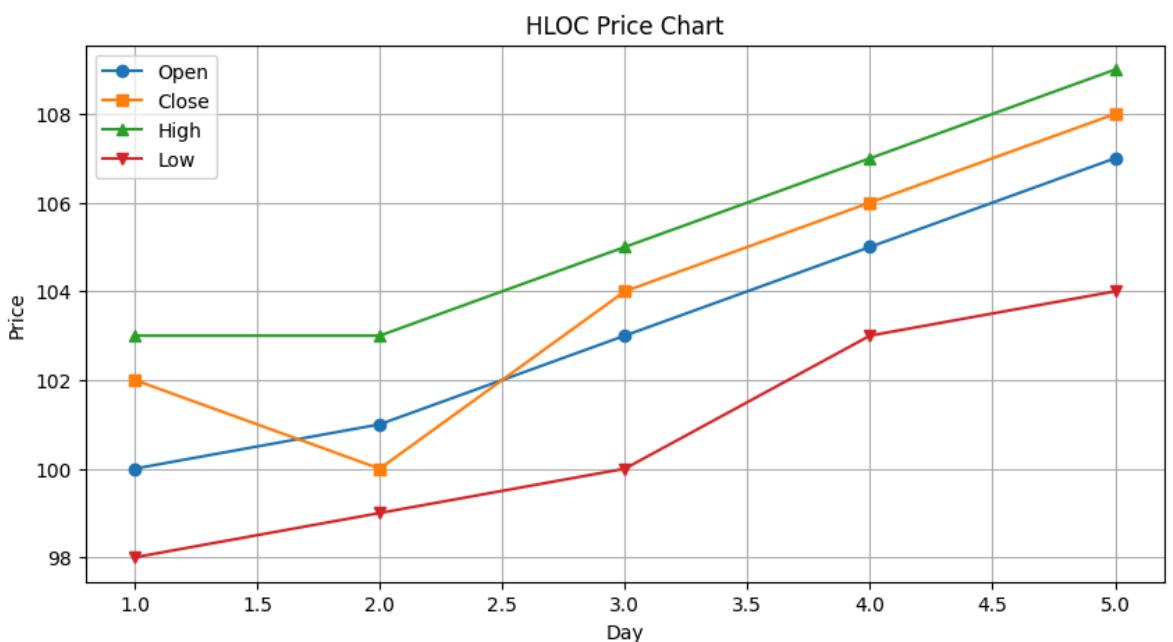
# Plot all series
plt.figure(figsize=(10, 5))

plt.plot(days, open_p, marker='o', label='Open')
plt.plot(days, close_p, marker='s', label='Close')
plt.plot(days, high_p, marker='^', label='High')
plt.plot(days, low_p, marker='v', label='Low')

# Labels and title
plt.xlabel('Day')
plt.ylabel('Price')
plt.title('HLOC Price Chart')

# Legend and grid
plt.legend()
plt.grid(True)

# Show plot
plt.show()
```



Question #13 – Scatter Plot for Synthetic Data

Generate 2D data using:

```
y = np.random.standard_normal((200, 2))
```

Create a scatter plot of:

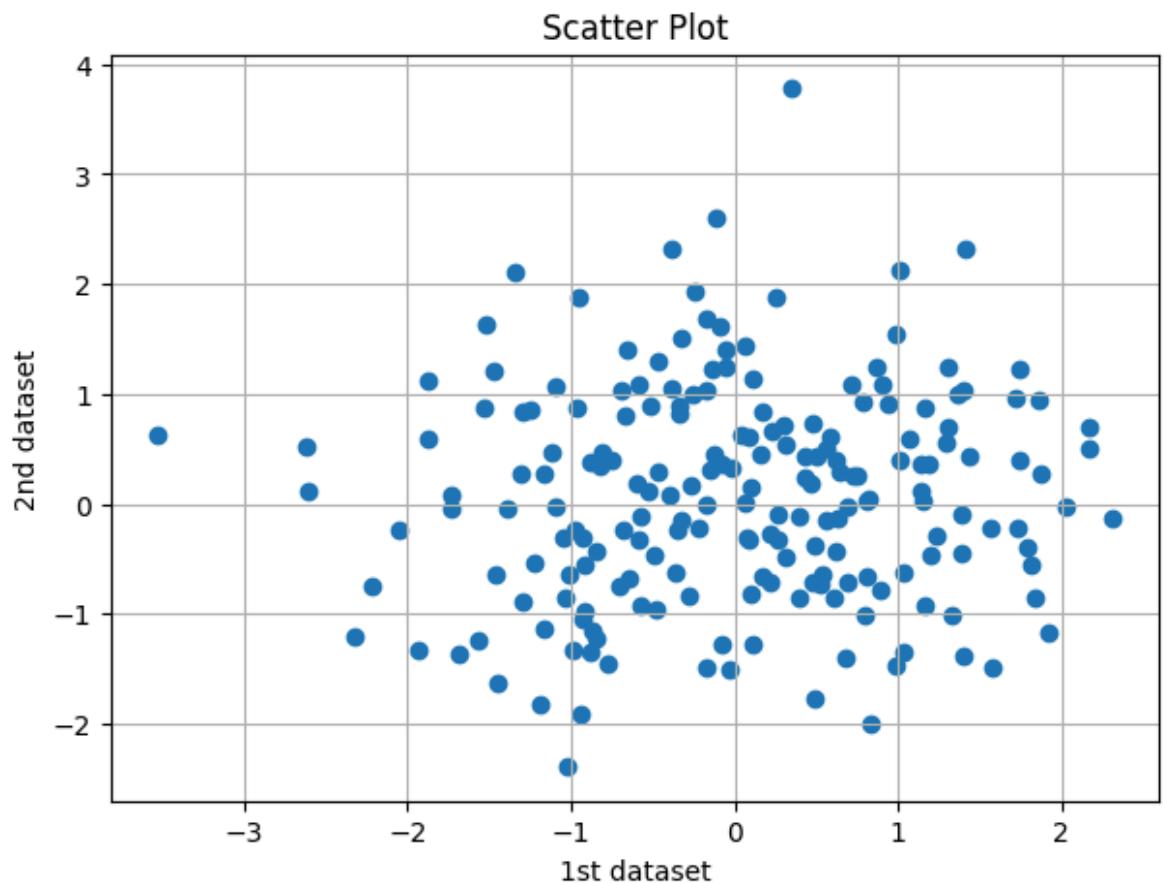
v[::, 0] vs v[::, 1]

```
In [18]: y = np.random.standard_normal((200, 2))

plt.figure(figsize=(7, 5))

plt.scatter(y[:, 0], y[:, 1], marker='o')

plt.grid(True)
plt.xlabel('1st dataset')
plt.ylabel('2nd dataset')
plt.title('Scatter Plot')
plt.show()
```



Question #14 – Histogram of Random Returns

Set the seed:

```
np.random.seed(50)
```

Generate:

```
ret = np.random.standard_normal(500)
```

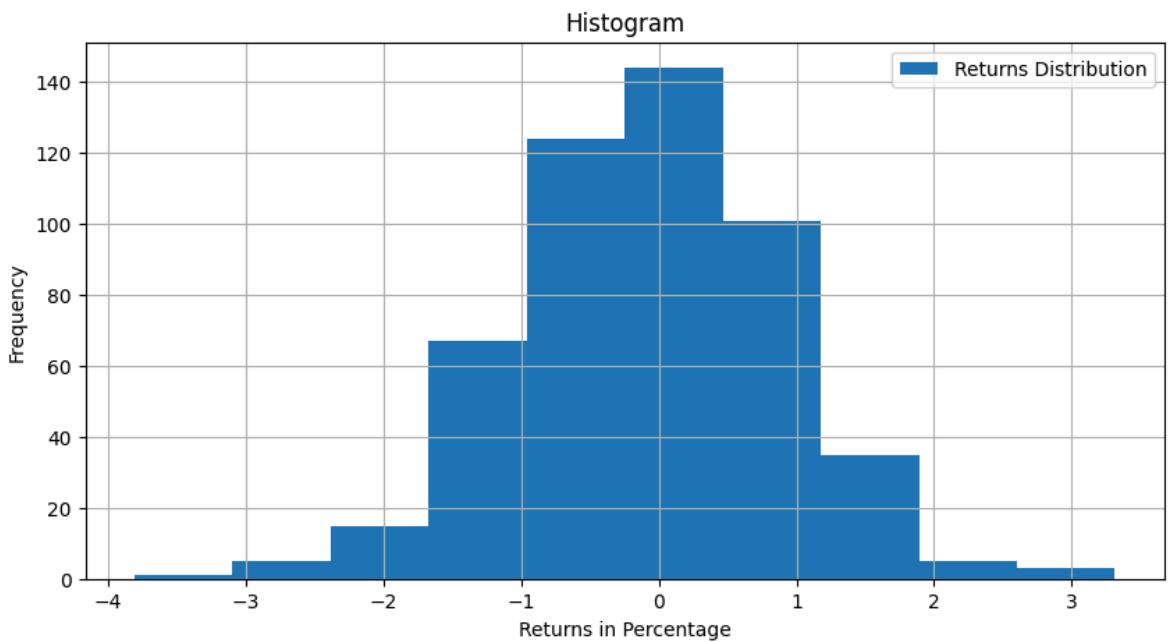
Plot a histogram of these values using `plt.hist()` and add axis labels.

```
In [19]: np.random.seed(50)
ret = np.random.standard_normal(500)

plt.figure(figsize=(10, 5))

plt.hist(ret, label=['Returns Distribution'])

plt.grid(True)
plt.legend(loc=0)
plt.ylabel('Frequency')
plt.xlabel('Returns in Percentage')
plt.title('Histogram')
plt.show()
```



```
In [ ]:
```

```
### TTP Foundation 9
```

Pandas Series

In Python, understanding Series is helpful to understanding dataframes.

Series are indexed data frame with only one data column. It is easier to understand them first before moving to study complex data frames. In Python, a Series is a one-dimensional labeled array in the Pandas library that can hold any data type - integers, floats, strings, or even Python objects.

◆ Basic Structure

A Series has two parts:

Values – the actual data (like numbers or strings) Index – labels that identify each value

Think of it like a column in a spreadsheet or a single column of a DataFrame.

```
In [1]: import pandas as pd  
import warnings  
warnings.filterwarnings("ignore")
```

The constructor for Series data structure is `pandas.Series`

```
pd.Series(data=[10, 20, 30], index=['a', 'b', 'c'])
```

`pd.Series()`

when you create a Pandas Series without specifying an index, Pandas automatically assigns a default index which starts with 0, 1, 2....with step +1

```
In [2]: # Series created using a List  
series_1 = pd.Series([100, 200, 300, 400, 500])  
print(series_1)
```

```
0    100  
1    200  
2    300  
3    400  
4    500  
dtype: int64
```

```
In [3]: # Series created using a List  
series_2 = pd.Series([10.1, 20, 'python', 40.4])  
  
print(series_2)
```

```
0      10.1  
1        20  
2    python  
3      40.4  
dtype: object
```

The above series returns an 'object' datatype since a Python object is created at this instance.

```
In [4]: # Create a Series of 5 states and their capitals
state_capitals = pd.Series(
    ['Mumbai', 'Chennai', 'Kolkata', 'Bengaluru', 'Jaipur'],
    index=['Maharashtra', 'Tamil Nadu', 'West Bengal', 'Karnataka', 'Rajasthan']
)
print(state_capitals)
```

```
Maharashtra      Mumbai
Tamil Nadu       Chennai
West Bengal      Kolkata
Karnataka        Bengaluru
Rajasthan        Jaipur
dtype: object
```

```
In [5]: # define the index
stocks_set1 = ['Reliance', 'TCS', 'Infosys', 'HDFC Bank']

# Price list
S1 = pd.Series([2500, 3600, 1550, 1700], index=stocks_set1)

print("Series S1:")
print(S1)
```

```
Series S1:
Reliance      2500
TCS          3600
Infosys      1550
HDFC Bank    1700
dtype: int64
```

```
In [6]: # define the index of Series S2
stocks_set2 = ['Reliance', 'TCS', 'Infosys', 'HDFC Bank']

# Second price list
S2 = pd.Series([2550, 3700, 1600, 1725], index=stocks_set2)

print("Series S2:")
print(S2)
```

```
Series S2:
Reliance      2550
TCS          3700
Infosys      1600
HDFC Bank    1725
dtype: int64
```

```
In [7]: # Add both Series
print("Sum of S1 and S2:")
print(S1 + S2)
```

```
Sum of S1 and S2:
Reliance      5050
TCS          7300
Infosys      3150
HDFC Bank    3425
dtype: int64
```

Adding lists that have different indexes will create 'NaN' values
 'NaN' is short for 'Not a Number'. It fills the space for missing or corrupt data.

Methods or functions

Few important methods or functions that can be applied on Series.

Series.index

It is useful to know the range of the index when the series is large.

```
In [8]: My_Series = pd.Series([10, 20, 30, 40, 50])
print(My_Series.index)
```

```
RangeIndex(start=0, stop=5, step=1)
```

```
In [9]: My_Series
```

```
Out[9]: 0    10
1    20
2    30
3    40
4    50
dtype: int64
```

```
In [10]: My_Series2 = pd.Series([100, 1, 1000, -5], index=['a', 'b', 'c', 'd'])
print(My_Series2.index)
```

```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
In [11]: My_Series2
```

```
Out[11]: a    100
b     1
c   1000
d    -5
dtype: int64
```

Series.values

It returns the values of the series.

```
In [12]: print(My_Series.values)
```

```
[10 20 30 40 50]
```

```
In [13]: My_Series.values
```

```
Out[13]: array([10, 20, 30, 40, 50], dtype=int64)
```

Series.isnull()

We can check for missing values with this method.

```
In [14]: print(S1 + S2)
```

```
Reliance      5050
TCS          7300
Infosys      3150
HDFC Bank    3425
dtype: int64
```

```
In [15]: # Returns whether the values are null or not. If it is 'True' then the value
          (S1 + S2).isnull()
```

```
Out[15]: Reliance      False
          TCS          False
          Infosys      False
          HDFC Bank    False
          dtype: bool
```

```
In [16]: stocks_set1 = ['Reliance', 'TCS', 'Infosys', 'HDFC Bank', 'ICICI Bank', 'Bharti Airtel']

# Create first Series
S1 = pd.Series([2500, 3650, 1550, 1720, 980, 950], index=stocks_set1)

stocks_set2 = ['Reliance', 'TCS', 'Infosys', 'HDFC Bank', 'ICICI Bank', 'Adani Ports']

# Create second Series
S2 = pd.Series([2550, 3725, 1600, 1740, 1000, 890], index=stocks_set2)

# Add both Series
print("Sum of S1 and S2:")
print(S1 + S2)
```

Sum of S1 and S2:

```
Adani Ports        NaN
Bharti Airtel     NaN
HDFC Bank       3460.0
ICICI Bank      1980.0
Infosys         3150.0
Reliance        5050.0
TCS            7375.0
dtype: float64
```

```
In [17]: (S1 + S2).isnull()
```

```
Out[17]: Adani Ports      True
          Bharti Airtel    True
          HDFC Bank       False
          ICICI Bank      False
          Infosys         False
          Reliance        False
          TCS              False
          dtype: bool
```

Series.dropna()

One way to deal with the 'NaN' values is to drop them completely from the series. This method filters out missing data.

```
In [18]: print(S1 + S2)
```

```
Adani Ports      NaN
Bharti Airtel    NaN
HDFC Bank       3460.0
ICICI Bank      1980.0
Infosys         3150.0
Reliance        5050.0
TCS              7375.0
dtype: float64
```

```
In [19]: print((S1 + S2).dropna())
```

```
HDFC Bank      3460.0
ICICI Bank     1980.0
Infosys        3150.0
Reliance       5050.0
TCS            7375.0
dtype: float64
```

```
In [20]: (S1 + S2).isnull()
```

```
Out[20]: Adani Ports      True
          Bharti Airtel    True
          HDFC Bank       False
          ICICI Bank      False
          Infosys         False
          Reliance        False
          TCS              False
          dtype: bool
```

dropna() doesn't change the original Series - it just returns a new Series that excludes missing values If you want to permanently update the Series, then we should store it in different variable

```
In [21]: S3 = (S1 + S2).dropna()
print(S3)
```

```
HDFC Bank    3460.0
ICICI Bank   1980.0
Infosys      3150.0
Reliance     5050.0
TCS          7375.0
dtype: float64
```

```
In [22]: S3.isnull()
```

```
Out[22]: HDFC Bank    False
          ICICI Bank   False
          Infosys      False
          Reliance     False
          TCS          False
          dtype: bool
```

Series.fillna(1)

Another way to deal with the 'NaN' values is to fill a custom value of your choice. Here, we are filling the 'NaN' values with the value '1'.

```
In [23]: print((S1 + S2).fillna(1000)) # Check the filled output
```

```
Adani Ports    1000.0
Bharti Airtel  1000.0
HDFC Bank     3460.0
ICICI Bank    1980.0
Infosys       3150.0
Reliance      5050.0
TCS           7375.0
dtype: float64
```

pd.Series.apply()

If at all one wants to 'apply' any functions on a particular series, e.g. one wants to 'sine' of each value in the series, then it is possible in pandas.

Series.apply (func)

func = A python function that will be applied to every single value of the series.

```
In [24]: import numpy as np
```

```
In [25]: math_Series = pd.Series([10, 20, 36, 40, 50, 64])
```

```
In [26]: math_Series.apply(np.sin) # Find 'sine' of each value in the series
```

```
Out[26]: 0    -0.544021
          1     0.912945
          2    -0.991779
          3     0.745113
          4    -0.262375
          5     0.920026
          dtype: float64
```

```
In [27]: math_Series.apply(np.sqrt)
```

```
Out[27]: 0    3.162278
          1    4.472136
          2    6.000000
          3    6.324555
          4    7.071068
          5    8.000000
         dtype: float64
```

```
In [28]: print(math_Series)
```

```
0    10
1    20
2    36
3    40
4    50
5    64
dtype: int64
```

```
In [29]: #Overwriting existing data
```

```
math_Series = math_Series.apply(np.sqrt)
print(math_Series)
```

```
0    3.162278
1    4.472136
2    6.000000
3    6.324555
4    7.071068
5    8.000000
dtype: float64
```

```
In [30]: #Create a random Pandas Series of float numbers
```

```
s = pd.Series(np.random.randn(10))
print(s)
```

```
0    -0.153177
1     0.511693
2     0.061868
3     0.215273
4    -0.152491
5     0.654514
6     0.413588
7     0.175266
8    -1.984394
9    -0.141943
dtype: float64
```

```
In [31]: #Check the default index
```

```
s.index
```

```
Out[31]: RangeIndex(start=0, stop=10, step=1)
```

```
In [32]: #Create a new series specifying the index
k = pd.Series(np.random.randn(5), index = ['a', 'b', 'c', 'd', 'e'])
print(k)

a    0.159386
b   -0.858283
c   -1.218989
d   -0.164702
e    0.942183
dtype: float64
```

```
In [33]: #Create a pandas series using dictionary
dictionary = {'a':1000, 'b':2000, 'c':3000, 'd':4000, 'e':5000}
w = pd.Series(dictionary)
print(w)

a    1000
b    2000
c    3000
d    4000
e    5000
dtype: int64
```

```
In [34]: #Create a pandas series using numpy array
arr = np.array([1, 2, 3, 4, 5])
arr = pd.Series(arr)
print(arr)

0    1
1    2
2    3
3    4
4    5
dtype: int32
```

```
In [35]: #Performing operations similar to Numpy Arrays
arr[0]
```

```
Out[35]: 1
```

```
In [36]: w['a']
```

```
Out[36]: 1000
```

```
In [37]: #Vectorized operation
w = w + 2
```

```
In [38]: print(w)
```

```
a    1002
b    2002
c    3002
d    4002
e    5002
dtype: int64
```

TTP Assignment 10

Pandas Series

Question #1 – Create and Print a Simple Pandas Series

Create a Series **S1** using:

```
S1 = pd.Series([15, 30, 45, 60, 75])
```

Print:

- The Series
- The index
- The values

```
In [1]: import pandas as pd

S1 = pd.Series([15, 30, 45, 60, 75])
print("Series:\n", S1)
print("Index:\n", S1.index)
print("Values:\n", S1.values)
```

```
Series:
 0    15
 1    30
 2    45
 3    60
 4    75
dtype: int64
Index:
RangeIndex(start=0, stop=5, step=1)
Values:
[15 30 45 60 75]
```

Question #2 – Series with Mixed Data Types

Create a Series **S2** using:

```
S2 = pd.Series([3.14, 'Data', True, 99])
```

Print:

- The Series
- The data type using `S2.dtype`

```
In [3]: S2 = pd.Series([3.14, 'Data', True, 99])
```

```
print("Series\n", S2)
print("Data type:", S2.dtype)
```

```
Series
0    3.14
1    Data
2    True
3    99
dtype: object
Data type: object
```

Question #3 – Series with Custom Index

Create a Series of five countries and their currencies:

```
pd.Series(['Dollar', 'Euro', 'Yen', 'Rupee', 'Pound'],
          index=['USA', 'Germany', 'Japan', 'India', 'UK'])
```

Print the full Series.

```
In [4]: S3 = pd.Series(['Dollar', 'Euro', 'Yen', 'Rupee', 'Pound'],
                      index=['USA', 'Germany', 'Japan', 'India', 'UK'])
print(S3)
```

```
USA      Dollar
Germany    Euro
Japan      Yen
India      Rupee
UK        Pound
dtype: object
```

Question #4 – Create Fruit Price Series

Create a Series F1 using fruit names as index and prices as values:

```
fruits = ['Apple', 'Banana', 'Orange', 'Grapes']
F1 = pd.Series([120, 40, 60, 150], index=fruits)
```

Print:

- The Series
- The price of **Orange**

```
In [5]: fruits = ['Apple', 'Banana', 'Orange', 'Grapes']
F1 = pd.Series([120, 40, 60, 150], index=fruits)
print("Series:\n", F1)
print("Price of Orange:", F1['Orange'])
```

```
Series:
Apple    120
Banana   40
Orange   60
Grapes   150
dtype: int64
Price of Orange: 60
```

Question #5 – Add Two Series with Same Index

Create two Series representing student marks:

```
T1 = pd.Series([80, 75, 90, 65])
T2 = pd.Series([70, 85, 95, 60])
```

Print:

- The total marks using `T1 + T2`

```
In [8]: T1 = pd.Series([80, 75, 90, 65])
T2 = pd.Series([70, 85, 95, 60])

print("Total Marks:\n", T1 + T2)
```

```
Total Marks:
0    150
1    160
2    185
3    125
dtype: int64
```

Question #6 – Add Series with Mismatched Index

Create:

```
S1 = pd.Series([4000, 4500, 5000, 5500], index=['Jan', 'Feb', 'Mar', 'Apr'])
S2 = pd.Series([4200, 4800, 5100], index=['Feb', 'Mar', 'Apr'])
```

Add them and print:

- The result
- Missing values using `(S1 + S2).isnull()`

```
In [9]: S1 = pd.Series([4000, 4500, 5000, 5500], index=['Jan', 'Feb', 'Mar', 'Apr'])
S2 = pd.Series([4200, 4800, 5100], index=['Feb', 'Mar', 'Apr'])

result = S1 + S2

print("Result:\n", result)
print("Missing Values:\n", result.isnull())
```

```
Result:
Apr    10600.0
Feb    8700.0
Jan      NaN
Mar    9800.0
dtype: float64
Missing Values:
Apr    False
Feb    False
Jan     True
Mar    False
dtype: bool
```

Question #7 – Drop Missing Values

Using the result of `S1 + S2` from Question 6, print:

- The cleaned Series using `.dropna()`

```
In [10]: cleaned = result.dropna()
print(cleaned)
```

```
Apr    10600.0
Feb    8700.0
Mar    9800.0
dtype: float64
```

Question #8 – Replace Missing Values

Using the same result from Question 6, replace missing values with **0**:

```
(S1 + S2).fillna(0)
```

```
In [11]: filled = result.fillna(0)
print(filled)
```

```
Apr    10600.0
Feb    8700.0
Jan     0.0
Mar    9800.0
dtype: float64
```

Question #9 – Apply NumPy Functions on a Series

Create a Series **temp** representing temperatures:

```
temp = pd.Series([25, 28, 30, 32, 35])
```

Apply and print:

- np.sqrt(temp)
- np.log(temp)

```
In [13]: import numpy as np

temp = pd.Series([25, 28, 30, 32, 35])

print("Square root\n",np.sqrt(temp))
print("Log values\n",np.log(temp))
```

```
Square root
0    5.000000
1    5.291503
2    5.477226
3    5.656854
4    5.916080
dtype: float64
Log values
0    3.218876
1    3.332205
2    3.401197
3    3.465736
4    3.555348
dtype: float64
```

Question #10 – Create Series from a Dictionary

Given the dictionary:

```
cars = {'Hyundai':900000, 'Maruti':700000, 'Honda':1100000, 'Tata':850000}
```

Create a Series **C** and print:

- The entire Series
- The price of **Honda**
- Prices of **Maruti** and **Tata** using list indexing

In [14]:

```
cars = {'Hyundai':900000, 'Maruti':700000, 'Honda':1100000, 'Tata':850000}
```

```
C = pd.Series(cars)
```

```
print("Series:\n", C)
```

```
print("Price of Honda:\n", C['Honda'])
```

```
print("Price of Maruti and Tata:\n", C[['Maruti', 'Tata']])
```

Series:

Hyundai 900000

Maruti 700000

Honda 1100000

Tata 850000

dtype: int64

Price of Honda:

1100000

Price of Maruti and Tata:

Maruti 700000

Tata 850000

dtype: int64

In []:

```
### TTP_Foundation 10
```

Pandas DataFrame()

The underlying idea of a dataframe is based on 'spreadsheets'. In other words, dataframes store data in discrete rows and columns, where each column can be named (something that is not possible in Arrays but is possible in Series). There are also multiple columns in a dataframe (as opposed to Series, where there can be only one discrete indexed column). In pandas, the object we want to create is a DataFrame (a 2-dimensional table of rows and columns).

The constructor for a dataframe is `pandas.DataFrame(data=None, index=None)` or if you are using 'pd' as an alias for pandas, then it would be `pd.DataFrame(data=None, index=None)`

```
df = pd.DataFrame(
```

```
data={'A': [1, 2, 3], 'B': [4, 5, 6]},
```

```
index=['row1', 'row2', 'row3']
```

```
In [1]: import pandas as pd
```

```
df = pd.DataFrame(data={'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['row1', 'row2',  
print(df)
```

	A	B
row1	1	4
row2	2	5
row3	3	6

```
In [2]: import numpy as np
import yfinance as yf
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")

# A DataFrame has both row and column indexes

#Python dictionary containing lists as values, stock_name is a key
my_portfolio = {
    "stock_name": ["Reliance", "TCS", "Infosys", "HDFC Bank", "ICICI Bank"],
    "quantity_owned": [120, 80, 150, 60, 90],
    "average_buy_price": ["₹2500", "₹3650", "₹1550", "₹1720", "₹980"] }

# Create the DataFrame
my_portfolio_frame = pd.DataFrame(my_portfolio)

# Display the DataFrame
my_portfolio_frame
```

Out[2]:

	stock_name	quantity_owned	average_buy_price
0	Reliance	120	₹2500
1	TCS	80	₹3650
2	Infosys	150	₹1550
3	HDFC Bank	60	₹1720
4	ICICI Bank	90	₹980

Customize index of the dataframe

In the above output, you can see that the 'index' is the default one which starts from 0. One can customize this index.

```
In [3]: new_index = ["stock 1", "stock 2", "stock 3", "stock 4", "stock 5"] # List

# Please notice that we have not kept index as default i.e.'none'
my_portfolio_frame = pd.DataFrame(my_portfolio, index=new_index)

my_portfolio_frame
```

Out[3]:

	stock_name	quantity_owned	average_buy_price
stock 1	Reliance	120	₹2500
stock 2	TCS	80	₹3650
stock 3	Infosys	150	₹1550
stock 4	HDFC Bank	60	₹1720
stock 5	ICICI Bank	90	₹980

Rearrange the columns in a dataframe

```
In [4]: # you can define the order of columns using the parameter columns=[] in the constructor

my_portfolio_frame = pd.DataFrame(my_portfolio, columns=[ "stock_name", "average_buy_price", "quantity_owned"])

my_portfolio_frame
```

Out[4]:

	stock_name	average_buy_price	quantity_owned
stock 1	Reliance	₹2500	120
stock 2	TCS	₹3650	80
stock 3	Infosys	₹1550	150
stock 4	HDFC Bank	₹1720	60
stock 5	ICICI Bank	₹980	90

Use an existing column as an index of a dataframe

We will use the column 'stock_name' as the index of the dataframe.

```
In [5]: my_portfolio_frame = pd.DataFrame(my_portfolio, columns=[ "average_buy_price", "quantity_owned"], index=my_portfolio["stock_name"])

my_portfolio_frame
```

Out[5]:

	average_buy_price	quantity_owned
Reliance	₹2500	120
TCS	₹3650	80
Infosys	₹1550	150
HDFC Bank	₹1720	60
ICICI Bank	₹980	90

Access a column in a dataframe

We can access or retrieve a single or multiple columns by their names or by their location.

```
In [6]: # Column name is passed within quotes within square brackets
print(my_portfolio_frame[ "quantity_owned"])
```

Reliance	120
TCS	80
Infosys	150
HDFC Bank	60
ICICI Bank	90

Name: quantity_owned, dtype: int64

Time Series Data

Time series data is a sequence of observations recorded **over time**, usually at **regular intervals** - such as every minute, hour, day, or week. Each observation includes a **timestamp** (when the data was captured) and one or more **values** (the measurements recorded at that time).

Example: Daily OHLCV Data

Date	Open	High	Low	Close	Volume
1 Jan	150	155	148	153	12,000
2 Jan	153	158	151	157	14,500
3 Jan	157	160	155	158	13,200

Note: OHLCV stands for **Open, High, Low, Close, and Volume** - common fields in **financial time series** that record how a stock or asset's price and trading activity change **over time**.

Key Points

- Time series helps track **trends, patterns, and seasonality** over time.
- Commonly used in **finance and economic research**.
- Each new observation depends on **time order**, so the **sequence matters**.
- **Granularity** refers to the **frequency of data collection** - for example, data can be recorded **every second, minute, hour, or day**.

Download financial market data from Yahoo Finance

```
yf.download(tickers,start=None,end=None,interval='1d')
```

Common values: '1d' → daily '1wk' → weekly '1mo' → monthly '1h' → hourly '5m' → 5 minute

df.head()

shows the first 5 rows of a DataFrame. It helps you quickly check whether the data loaded correctly

df.tail()

shows the last 5 rows of a DataFrame.

```
In [7]: import yfinance as yf
import warnings
import numpy as np
warnings.filterwarnings("ignore")

df_TCS = yf.download("TCS.NS", start="2015-01-01", end="2024-12-31")
print(df_TCS.head())
```

[*****100%*****] 1 of 1 completed

Price Ticker	Close TCS.NS	High TCS.NS	Low TCS.NS	Open TCS.NS	Volume TCS.NS
Date					
2015-01-01	1006.856323	1015.340548	1005.056616	1015.340548	366830
2015-01-02	1020.264954	1024.813616	1008.853802	1009.011978	925740
2015-01-05	1004.760376	1028.354061	998.589975	1020.878469	1754242
2015-01-06	967.718323	1000.350040	965.107747	1000.350040	2423784
2015-01-07	956.287476	980.593146	952.233231	976.974030	2636332

```
In [8]: # Rename columns to remove multi-column headers
df_TCS.columns = ['close', 'high', 'low', 'open', 'volume']
```

```
In [9]: # Rearrange to OHLCV sequence
df_TCS = df_TCS[["open", "high", "low", "close", "volume"]]
```

```
In [10]: df_TCS.head(10)
```

```
Out[10]:
```

Date	open	high	low	close	volume
2015-01-01	1015.340548	1015.340548	1005.056616	1006.856323	366830
2015-01-02	1009.011978	1024.813616	1008.853802	1020.264954	925740
2015-01-05	1020.878469	1028.354061	998.589975	1004.760376	1754242
2015-01-06	1000.350040	1000.350040	965.107747	967.718323	2423784
2015-01-07	976.974030	980.593146	952.233231	956.287476	2636332
2015-01-08	966.056544	968.667118	957.414145	966.610352	1565408
2015-01-09	971.040608	996.710849	969.062929	993.704834	3197642
2015-01-12	995.563945	999.914841	981.028000	992.676514	1596006
2015-01-13	996.750784	1000.864320	980.968935	988.009399	1468432
2015-01-14	995.168492	1001.417980	989.037684	997.521912	1787096

Data sanity check using df.info()

In [11]: `df_TCS.info()`

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2466 entries, 2015-01-01 to 2024-12-30
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   open     2466 non-null   float64 
 1   high     2466 non-null   float64 
 2   low      2466 non-null   float64 
 3   close    2466 non-null   float64 
 4   volume   2466 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 115.6 KB
```

A DatetimeIndex is an index in a DataFrame or Series where the row labels are dates or timestamps instead of simple numbers (0,1,2,3...). It is an index made up of datetime objects. This is extremely useful for time-series data like stock prices, portfolio values, sales data, sensor readings, etc.

In [12]: `# Creating a copy of the data which we downloaded in df_TCS
df = df_TCS.copy()`

In [13]: `df.head()`

Out[13]:

	open	high	low	close	volume
Date					
2015-01-01	1015.340548	1015.340548	1005.056616	1006.856323	366830
2015-01-02	1009.011978	1024.813616	1008.853802	1020.264954	925740
2015-01-05	1020.878469	1028.354061	998.589975	1004.760376	1754242
2015-01-06	1000.350040	1000.350040	965.107747	967.718323	2423784
2015-01-07	976.974030	980.593146	952.233231	956.287476	2636332

Dropping rows and/or columns

In the above stock data, it might not be necessary that you need all the columns. Hence, to make your dataframe more understandable, you may drop the columns that you do not need using drop function.

General Syntax for dropping columns:

```
DataFrame.drop(['Column_name'], axis = 1)
```

General Syntax for dropping rows:

```
DataFrame.drop(DataFrame.index[[x,y,z...]])
```

where x,y,z are row index values

In [14]: `# The axis=1 represents that we are considering columns while dropping.
df = df.drop(['high', 'low', 'open'], axis = 1)`

In [15]: `df.head()`

Out[15]:

Date	close	volume
2015-01-01	1006.856323	366830
2015-01-02	1020.264954	925740
2015-01-05	1004.760376	1754242
2015-01-06	967.718323	2423784
2015-01-07	956.287476	2636332

In [16]: `#To make our dataset more readable
df = np.round(df, 2)`

In [17]: `df.head(4)`

Out[17]:

Date	close	volume
2015-01-01	1006.86	366830
2015-01-02	1020.26	925740
2015-01-05	1004.76	1754242
2015-01-06	967.72	2423784

In [18]: `# Drop 1st, 2nd and 3rd rows from data
Here, we are removing multiple rows at once. Therefore double bracket is used
df.drop(df.index[[0,1,2]]).head()`

Out[18]:

Date	close	volume
2015-01-06	967.72	2423784
2015-01-07	956.29	2636332
2015-01-08	966.61	1565408
2015-01-09	993.70	3197642
2015-01-12	992.68	1596006

Rename columns

If we want to rename the column names, while dealing with the dataframe we need to use the `rename` function.

```
In [19]: df = df.rename(columns={'close' : 'Close Price', 'volume' : 'Total_Volume'})
```

```
In [20]: df.tail()
```

```
Out[20]:
```

	Close Price	Total_Volume
Date		
2024-12-23	4018.52	2195338
2024-12-24	4039.01	1181886
2024-12-26	4028.96	1208464
2024-12-27	4024.85	858100
2024-12-30	4019.00	1527169

Sort a dataframe using a column

Sometimes it becomes necessary to sort a stock price dataframe, based on the 'Closing Price'.

```
In [21]: # Sorting dataframe in descending order
```

```
df_filtered = df.sort_values(by="Close Price", ascending=False)
```

```
# Prints top 20 values of closing prices in the given dataset
print(df_filtered.head(20))
```

	Close Price	Total_Volume
Date		
2024-08-30	4389.97	3637222
2024-08-21	4387.80	1896990
2024-08-20	4360.61	2212298
2024-09-13	4359.94	1458786
2024-09-02	4358.44	1216500
2024-09-12	4355.21	2742216
2024-09-16	4350.92	1155120
2024-09-03	4350.05	1717263
2024-08-29	4349.52	2133641
2024-09-10	4345.72	1385591
2024-08-28	4343.98	1848058
2024-09-17	4343.60	1699364
2024-08-26	4340.51	1844164
2024-08-22	4340.08	1829872
2024-08-27	4335.40	930697
2024-08-19	4328.51	2055210
2024-12-13	4323.51	1967048
2024-09-11	4318.24	1325919
2024-09-04	4318.14	1265505
2024-09-05	4314.96	1688793

Downloading Intra-Day data and making it usable

```
In [22]: df_intra = np.round(yf.download("^NSEI", interval="5m"),2)
```

```
[*****100*****] 1 of 1 completed
```

In [23]: `df_intra.head()`

Out[23]:

	Price	Close	High	Low	Open	Volume
	Ticker	^NSEI	^NSEI	^NSEI	^NSEI	^NSEI
	Datetime					
2025-11-10 03:45:00+00:00		25554.60	25590.35	25515.80	25515.80	0
2025-11-10 03:50:00+00:00		25526.55	25565.70	25521.50	25551.65	0
2025-11-10 03:55:00+00:00		25539.25	25541.30	25515.30	25524.45	0
2025-11-10 04:00:00+00:00		25559.35	25561.90	25538.65	25539.60	0
2025-11-10 04:05:00+00:00		25574.70	25579.15	25558.00	25558.00	0

In [24]: `# Rename columns`

```
df_intra.columns = ['close', 'high', 'low', 'open', 'volume']

# Rearrange to OHLCV sequence
df_intra = df_intra[['open', "high", "low", "close", "volume"]]
```

In [25]: `df_intra.head()`

Out[25]:

	open	high	low	close	volume
	Datetime				
2025-11-10 03:45:00+00:00	25515.80	25590.35	25515.80	25554.60	0
2025-11-10 03:50:00+00:00	25551.65	25565.70	25521.50	25526.55	0
2025-11-10 03:55:00+00:00	25524.45	25541.30	25515.30	25539.25	0
2025-11-10 04:00:00+00:00	25539.60	25561.90	25538.65	25559.35	0
2025-11-10 04:05:00+00:00	25558.00	25579.15	25558.00	25574.70	0

In [26]: `df_intra.drop(['volume'], inplace=True, axis=1)`

In [27]: `df_intra.head()`

Out[27]:

	open	high	low	close
	Datetime			
2025-11-10 03:45:00+00:00	25515.80	25590.35	25515.80	25554.60
2025-11-10 03:50:00+00:00	25551.65	25565.70	25521.50	25526.55
2025-11-10 03:55:00+00:00	25524.45	25541.30	25515.30	25539.25
2025-11-10 04:00:00+00:00	25539.60	25561.90	25538.65	25559.35
2025-11-10 04:05:00+00:00	25558.00	25579.15	25558.00	25574.70

In [28]: `df_intra_nifty = df_intra.copy()`

```
In [29]: # Step 1: Convert index to datetime (it already is, but ensures consistency)
df_intra_nifty.index = pd.to_datetime(df_intra.index)

# Step 2: Convert UTC → IST if not in IST
#df_intra_nifty.index = df_intra.index.tz_convert("Asia/Kolkata")

# Step 3: Remove timezone info
df_intra_nifty.index = df_intra.index.tz_localize(None)

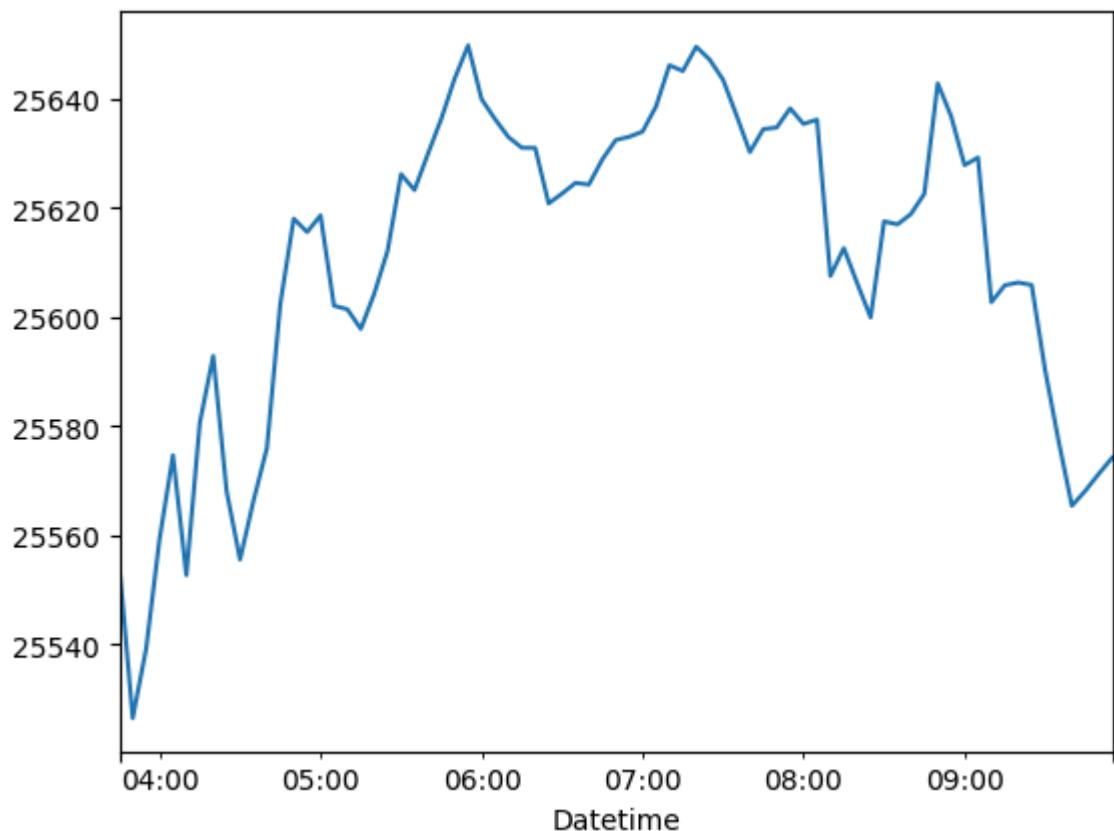
# Step 4: Format to 12-hour clock
df_intra_nifty.index = df_intra.index.strftime("%Y-%m-%d %I:%M:%S:%p")

print(df_intra_nifty.head())
```

Datetime	open	high	low	close
2025-11-10 03:45:00:AM	25515.80	25590.35	25515.80	25554.60
2025-11-10 03:50:00:AM	25551.65	25565.70	25521.50	25526.55
2025-11-10 03:55:00:AM	25524.45	25541.30	25515.30	25539.25
2025-11-10 04:00:00:AM	25539.60	25561.90	25538.65	25559.35
2025-11-10 04:05:00:AM	25558.00	25579.15	25558.00	25574.70

.strftime() destroys the time-series functionality, its a formatting tool only

```
In [42]: df_intra['close']['2025-11-10'].plot();
```



Reading data from a .csv file or .xlsx (excel) file

```
In [36]: df1 = pd.read_csv("NSE_TCS.csv", index_col=0, parse_dates=True)
#df1 = pd.read_excel("NSE_TCS.xlsx",index_col=0,parse_dates=True)
```

```
In [37]: df1.head()
```

```
Out[37]:
```

Date	Open	High	Low	Close
2007-11-23	242.50	242.500	237.225005	240.237503
2007-11-26	245.00	248.500	242.500000	246.274995
2007-11-27	247.50	249.750	242.524995	248.987503
2007-11-28	245.25	251.000	242.500000	244.125000
2007-11-29	250.00	250.425	237.750000	241.512497

```
In [38]: df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4452 entries, 2007-11-23 to 2025-12-02
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   Open     4452 non-null   float64
 1   High    4452 non-null   float64
 2   Low     4452 non-null   float64
 3   Close   4452 non-null   float64
dtypes: float64(4)
memory usage: 173.9 KB
```

```
In [39]: plt.figure(figsize=(12, 8))
plt.plot(df1['High'])
plt.title("TCS Closing Prices")
plt.xlabel("Dates")
plt.ylabel("Price")
plt.grid()
plt.show()
```



```
In [40]: #import os
```

```
In [41]: #os.getcwd()
```

```
In [ ]:
```

TTP Assignment 11

Pandas DataFrame

Question #1 – Create a DataFrame from a Dictionary

Create a DataFrame named **df** using the dictionary:

```
{ "A": [1, 2, 3], "B": [4, 5, 6] }
```

Set the index manually to:

```
["row1", "row2", "row3"]
```

Print the full DataFrame.

```
In [1]: import pandas as pd

df = pd.DataFrame(
    { "A": [1, 2, 3], "B": [4, 5, 6] },
    index=["row1", "row2", "row3"]
)

print(df)
```

	A	B
row1	1	4
row2	2	5
row3	3	6

Question #2 – Create a Portfolio DataFrame

Create a DataFrame called **portfolio_df** from the dictionary:

```
{ "stock": ["A", "B", "C"], "qty": [50, 20, 80], "avg_price": ["₹100", "₹250", "₹400"] }
```

Then change the index to:

```
["s1", "s2", "s3"]
```

Display the DataFrame.

```
In [2]: portfolio_df = pd.DataFrame(
    {
        "stock": ["A", "B", "C"],
        "qty": [50, 20, 80],
        "avg_price": ["₹100", "₹250", "₹400"]
    },
    index = ["s1", "s2", "s3"]
)

print(portfolio_df)
```

	stock	qty	avg_price
s1	A	50	₹100
s2	B	20	₹250
s3	C	80	₹400

Question #3 – Rearranging Columns

Using the DataFrame created in Question 2:

Reorder columns to the sequence:

```
["avg_price", "qty", "stock"]
```

Print the updated DataFrame.

```
In [4]: portfolio_df = portfolio_df[["avg_price", "qty", "stock"]]
print(portfolio_df)
```

	avg_price	qty	stock
s1	₹100	50	A
s2	₹250	20	B
s3	₹400	80	C

Question #4 – Selecting a Column

Using the modified `portfolio_df`, print only the column:

```
"qty"
```

```
In [5]: print(portfolio_df["qty"])
```

```
s1    50
s2    20
s3    80
Name: qty, dtype: int64
```

Question #5 – Download OHLCV Data and Rename Columns

Download historical data for any stock using:

```
yf.download("INFY.NS", start="2015-01-01", end="2024-12-31")
```

Rename the columns to:

```
["close", "high", "low", "open", "volume"]
```

Rearrange them to:

```
["open", "high", "low", "close", "volume"]
```

Show the first 5 rows.

In [27]: `import yfinance as yf`

```
df_INFY = yf.download("INFY.NS", start="2015-01-01", end="2024-12-31")
print(df_INFY.head())
```

```
C:\Users\Kedar\AppData\Local\Temp\ipykernel_26084\3864344414.py:3: FutureWarning: YF.download() has changed argument auto_adjust default to True
  df_INFY = yf.download("INFY.NS", start="2015-01-01", end="2024-12-31")
[*****100%*****] 1 of 1 completed
```

Price Ticker Date	Close INFY.NS	High INFY.NS	Low INFY.NS	Open INFY.NS	Volume INFY.NS
2015-01-01	367.770782	369.186427	364.511064	366.755599	2002764
2015-01-02	374.998077	376.087773	367.323776	367.323776	6778320
2015-01-05	371.775665	378.127457	368.348299	374.383441	9937024
2015-01-06	364.008209	369.745325	360.264193	368.813976	9667316
2015-01-07	365.749786	367.835997	363.225823	366.019868	7249916

In [30]: `df_INFY.columns = ['close', 'high', 'low', 'open', 'volume']`

```
df_INFY = df_INFY[['open', "high", "low", "close", "volume"]]
print(df_INFY.head())
```

Date	open	high	low	close	volume
2015-01-01	366.755599	369.186427	364.511064	367.770782	2002764
2015-01-02	367.323776	376.087773	367.323776	374.998077	6778320
2015-01-05	374.383441	378.127457	368.348299	371.775665	9937024
2015-01-06	368.813976	369.745325	360.264193	364.008209	9667316
2015-01-07	366.019868	367.835997	363.225823	365.749786	7249916

Question #6 – Dropping Columns and Rounding Values

Using the OHLCV DataFrame:

Drop the columns "high", "low", and "open".

Round the DataFrame values to 2 decimals using `np.round()`.

Print the first 4 rows.

In [33]: `import numpy as np`

```
df2 = df_INFY.drop(columns=["high", "low", "open"])
df2 = df2.round(2)

print(df2.head(4))
```

Date	close	volume
2015-01-01	367.77	2002764
2015-01-02	375.00	6778320
2015-01-05	371.78	9937024
2015-01-06	364.01	9667316

Question #7 – Dropping Multiple Rows

Drop the first 3 rows of the rounded DataFrame using:

```
df.drop(df.index[[0,1,2]])
```

Print the output.

```
In [34]: df3 = df2.drop(df2.index[[0,1,2]])  
print(df3)
```

```
          close      volume  
Date  
2015-01-06  364.01  9667316  
2015-01-07  365.75  7249916  
2015-01-08  367.59  13564920  
2015-01-09  386.41  44863328  
2015-01-12  394.14  12758888  
...       ...      ...  
2024-12-23  1868.99  2781793  
2024-12-24  1854.18  2360544  
2024-12-26  1852.57  3623321  
2024-12-27  1861.66  3937500  
2024-12-30  1851.21  7789055
```

[2463 rows x 2 columns]

Question #8 – Renaming Columns

Rename the columns:

```
{"close" : "Close Price", "volume" : "Total Volume"}
```

Print the last 5 rows.

```
In [35]: df4 = df3.rename(columns={  
    "close" : "Close Price",  
    "volume" : "Total Volume"  
})  
print(df4.tail())
```

```
          Close Price  Total Volume  
Date  
2024-12-23        1868.99        2781793  
2024-12-24        1854.18        2360544  
2024-12-26        1852.57        3623321  
2024-12-27        1861.66        3937500  
2024-12-30        1851.21        7789055
```

Question #9 – Sorting a DataFrame

Sort the DataFrame by the column "**Close Price**" in descending order.

Print the top 15 values.

```
In [36]: sorted_df = df4.sort_values("Close Price", ascending = False)
print(sorted_df.head(15))
```

Date	Close Price	Total Volume
2024-12-13	1942.22	5362693
2024-12-12	1929.89	6462130
2024-12-16	1923.14	3119221
2024-12-18	1922.26	3064095
2024-12-17	1919.79	5811050
2024-12-11	1917.41	5025612
2024-12-10	1892.54	6174921
2024-12-19	1890.26	5323592
2024-10-17	1890.01	8162822
2024-09-02	1886.55	5573739
2024-10-15	1881.56	5353705
2024-10-14	1881.17	3751378
2024-12-05	1879.24	9050038
2024-10-09	1875.27	4857163
2024-09-17	1875.08	2793127

Question #10 – Process Intraday Data

Download 5-minute data using:

```
df_intra = yf.download("^NSEI", interval="5m")
```

Perform the steps:

1. Rename columns to OHLCV order
2. Drop the "volume" column
3. Convert index to datetime:

```
df.index = pd.to_datetime(df.index)
```
4. Remove timezone:

```
df.index = df.index.tz_localize(None)
```
5. Format index to:

```
"%Y-%m-%d %I:%M:%S"
```

Print the first 5 rows.

```
In [39]: df_intra = yf.download("^NSEI", interval="5m")
print(df_intra.head())
```

```
C:\Users\Kedar\AppData\Local\Temp\ipykernel_26084\3563804825.py:1: FutureWarning: YF.download() has changed argument auto_adjust default to True
  df_intra = yf.download("^NSEI", interval="5m")
[*****100*****] 1 of 1 completed
```

Price Ticker Datetime	Close ^NSEI	High ^NSEI	Low ^NSEI	\
2025-11-11 09:15:00+00:00	25667.349609	25673.050781	25660.750000	
2025-11-11 09:20:00+00:00	25681.599609	25693.199219	25666.800781	
2025-11-11 09:25:00+00:00	25676.550781	25683.449219	25670.750000	
2025-11-11 09:30:00+00:00	25693.849609	25702.150391	25677.949219	
2025-11-11 09:35:00+00:00	25681.349609	25700.650391	25676.500000	

Price Ticker Datetime	Open ^NSEI	Volume ^NSEI	
2025-11-11 09:15:00+00:00	25662.849609	0	
2025-11-11 09:20:00+00:00	25667.800781	0	
2025-11-11 09:25:00+00:00	25680.800781	0	
2025-11-11 09:30:00+00:00	25677.949219	0	
2025-11-11 09:35:00+00:00	25692.849609	0	

```
In [45]: #Rename columns to OHLCV order
df_intra.columns =['close', 'high', 'low', 'open', 'volume']

df_intra = df_intra[["open", "high", "low", "close", "volume"]]

print(df_intra.head())
```

Datetime	open	high	low	\
2025-11-11 09:15:00+00:00	25662.849609	25673.050781	25660.750000	
2025-11-11 09:20:00+00:00	25667.800781	25693.199219	25666.800781	
2025-11-11 09:25:00+00:00	25680.800781	25683.449219	25670.750000	
2025-11-11 09:30:00+00:00	25677.949219	25702.150391	25677.949219	
2025-11-11 09:35:00+00:00	25692.849609	25700.650391	25676.500000	

Datetime	close	volume	
2025-11-11 09:15:00+00:00	25667.349609	0	
2025-11-11 09:20:00+00:00	25681.599609	0	
2025-11-11 09:25:00+00:00	25676.550781	0	
2025-11-11 09:30:00+00:00	25693.849609	0	
2025-11-11 09:35:00+00:00	25681.349609	0	

In [47]: `# Drop the "volume" column`

```
df_intra = df_intra.drop(columns= ["volume"])
print(df_intra.head())
```

Datetime	open	high	low	\
2025-11-11 09:15:00+00:00	25662.849609	25673.050781	25660.750000	
2025-11-11 09:20:00+00:00	25667.800781	25693.199219	25666.800781	
2025-11-11 09:25:00+00:00	25680.800781	25683.449219	25670.750000	
2025-11-11 09:30:00+00:00	25677.949219	25702.150391	25677.949219	
2025-11-11 09:35:00+00:00	25692.849609	25700.650391	25676.500000	

Datetime	close
2025-11-11 09:15:00+00:00	25667.349609
2025-11-11 09:20:00+00:00	25681.599609
2025-11-11 09:25:00+00:00	25676.550781
2025-11-11 09:30:00+00:00	25693.849609
2025-11-11 09:35:00+00:00	25681.349609

In [50]: `#Convert index to datetime:`

```
df_intra.index = pd.to_datetime(df_intra.index)
print(df_intra.head())
```

Datetime	open	high	low	\
2025-11-11 09:15:00+00:00	25662.849609	25673.050781	25660.750000	
2025-11-11 09:20:00+00:00	25667.800781	25693.199219	25666.800781	
2025-11-11 09:25:00+00:00	25680.800781	25683.449219	25670.750000	
2025-11-11 09:30:00+00:00	25677.949219	25702.150391	25677.949219	
2025-11-11 09:35:00+00:00	25692.849609	25700.650391	25676.500000	

Datetime	close
2025-11-11 09:15:00+00:00	25667.349609
2025-11-11 09:20:00+00:00	25681.599609
2025-11-11 09:25:00+00:00	25676.550781
2025-11-11 09:30:00+00:00	25693.849609
2025-11-11 09:35:00+00:00	25681.349609

In [51]: `#Remove timezone:`

```
df_intra.index = df_intra.index.tz_localize(None)
print(df_intra.head())
```

Datetime	open	high	low	close
2025-11-11 09:15:00	25662.849609	25673.050781	25660.750000	25667.349609
2025-11-11 09:20:00	25667.800781	25693.199219	25666.800781	25681.599609
2025-11-11 09:25:00	25680.800781	25683.449219	25670.750000	25676.550781
2025-11-11 09:30:00	25677.949219	25702.150391	25677.949219	25693.849609
2025-11-11 09:35:00	25692.849609	25700.650391	25676.500000	25681.349609

```
In [52]: # Format index to:  
df_intra.index = df_intra.index.strftime("%Y-%m-%d %I:%M:%S")  
  
print(df_intra.head())
```

Datetime	open	high	low	close
2025-11-11 09:15:00	25662.849609	25673.050781	25660.750000	25667.349609
2025-11-11 09:20:00	25667.800781	25693.199219	25666.800781	25681.599609
2025-11-11 09:25:00	25680.800781	25683.449219	25670.750000	25676.550781
2025-11-11 09:30:00	25677.949219	25702.150391	25677.949219	25693.849609
2025-11-11 09:35:00	25692.849609	25700.650391	25676.500000	25681.349609

```
In [ ]:
```

TTP Assignment 11

Pandas Statistical Functions

```
In [1]: import pandas as pd
import numpy as np
import yfinance as yf
import warnings
warnings.filterwarnings('ignore')
```

```
In [2]: data = yf.download("AUROPHARMA.NS", start='2015-01-01', end='2020-12-31')

[*****100%*****] 1 of 1 completed
```

```
In [3]: data.head()
```

	Price	Close	High	Low	Open
Ticker	AUROPHARMA.NS	AUROPHARMA.NS	AUROPHARMA.NS	AUROPHARMA.NS	AUROPHARMA.NS
Date					
2015-01-01	532.220764	538.028893	530.379158	538.028893	
2015-01-02	534.770813	536.329109	530.757081	531.229284	
2015-01-05	534.510986	537.981730	527.427930	534.581828	1
2015-01-06	513.663208	532.338830	509.979996	530.284755	1
2015-01-07	521.808716	528.372301	518.054662	521.076777	1

```
In [4]: # Rename columns
data.columns = ['close', 'high', 'low', 'open', 'volume']
```

```
In [5]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1479 entries, 2015-01-01 to 2020-12-30
Data columns (total 5 columns):
 #   Column   Non-Null Count  Dtype  
--- 
 0   close    1479 non-null   float64
 1   high     1479 non-null   float64
 2   low      1479 non-null   float64
 3   open     1479 non-null   float64
 4   volume   1479 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 69.3 KB
```

In [6]: `data.head()`

Out[6]:

	close	high	low	open	volume
Date					
2015-01-01	532.220764	538.028893	530.379158	538.028893	444720
2015-01-02	534.770813	536.329109	530.757081	531.229284	608132
2015-01-05	534.510986	537.981730	527.427930	534.581828	1056146
2015-01-06	513.663208	532.338830	509.979996	530.284755	1383922
2015-01-07	521.808716	528.372301	518.054662	521.076777	1366968

Rearrange Columns

In [7]: `data = data[['open', 'high', 'low', 'close', 'volume']]`

In [8]: `data.head(2)`

Out[8]:

	open	high	low	close	volume
Date					
2015-01-01	538.028893	538.028893	530.379158	532.220764	444720
2015-01-02	531.229284	536.329109	530.757081	534.770813	608132

In [9]: `df = data.copy()`

In [10]: `df = df.drop(['low'], axis=1)`

In [11]: `df.head(2)`

Out[11]:

	open	high	close	volume
Date				
2015-01-01	538.028893	538.028893	532.220764	444720
2015-01-02	531.229284	536.329109	534.770813	608132

In [12]: `df.info()`

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1479 entries, 2015-01-01 to 2020-12-30
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype  
---  --  
 0   open    1479 non-null   float64 
 1   high    1479 non-null   float64 
 2   close   1479 non-null   float64 
 3   volume  1479 non-null   int64  
dtypes: float64(3), int64(1)
memory usage: 57.8 KB
```

In [13]: `#df = np.round(df, 2)`

In [14]: `df.head()`

Out[14]:

	open	high	close	volume
Date				
2015-01-01	538.028893	538.028893	532.220764	444720
2015-01-02	531.229284	536.329109	534.770813	608132
2015-01-05	534.581828	537.981730	534.510986	1056146
2015-01-06	530.284755	532.338830	513.663208	1383922
2015-01-07	521.076777	528.372301	521.808716	1366968

DataFrame.count()

This method returns the number of non-null observations over the requested observations.

In [15]: `df.count()`

Out[15]:

open	1479
high	1479
close	1479
volume	1479
dtype:	int64

To find non-null observations or number of rows in a particular column then specify it as done below.

In [16]: `df['close'].count()`

Out[16]: 1479

DataFrame.min()

This method returns the minimum value over the requested observations.

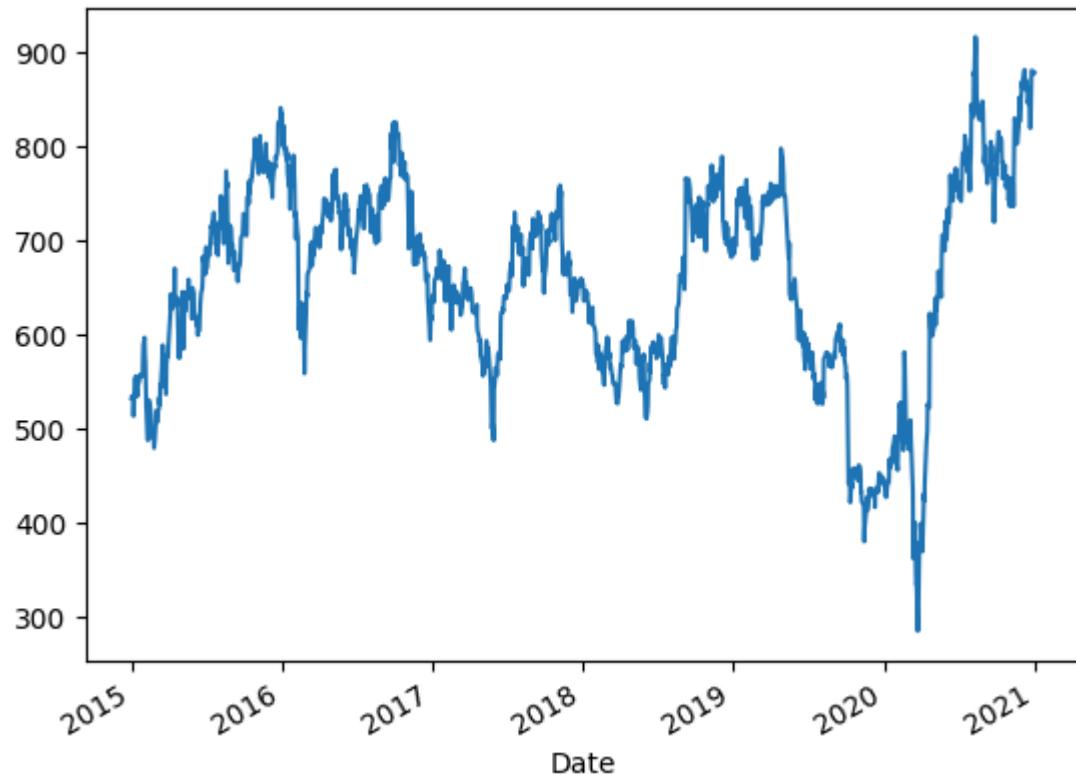
Imp Note: All these functions are applied to the entire dataframe and not just a single column.

In [17]: `df.min()`

Out[17]:

open	305.374141
high	327.938960
close	284.790375
volume	208624.000000
dtype:	float64

```
In [18]: import matplotlib.pyplot as plt
df['close'].plot();
```



DataFrame.max()

This method returns the maximum value over the requested observations.

```
In [19]: df['open'].max()
```

```
Out[19]: 927.7189240576777
```

For large numbers Pandas can output scientific notation, to avoid this we change the display format to show normal 2 decimal numbers

```
In [20]: pd.set_option('display.float_format', '{:.2f}'.format)
```

```
In [21]: print(df.max())
```

open	927.72
high	934.97
close	916.94
volume	37163094.00
dtype:	float64

DataFrame.mean()

The mean of a set of observations is the arithmetic average of the values.

This method returns the mean of the requested observations.

```
In [22]: df.mean()
```

```
Out[22]: open      660.83
          high      671.09
          close     659.48
          volume   3019356.79
          dtype: float64
```

DataFrame.median()

The median is a statistical measure that determines the middle value of a dataset listed in ascending order.

This method returns the median of the requested observations.

```
In [23]: df['close'].median()
```

```
Out[23]: 673.66748046875
```

DataFrame.mode()

The mode is the value that appears most frequently in a data set.

This method returns all the closing prices that occurred more than once.

```
In [24]: df['close'].mode()
```

```
Out[24]: 0    432.18
         1    567.68
         2    568.90
         3    573.54
         4    574.07
         5    579.88
         6    580.30
         7    582.54
         8    587.62
         9    592.36
        10   597.43
        11   602.30
        12   626.20
        13   631.47
        14   653.45
        15   670.78
        16   697.13
        17   711.67
        18   724.73
        19   730.91
        20   745.75
        21   746.55
        22   747.75
        23   758.60
        24   761.94
        25   763.52
        26   781.34
        27   784.68
        28   793.15
Name: close, dtype: float64
```

"In financial markets, the mode of prices is usually meaningless because they seldom repeat.

When Pandas returns many modes, it simply means no single price occurred frequently enough to matter."

In [25]: `df.head()`

Out[25]:

	open	high	close	volume
Date				
2015-01-01	538.03	538.03	532.22	444720
2015-01-02	531.23	536.33	534.77	608132
2015-01-05	534.58	537.98	534.51	1056146
2015-01-06	530.28	532.34	513.66	1383922
2015-01-07	521.08	528.37	521.81	1366968

DataFrame.sum()

This method returns the sum of all the values of the requested observations.

In [26]: `df['volume'].sum()`

Out[26]: 4465628686

DataFrame.diff()

This method returns the 'difference' between the current observation and the previous observation.

In [27]: `df['close'].diff()`

Out[27]:

Date	
2015-01-01	NaN
2015-01-02	2.55
2015-01-05	-0.26
2015-01-06	-20.85
2015-01-07	8.15
	...
2020-12-23	31.26
2020-12-24	19.09
2020-12-28	-0.63
2020-12-29	-3.20
2020-12-30	1.65

Name: close, Length: 1479, dtype: float64

DataFrame.pct_change()

This method returns the percentage change of the current observation with the previous observation.

In [28]: `df['close'].pct_change()`

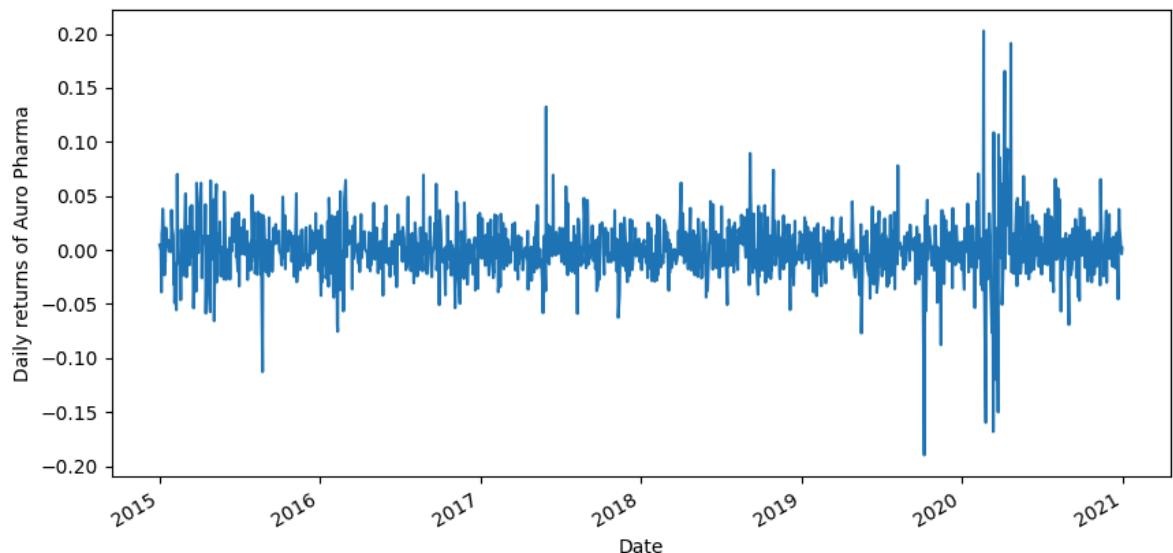
Out[28]: Date

Date	close
2015-01-01	NaN
2015-01-02	0.00
2015-01-05	-0.00
2015-01-06	-0.04
2015-01-07	0.02
...	
2020-12-23	0.04
2020-12-24	0.02
2020-12-28	-0.00
2020-12-29	-0.00
2020-12-30	0.00

Name: close, Length: 1479, dtype: float64

In [29]: `import matplotlib.pyplot as plt
%matplotlib inline`

```
plt.figure(figsize=(10, 5))  
plt.ylabel('Daily returns of Auro Pharma')  
df["close"].pct_change().plot()  
plt.show()
```



DataFrame.var()

Variance is a statistical measurement of the spread between numbers in a data set.

This method returns the variance of all values in that column

In [30]: `#How much do the open prices move around the overall average open price across
df['open'].var()`

Out[30]: 11342.750786750321

DataFrame.std()

Standard deviation is a statistical measure that measures the dispersion of a dataset relative to its mean.

This method returns the standard deviation of the requested observations.

```
In [31]: df['close'].std()
```

```
Out[31]: 106.04832182239332
```

```
In [32]: df['pct_change'] = df['close'].pct_change()
```

```
In [33]: df['pct_change'].std()
```

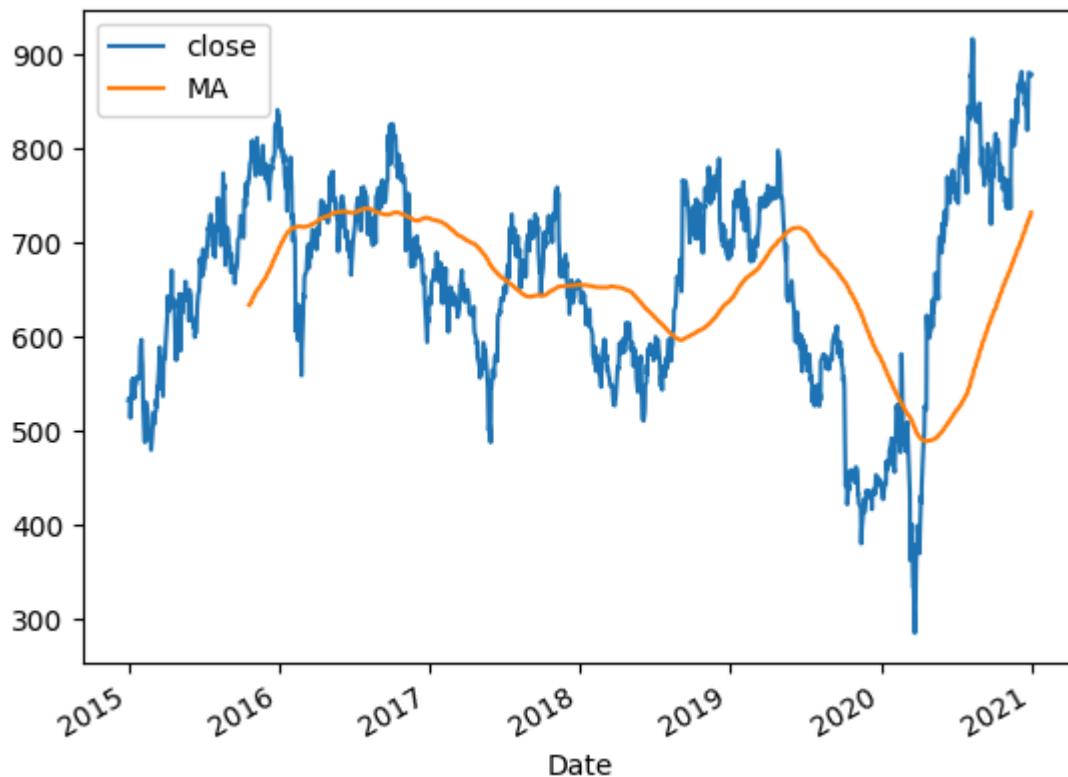
```
Out[33]: 0.025792282991251442
```

DataFrame.rolling(window=).mean()

A moving average(also known as rolling average) is a calculation used to analyze data points by creating a series of averages of different subsets of the full data set. It is commonly used as a technical indicator.

```
In [34]: # The moving average window is 200 in this case  
df['MA'] = df['close'].rolling(window=200).mean()
```

```
In [35]: df[['close', 'MA']].plot();
```



```
In [36]: df1 = yf.download('INFY.NS', start='2015-01-01', end='2020-12-31')
```

```
[*****100%*****] 1 of 1 completed
```

In [37]: df1.head()

Out[37]:

	Price	Close	High	Low	Open	Volume
Ticker	INFY.NS	INFY.NS	INFY.NS	INFY.NS	INFY.NS	INFY.NS
Date						
2015-01-01	367.77	369.19	364.51	366.76	2002764	
2015-01-02	375.00	376.09	367.32	367.32	6778320	
2015-01-05	371.78	378.13	368.35	374.38	9937024	
2015-01-06	364.01	369.75	360.26	368.81	9667316	
2015-01-07	365.75	367.84	363.23	366.02	7249916	

In [38]: # Rename columns
df1.columns = ['close', 'high', 'low', 'open', 'volume']

In [39]: df1.head()

Out[39]:

	close	high	low	open	volume
Date					
2015-01-01	367.77	369.19	364.51	366.76	2002764
2015-01-02	375.00	376.09	367.32	367.32	6778320
2015-01-05	371.78	378.13	368.35	374.38	9937024
2015-01-06	364.01	369.75	360.26	368.81	9667316
2015-01-07	365.75	367.84	363.23	366.02	7249916

In [40]: df2 = yf.download('^NSEI', start='2015-01-01', end='2020-12-31')
df2.columns = ['close', 'high', 'low', 'open', 'volume']

[*****100%*****] 1 of 1 completed

DataFrame.corr()

Correlation is a statistical measure that expresses the extent to which two variables are linearly related.

This method returns the correlation between the closing price of the nifty with the closing price of Infosys

In [41]: df2['close'].corr(df1['close'])

Out[41]: 0.7506280096886409

A correlation of 0.75 indicates a strong correlation between Nifty and Infosys

In [42]: df1['infy_pct_change'] = df1['close'].pct_change()
df2['nifty_pct_change'] = df2['close'].pct_change()

DataFrame.kurt()

Kurtosis is a statistical measure that defines how heavily the tails of a distribution differ from the tails of a normal distribution. This method returns unbiased kurtosis over the requested data set using Fisher's definition of kurtosis (where kurtosis of normal distribution = 0).

```
In [43]: print("Kurtosis of Infosys is:", df1['close'].kurt())
```

Kurtosis of Infosys is: 1.8713081252005828

```
In [44]: print("Kurtosis of Infosys is:", df1['infy_pct_change'].kurt())
```

Kurtosis of Infosys is: 10.710706588944234

Infosys has a higher tendency for extreme return spikes compared to Nifty.

```
In [45]: print("Kurtosis of Nifty(Index) is:", df2['close'].kurt())
```

Kurtosis of Nifty(Index) is: -1.0086219165060133

```
In [46]: print("Kurtosis of Nifty is:", df2['nifty_pct_change'].kurt())
```

Kurtosis of Nifty is: 20.545652043836434

Index here is more stable (in this period) also because it is diversified

DataFrame.skew()

In statistics, skewness is a measure of the asymmetry of the distribution of a variable about its mean. This method gives unbiased skew over the requested data set.

```
In [47]: df1['close'].skew()
```

```
Out[47]: 1.41941411816345
```

```
In [48]: df2['close'].skew()
```

```
Out[48]: 0.10736482778954834
```

```
In [49]: #df1['infy_pct_change'] = df1['close'].pct_change()
#df2['nifty_pct_change'] = df2['close'].pct_change()
```

```
In [50]: print("Infosys pct_change Skew", df1['infy_pct_change'].skew())
print("Nifty pct change Skew", df2['nifty_pct_change'].skew())
```

Infosys pct_change Skew -0.32363593637320054
Nifty pct change Skew -1.299790949445222

Infosys returns are mildly negatively skewed, but index returns show strong negative skew because market crashes create deep left tails.

DataFrame.describe()

`DataFrame.describe()` generates a summary of the most important descriptive statistics for each numerical column in a DataFrame.

It helps you quickly understand:

- central tendency
- spread
- shape of the distribution
- presence of extreme values

When applied, it returns a table containing:

Statistic	Meaning
count	Number of non-missing observations
mean	Average value
std	Standard deviation (measure of spread)
min	Smallest value
25%	First quartile (25th percentile)
50%	Median (50th percentile)
75%	Third quartile (75th percentile)
max	Largest value

```
In [51]: print("Nifty's Descriptive Statistics")
df2.describe()
```

Nifty's Descriptive Statistics

	close	high	low	open	volume	nifty_pct_change
count	1471.00	1471.00	1471.00	1471.00	1471.00	1470.00
mean	9882.65	9939.06	9825.60	9892.31	325472.33	0.00
std	1508.84	1511.50	1503.58	1510.11	226241.35	0.01
min	6970.60	7034.20	6825.80	7023.65	0.00	-0.13
25%	8515.45	8557.83	8450.25	8508.25	171400.00	-0.00
50%	10079.30	10126.50	10021.45	10074.80	229700.00	0.00
75%	11068.43	11117.72	10999.70	11069.90	445350.00	0.01
max	13981.95	13997.00	13864.95	13980.90	1811000.00	0.09

25% 1st quartile, 25% of the values below this

50% 2nd quartile, 50% of the values below this

75% 3rd quartile, 75% of the values below this

```
In [52]: print("Infosys Descriptive Statistics")
df1.describe()
```

Infosys Descriptive Statistics

	close	high	low	open	volume	infy_pct_change
count	1479.00	1479.00	1479.00	1479.00	1479.00	1478.00
mean	517.89	523.82	512.09	517.99	8902310.89	0.00
std	155.41	157.26	153.45	155.12	7910049.18	0.02
min	346.07	351.70	340.76	348.68	758956.00	-0.16
25%	399.93	403.56	396.11	400.16	5503889.00	-0.01
50%	459.45	463.34	454.14	459.08	7180728.00	0.00
75%	608.91	614.18	601.72	608.15	9922791.00	0.01
max	1103.09	1108.19	1089.97	1103.04	164404960.00	0.12

Exponential Moving Average

An Exponential Moving Average is a type of moving average that gives more weight to recent data points, making it respond faster to new price information. It is widely used as a technical indicator to identify trends, momentum shifts, and dynamic support or resistance levels.

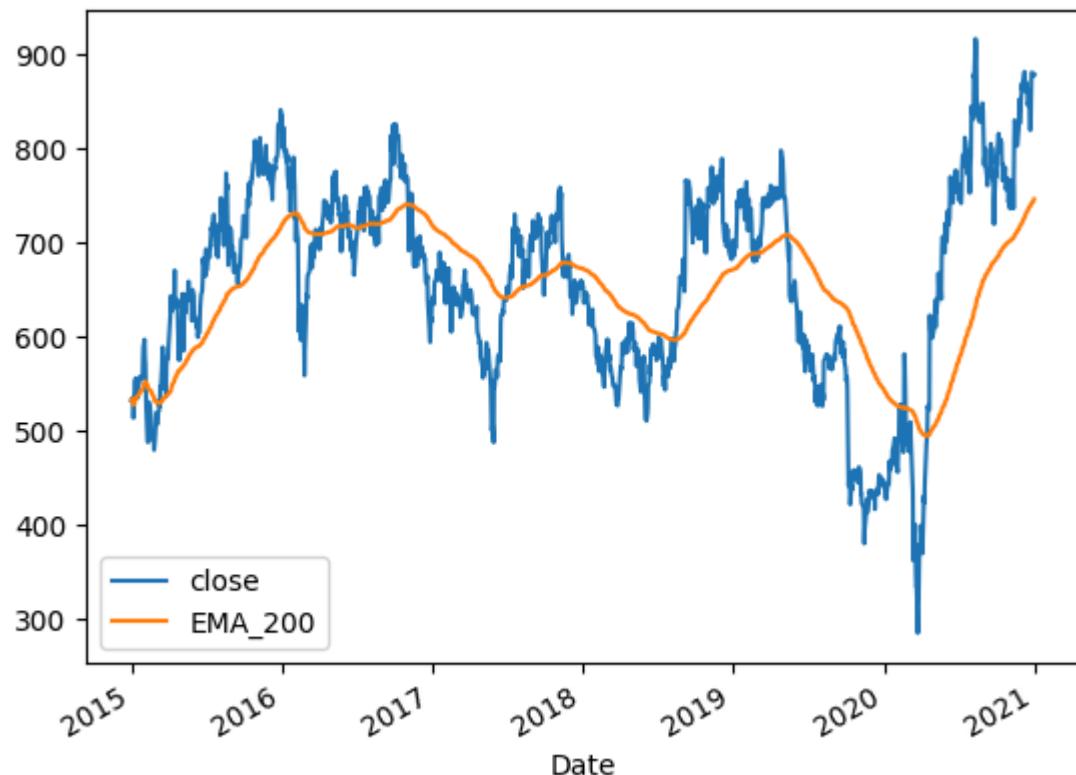
`DataFrame.ewm(span=200).mean()`

```
In [53]: df2 = data.copy()
df2['EMA_200'] = df2['close'].ewm(span=200).mean() # Exponential MA
```

```
In [54]: df2.tail()
```

Date	open	high	low	close	volume	EMA_200
2020-12-23	832.58	864.56	824.82	861.94	2011957	740.95
2020-12-24	874.74	883.95	851.23	881.04	4176663	742.35
2020-12-28	891.51	892.67	876.19	880.41	3136265	743.72
2020-12-29	884.82	891.07	868.68	877.21	1920734	745.05
2020-12-30	878.13	881.04	864.71	878.86	1667050	746.38

```
In [55]: df2[['close', 'EMA_200']].plot();
```



Class Exercise

1. Download EOD data of HDFCBANK.NS

Period: 1 Jan 2020 to 31 Dec 2024

Save it in `df4`

2. Download EOD data of ^NSEBANK

Same period

Save it in `df5`

3. Perform data sanity checks for both time-series dataframes

- Check for missing values if NaN
- Ensure Date is the datetimeindex

4. Calculate SMA and EMA for both

- SMA200
- EMA50

Add them as new columns.

5. Plot respective price charts with their MA and EMA

6. Calculate daily % change using pct_change()

Store in:

- df4['daily_pct_change']
- df5['daily_pct_change']

7. Calculate correlation of daily returns

Compute correlation between BankNifty and HDFC Bank:

```
In [56]: df4 = yf.download("HDFCBANK.NS", start="2020-01-01", end="2024-12-31")
df5 = yf.download("^NSEBANK",      start="2020-01-01", end="2024-12-31")
```

[*****100%*****] 1 of 1 completed
[*****100%*****] 1 of 1 completed

```
In [57]: # Check Date for missing values
print("HDFCBANK missing values:", df4.isnull().sum())
print("NSEBANK missing values:", df5.isnull().sum())
```

```
HDFCBANK missing values: Price Ticker
Close    HDFCBANK.NS    0
High     HDFCBANK.NS    0
Low      HDFCBANK.NS    0
Open     HDFCBANK.NS    0
Volume   HDFCBANK.NS    0
dtype: int64
NSEBANK missing values: Price Ticker
Close    ^NSEBANK    0
High     ^NSEBANK    0
Low      ^NSEBANK    0
Open     ^NSEBANK    0
Volume   ^NSEBANK    0
dtype: int64
```

```
In [58]: df4.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1237 entries, 2020-01-01 to 2024-12-30
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   (Close, HDFCBANK.NS)  1237 non-null   float64
 1   (High, HDFCBANK.NS)   1237 non-null   float64
 2   (Low, HDFCBANK.NS)    1237 non-null   float64
 3   (Open, HDFCBANK.NS)   1237 non-null   float64
 4   (Volume, HDFCBANK.NS) 1237 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 58.0 KB
```

In [59]: `df5.info()`

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1231 entries, 2020-01-02 to 2024-12-30
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   (Close, ^NSEBANK)    1231 non-null   float64
 1   (High, ^NSEBANK)     1231 non-null   float64
 2   (Low, ^NSEBANK)      1231 non-null   float64
 3   (Open, ^NSEBANK)     1231 non-null   float64
 4   (Volume, ^NSEBANK)   1231 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 57.7 KB
```

In [60]: `df4["SMA200"] = df4["Close"].rolling(200).mean()
df4["EMA50"] = df4["Close"].ewm(span=50, adjust=False).mean()`

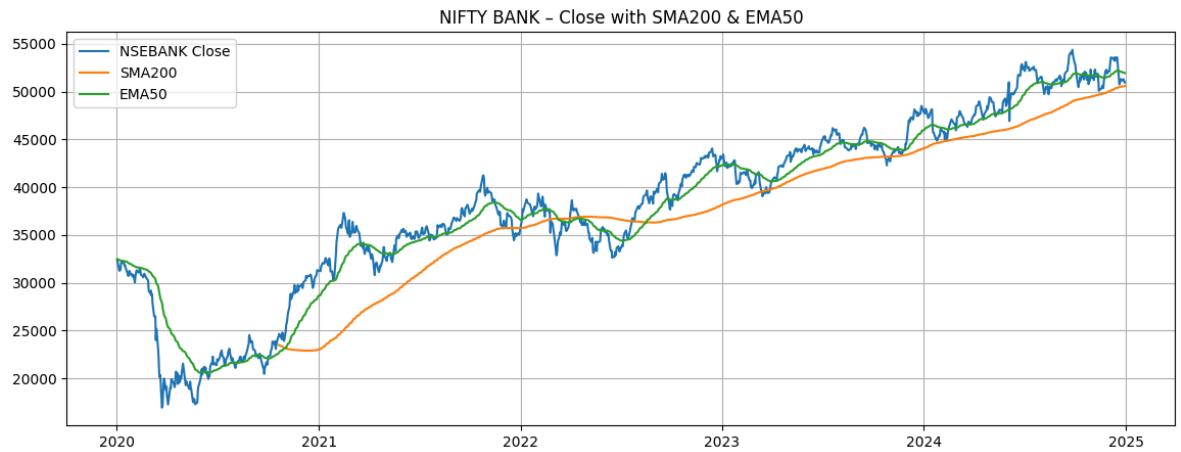
`df5["SMA200"] = df5["Close"].rolling(200).mean()
df5["EMA50"] = df5["Close"].ewm(span=50, adjust=False).mean()`

In [61]: `plt.figure(figsize=(14,5))
plt.plot(df4["Close"], label="HDFC Bank Close")
plt.plot(df4["SMA200"], label="SMA200")
plt.plot(df4["EMA50"], label="EMA50")
plt.title("HDFCBANK - Close with SMA200 & EMA50")
plt.legend()
plt.grid(True)
plt.show()`

`#df4[["Close", "SMA200", "EMA50"]].plot(figsize=(14,5), grid=True, title="HDFC`



```
In [62]: plt.figure(figsize=(14,5))
plt.plot(df5["Close"], label="NSEBANK Close")
plt.plot(df5["SMA200"], label="SMA200")
plt.plot(df5["EMA50"], label="EMA50")
plt.title("NIFTY BANK - Close with SMA200 & EMA50")
plt.legend()
plt.grid(True)
plt.show()
```



```
In [63]: df4["daily_pct_change"] = df4["Close"].pct_change()
df5["daily_pct_change"] = df5["Close"].pct_change()
```

```
In [64]: correlation = np.round(df4["daily_pct_change"].corr(df5["daily_pct_change"])), 2
print("Correlation (HDFCBANK vs NIFTY BANK) =", correlation)
```

Correlation (HDFCBANK vs NIFTY BANK) = 0.83

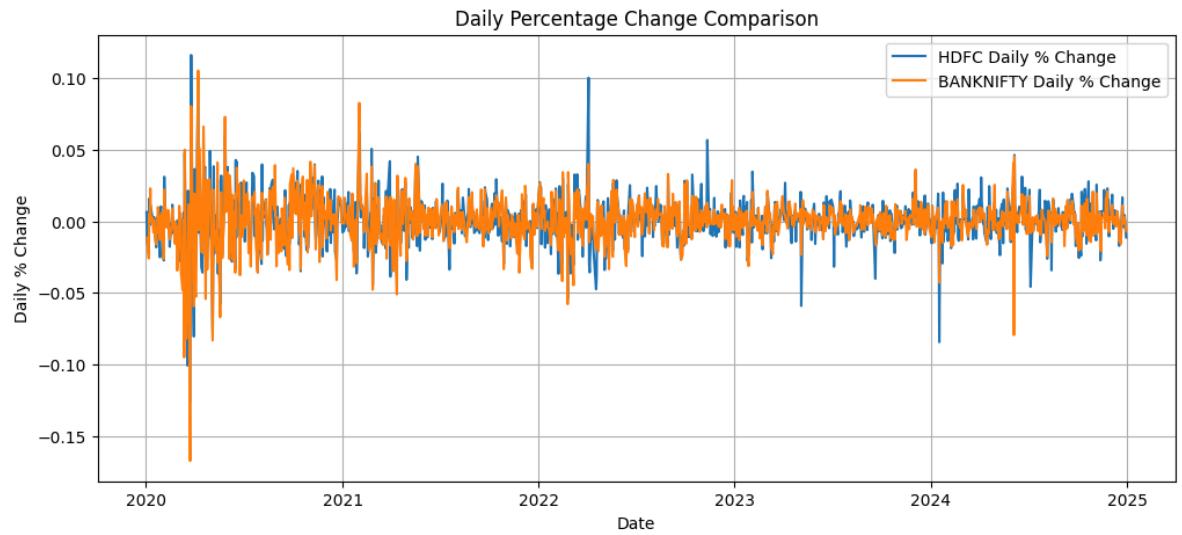
HDFC Bank moves in strong sync with Bank Nifty, so holding both gives highly overlapping exposure.

```
In [65]: plt.figure(figsize=(12,5))

plt.plot(df4.index, df4["daily_pct_change"], label="HDFC Daily % Change")
plt.plot(df5.index, df5["daily_pct_change"], label="BANKNIFTY Daily % Change")

plt.title("Daily Percentage Change Comparison")
plt.xlabel("Date")
plt.ylabel("Daily % Change")

plt.grid(True)
plt.legend()
plt.show()
```



HDFC Bank is slightly more volatile on individual days, while Bank Nifty is smoother and more stable due to sector diversification.

```
In [ ]:
```

TTP Assignment 12

Pandas Statistical Functions, Analysis & Plotting

Question #1 – Download and Prepare the Data

Download EOD price data for TECHM.NS and LT.NS for the period 1 January 2020 to 31 December 2024.

Convert the data into DateTimeIndex and store the cleaned dataframes as **df1** and **df2**.

```
In [1]: import yfinance as yf
import pandas as pd
df1= yf.download("TECHM.NS", start="2020-01-01", end="2024-12-31")
df2= yf.download("LT.NS", start="2020-01-01", end="2024-12-31")

df1.index = pd.to_datetime(df1.index)
df2.index = pd.to_datetime(df2.index)

print(df1.head())
print(df2.head())
```

```
C:\Users\Kedar\AppData\Local\Temp\ipykernel_24036\1244163521.py:3: FutureWarning: YF.download() has changed argument auto_adjust default to True
  df1= yf.download("TECHM.NS", start="2020-01-01", end="2024-12-31")
[*****100%*****] 1 of 1 completed
C:\Users\Kedar\AppData\Local\Temp\ipykernel_24036\1244163521.py:4: FutureWarning: YF.download() has changed argument auto_adjust default to True
  df2= yf.download("LT.NS", start="2020-01-01", end="2024-12-31")
[*****100%*****] 1 of 1 completed
```

Price	Close	High	Low	Open	Volume
Ticker	TECHM.NS	TECHM.NS	TECHM.NS	TECHM.NS	TECHM.NS
Date					
2020-01-01	602.617493	606.333947	601.233710	605.701370	746371
2020-01-02	605.740845	608.192145	601.747654	602.617441	1133488
2020-01-03	612.896973	616.692529	603.764018	605.701323	2121704
2020-01-06	609.180603	615.822730	607.401454	612.699326	1734439
2020-01-07	614.478455	615.585500	601.431363	608.864271	1937143
Price	Close	High	Low	Open	Volume
Ticker	LT.NS	LT.NS	LT.NS	LT.NS	LT.NS
Date					
2020-01-01	1192.206787	1200.352395	1185.881523	1190.796174	3123998
2020-01-02	1224.379395	1226.836664	1193.162364	1194.072480	4335359
2020-01-03	1215.050781	1224.060844	1210.591189	1224.060844	2059871
2020-01-06	1198.395386	1212.274656	1195.983556	1211.364540	2646905
2020-01-07	1201.808472	1219.100680	1195.665187	1208.634343	2077893

Question #2 – Perform Data Integrity Checks

Check both dataframes to confirm there are no missing values and no duplicate dates.
Verify that the index is correctly set to datetime.

```
In [2]: print("TECHM missing values:\n", df1.isnull().sum())
print("LT missing values:\n", df2.isnull().sum())

print("TECHM duplicate dates:\n", df1.index.duplicated().sum())
print("LT duplicate dates:\n", df2.index.duplicated().sum())

print(type(df1.index))
print(type(df2.index))
```

```
TECHM missing values:
  Price   Ticker
Close  TECHM.NS    0
High   TECHM.NS    0
Low    TECHM.NS    0
Open   TECHM.NS    0
Volume TECHM.NS    0
dtype: int64
LT missing values:
  Price   Ticker
Close  LT.NS      0
High   LT.NS      0
Low    LT.NS      0
Open   LT.NS      0
Volume LT.NS      0
dtype: int64
TECHM duplicate dates:
  0
LT duplicate dates:
  0
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
```

Question #3 – Compute Moving Averages

Create two new columns in each dataframe: a 100-day Simple Moving Average and a 50-day Exponential Moving Average.

```
In [3]: df1["MA100"] = df1["Close"].rolling(100).mean()
df1["EMA50"] = df1["Close"].ewm(span=50, adjust=False).mean()

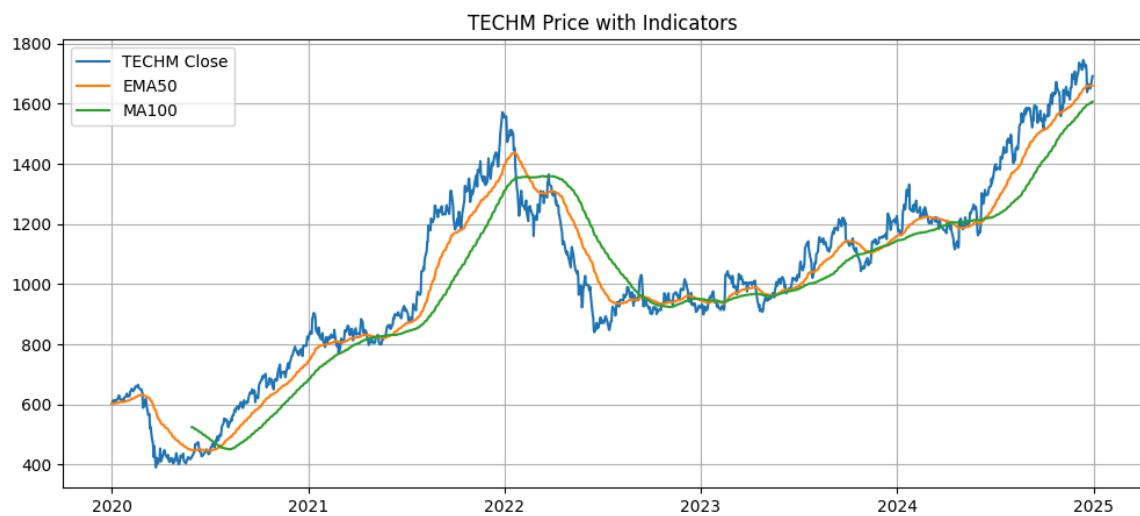
df2["MA100"] = df2["Close"].rolling(100).mean()
df2["EMA50"] = df2["Close"].ewm(span=50, adjust=False).mean()
```

Question #4 – Plot Price with Indicators

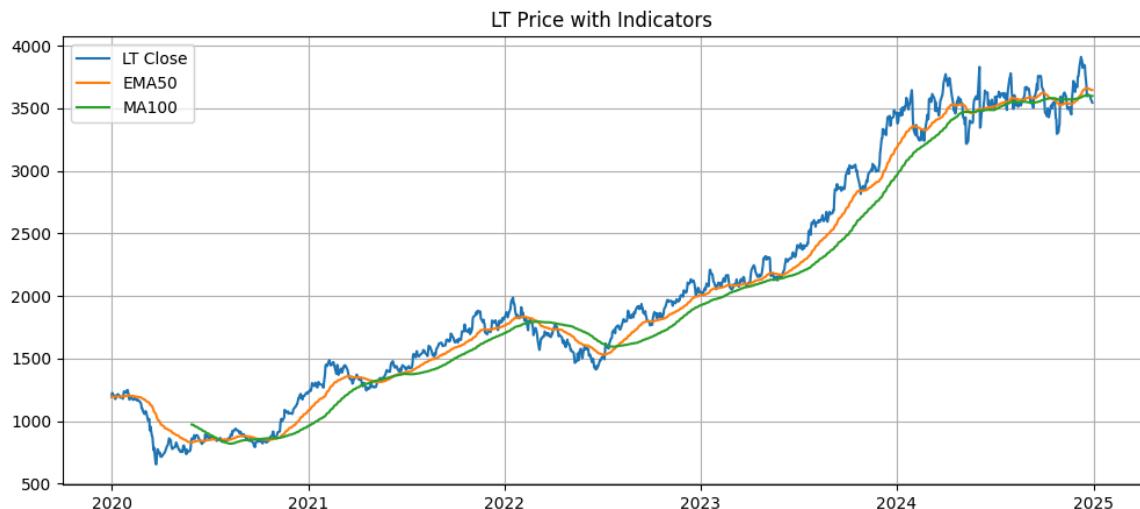
Plot the Closing Price, EMA50, and MA100 for TECHM and LT on separate charts using Matplotlib.

Ensure all lines are clearly labeled.

```
In [4]: import matplotlib.pyplot as plt  
#-----TECHM-----  
plt.figure(figsize=(12,5))  
plt.plot(df1["Close"], label="TECHM Close")  
plt.plot(df1["EMA50"], label="EMA50")  
plt.plot(df1["MA100"], label="MA100")  
plt.title("TECHM Price with Indicators")  
plt.legend()  
plt.grid(True)  
plt.show()
```



```
In [5]: #-----LT-----  
plt.figure(figsize=(12,5))  
plt.plot(df2["Close"], label="LT Close")  
plt.plot(df2["EMA50"], label="EMA50")  
plt.plot(df2["MA100"], label="MA100")  
plt.title("LT Price with Indicators")  
plt.legend()  
plt.grid(True)  
plt.show()
```



Question #5 – Calculate Daily Percentage Returns

Compute the daily percentage return for both stocks and store the results in a new column called `daily_pct_change` in each `Dataframe`.

```
In [6]: df1["daily_pct_change"] = df1["Close"].pct_change() * 100  
df2["daily_pct_change"] = df2["Close"].pct_change() * 100
```

Question #6 – Compute Statistical Properties

Calculate the skewness, kurtosis, and standard deviation of the daily percentage returns for both stocks.

```
In [7]: print("TECHM Stats:")  
print("skewness:",df1["daily_pct_change"].skew())  
print(" kurtosis:",df1["daily_pct_change"].kurtosis())  
print("std dev:",df1["daily_pct_change"].std())  
  
print("\n LT Stats:")  
print("skewness:",df2["daily_pct_change"].skew())  
print("kurtosis:",df2["daily_pct_change"].kurtosis())  
print("std dev:",df2["daily_pct_change"].std())
```

TECHM Stats:
skewness: -0.5900270235265466
kurtosis: 5.345300374289533
std dev: 1.9575273815756535

LT Stats:
skewness: -0.6654337371285358
kurtosis: 9.913641559062548
std dev: 1.8099705900520033

Question #7 – Compute Return Correlation

Calculate the correlation value between the daily percentage returns of TECHM and LT. Interpret whether the two stocks move together or independently.

```
In [8]: corr = df1["daily_pct_change"].corr(df2["daily_pct_change"])  
  
print("Correlation between TECHM nad LT returns=",corr)
```

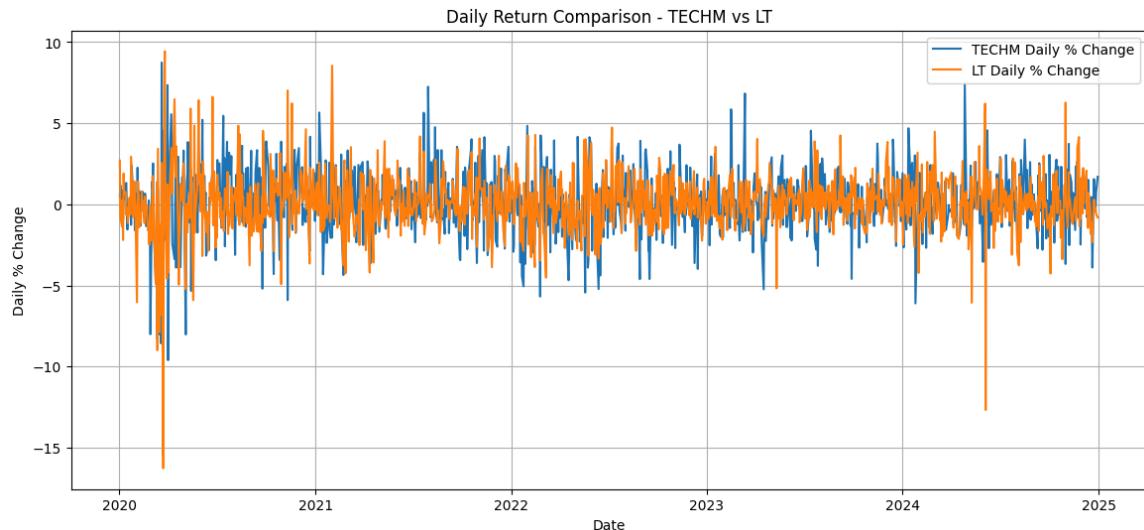
Correlation between TECHM nad LT returns= 0.35572970454261266

Question #8 – Plot Daily Return Movements

Plot the daily percentage return series of both TECHM and LT on the same Matplotlib chart to compare volatility patterns.

```
In [9]: plt.figure(figsize=(14,6))
```

```
plt.plot(df1.index, df1["daily_pct_change"], label="TECHM Daily % Change")
plt.plot(df2.index, df2["daily_pct_change"], label="LT Daily % Change")
plt.xlabel("Date")
plt.ylabel("Daily % Change")
plt.title("Daily Return Comparison - TECHM vs LT")
plt.grid()
plt.legend()
plt.show()
```



Question #9 – Identify Relative Volatility

Using the standard deviation values obtained earlier, determine which stock demonstrates higher volatility during the selected period.

```
In [10]: std_techm = df1["daily_pct_change"].std()
std_lt = df2["daily_pct_change"].std()

print("TECHM STD:", std_techm)
print("LT STD:", std_lt)

if std_techm > std_lt:
    print("TECHM is more volatile.")
else:
    print("LT is more volatile.)
```

TECHM STD: 1.9575273815756535

LT STD: 1.8099705900520033

TECHM is more volatile.

Question #10 – Interpret Skewness and Kurtosis

Based on the skewness and kurtosis values, describe the shape of the return distributions for TECHM and LT.

Explain whether the returns are symmetric, skewed, normal, or fat-tailed.

```
In [11]: techm_skew= df1["daily_pct_change"].skew()
techm_kurt = df1["daily_pct_change"].kurtosis()

lt_skew = df2["daily_pct_change"].skew()
lt_kurt = df2["daily_pct_change"].kurtosis()

print("TECHM Skewness:", techm_skew)
print("TECHM Kurtosis:", techm_kurt)
print("LT Skewness:", lt_skew)
print("LT Kurtosis:", lt_kurt)
```

TECHM Skewness: -0.5900270235265466
TECHM Kurtosis: 5.345300374289533
LT Skewness: -0.6654337371285358
LT Kurtosis: 9.913641559062548

```
In [12]: def interpret_skewness(sk):
    if sk > 0:
        return "Right-skewed(more large positive returns)"
    elif sk < 0:
        return "Left-skewed(more large negative returns)"
    else:
        return "Symetric distribution"

def interpret_kurtosis(ku):
    if ku > 3:
        return "Fat-tailed(more extreme return events)"
    elif ku < 3:
        return "Thin-tailed(fewer extrime events)"
    else:
        return "Normal distribution"

print("\n INTERPRETATION")

print("\nTECHM:")
print("Skewness Interpretation:",interpret_skewness(techm_skew))
print("Kurtosis Interpretation:",interpret_kurtosis(techm_kurt))

print("\nLT:")
print("Skewness Interpretation:",interpret_skewness(lt_skew))
print("Kurtosis Interpretation:",interpret_kurtosis(lt_kurt))
```

INTERPRETATION

TECHM:

Skewness Interpretation: Left-skewed(more large negative returns)
Kurtosis Interpretation: Fat-tailed(more extreme return events)

LT:

Skewness Interpretation: Left-skewed(more large negative returns)
Kurtosis Interpretation: Fat-tailed(more extreme return events)

In []:

Pandas Indexing

Indexing using .iloc()

.iloc → Integer-location based indexing

Think: “I” for **Index number (position)**

Used to access rows and columns **strictly by numeric positions**, similar to Python list indexing.

Syntax:

```
df.iloc[row_index, column_index]
```

.iloc uses **zero-based indexing** and **excludes the stop value** (just like Python slicing).

Only **integer positions** are allowed – labels will not work here.

Useful when your DataFrame index is not a DatetimeIndex.

```
In [1]: import numpy as np  
import pandas as pd
```

```
ksm = pd.read_csv('stock_detailed.csv')
```

```
In [2]: ksm
```

Out[2]:

	Symbol	Series	Date	Prev Close	Open Price	High Price	Low Price	Last Price	Close Price	Average Price	Total Traded Quantity	No. of Trades
0	KSM	EQ	02-Apr-18	1131.80	1141.00	1149.55	1121.30	1136.70	1137.15	1135.86	4036351	142078
1	KSM	EQ	03-Apr-18	1137.15	1134.70	1143.55	1128.10	1139.40	1140.45	1135.21	2038584	114034
2	KSM	EQ	04-Apr-18	1140.45	1144.00	1144.55	1120.00	1122.70	1124.20	1131.81	2406651	137029
3	KSM	EQ	05-Apr-18	1124.20	1139.55	1151.30	1129.10	1146.05	1147.55	1140.71	3881772	101745
4	KSM	EQ	06-Apr-18	1147.55	1143.00	1146.00	1122.10	1127.55	1127.00	1128.81	2968871	137277
5	KSM	EQ	09-Apr-18	1127.00	1125.00	1125.80	1106.55	1111.45	1111.25	1113.12	3601441	171118
6	KSM	EQ	10-Apr-18	1111.25	1112.00	1124.50	1105.40	1114.65	1113.40	1115.77	4463029	144468
7	KSM	EQ	11-Apr-18	1113.40	1118.00	1131.50	1116.50	1124.50	1124.25	1124.78	4512787	102586
8	KSM	EQ	12-Apr-18	1124.25	1129.45	1172.75	1125.00	1164.05	1162.60	1157.71	8522183	216130
9	KSM	EQ	13-Apr-18	1162.60	1174.00	1185.90	1150.25	1168.00	1171.45	1172.37	10613519	180965

```
In [3]: # Using .iloc()  
# Select the first four rows of all the columns  
# '.iloc()' method DOES NOT include the rows and columns in its stop argument  
ksm.iloc[:4]
```

Out[3]:

	Symbol	Series	Date	Prev Close	Open Price	High Price	Low Price	Last Price	Close Price	Average Price	Total Traded Quantity	No. of Trades
0	KSM	EQ	02-Apr-18	1131.80	1141.00	1149.55	1121.3	1136.70	1137.15	1135.86	4036351	142078
1	KSM	EQ	03-Apr-18	1137.15	1134.70	1143.55	1128.1	1139.40	1140.45	1135.21	2038584	114034
2	KSM	EQ	04-Apr-18	1140.45	1144.00	1144.55	1120.0	1122.70	1124.20	1131.81	2406651	137029
3	KSM	EQ	05-Apr-18	1124.20	1139.55	1151.30	1129.1	1146.05	1147.55	1140.71	3881772	101745

```
In [4]: # Select the rows from index 1 to index 4 (4 rows in total) and Columns with index from 2 to 3 (2 columns)
# .iloc() is similar to numpy array indexing
# .iloc() is extremely useful when your data is not labelled and you need to refer to columns
# using their integer location instead
print(ksm.iloc[1:5, 2:4])
```

	Date	Prev Close
1	03-Apr-18	1137.15
2	04-Apr-18	1140.45
3	05-Apr-18	1124.20
4	06-Apr-18	1147.55

```
In [5]: # Selecting the exact requested rows and columns
```

```
print(ksm.iloc[[1, 3, 5, 7], [1, 3, 5, 7, 9]])
```

	Series	Prev Close	High Price	Last Price	Average Price
1	EQ	1137.15	1143.55	1139.40	1135.21
3	EQ	1124.20	1151.30	1146.05	1140.71
5	EQ	1127.00	1125.80	1111.45	1113.12
7	EQ	1113.40	1131.50	1124.50	1124.78

```
In [6]: # Selecting the first two rows and all the columns
```

```
print(ksm.iloc[1:3, :])
```

	Symbol	Series	Date	Prev Close	Open Price	High Price	Low Price	\
1	KSM	EQ	03-Apr-18	1137.15	1134.7	1143.55	1128.1	
2	KSM	EQ	04-Apr-18	1140.45	1144.0	1144.55	1120.0	

	Last Price	Close Price	Average Price	Total Traded	Quantity	\
1	1139.4	1140.45	1135.21		2038584	
2	1122.7	1124.20	1131.81		2406651	

	No. of Trades
1	114034
2	137029

```
In [7]: #selecting all rows and first two columns
print(ksm.iloc[:, 1:3])
```

	Series	Date
0	EQ	02-Apr-18
1	EQ	03-Apr-18
2	EQ	04-Apr-18
3	EQ	05-Apr-18
4	EQ	06-Apr-18
5	EQ	09-Apr-18
6	EQ	10-Apr-18
7	EQ	11-Apr-18
8	EQ	12-Apr-18
9	EQ	13-Apr-18

Indexing using .loc()

It is a label-location based indexer for selecting data points.

Think: “L” for Label (name)

Used to access rows and columns by **their labels** – index names or column names.

Syntax:

```
df.loc[row_label, column_label]
```

.loc() includes the rows and columns in its stop argument.

The .loc indexer takes **row arguments first** and **column arguments second**.

```
In [8]: import pandas as pd
import numpy as np
import yfinance as yf
import warnings
warnings.filterwarnings('ignore')
```

```
In [9]: data = yf.download("HDFCBANK.NS", start="2020-01-01", end="2024-12-31")
```

```
[*****100%*****] 1 of 1 completed
```

```
In [10]: data.head()
```

```
Out[10]:
```

Price	Close	High	Low	Open	Volume
Ticker	HDFCBANK.NS	HDFCBANK.NS	HDFCBANK.NS	HDFCBANK.NS	HDFCBANK.NS
Date					
2020-01-01	605.529358	606.192392	601.740655	604.345388	3673698
2020-01-02	609.389099	609.981084	605.718794	605.718794	6137166
2020-01-03	600.698792	608.560338	598.425547	607.234269	10855550
2020-01-06	587.698792	597.573144	585.354555	596.720663	10890186
2020-01-07	597.004761	602.143177	593.050314	596.199685	14724494

```
In [11]: data.shape
```

```
Out[11]: (1237, 5)
```

```
In [12]: data.columns = ['close', 'high', 'low', 'open', 'volume']
```

```
In [13]: data.head()
```

```
Out[13]:
```

	close	high	low	open	volume
Date					
2020-01-01	605.529358	606.192392	601.740655	604.345388	3673698
2020-01-02	609.389099	609.981084	605.718794	605.718794	6137166
2020-01-03	600.698792	608.560338	598.425547	607.234269	10855550
2020-01-06	587.698792	597.573144	585.354555	596.720663	10890186
2020-01-07	597.004761	602.143177	593.050314	596.199685	14724494

```
In [14]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1237 entries, 2020-01-01 to 2024-12-30
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   close    1237 non-null   float64
 1   high     1237 non-null   float64
 2   low      1237 non-null   float64
 3   open     1237 non-null   float64
 4   volume   1237 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 58.0 KB
```

```
In [15]: # Select all rows for a specific column
print(data.loc[:, ['close']])
```

```
close
Date
2020-01-01  605.529358
2020-01-02  609.389099
2020-01-03  600.698792
2020-01-06  587.698792
2020-01-07  597.004761
...
2024-12-23  888.490295
2024-12-24  887.059631
2024-12-26  883.433655
2024-12-27  887.133667
2024-12-30  877.094360
```

```
[1237 rows x 1 columns]
```

```
In [16]: # Select all the rows of these specific columns  
print(data.loc[:, ['close', 'volume']])
```

Date	close	volume
2020-01-01	605.529358	3673698
2020-01-02	609.389099	6137166
2020-01-03	600.698792	10855550
2020-01-06	587.698792	10890186
2020-01-07	597.004761	14724494
...
2024-12-23	888.490295	11044592
2024-12-24	887.059631	14485834
2024-12-26	883.433655	10481678
2024-12-27	887.133667	7259330
2024-12-30	877.094360	22222218

[1237 rows x 2 columns]

```
In [17]: # Select the first six rows of the specific columns  
data.loc[data.index[:6], ['close', 'open']]
```

```
Out[17]:
```

Date	close	open
2020-01-01	605.529358	604.345388
2020-01-02	609.389099	605.718794
2020-01-03	600.698792	607.234269
2020-01-06	587.698792	596.720663
2020-01-07	597.004761	596.199685
2020-01-08	595.441956	590.540276

```
In [18]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 1237 entries, 2020-01-01 to 2024-12-30  
Data columns (total 5 columns):  
 #   Column   Non-Null Count   Dtype     
---    
 0   close    1237 non-null    float64  
 1   high     1237 non-null    float64  
 2   low      1237 non-null    float64  
 3   open     1237 non-null    float64  
 4   volume   1237 non-null    int64  
dtypes: float64(4), int64(1)  
memory usage: 90.3 KB
```

```
In [19]: data.head()
```

```
Out[19]:
```

Date	close	high	low	open	volume
2020-01-01	605.529358	606.192392	601.740655	604.345388	3673698
2020-01-02	609.389099	609.981084	605.718794	605.718794	6137166
2020-01-03	600.698792	608.560338	598.425547	607.234269	10855550
2020-01-06	587.698792	597.573144	585.354555	596.720663	10890186
2020-01-07	597.004761	602.143177	593.050314	596.199685	14724494

```
In [20]: #Slicing By Date Range (from the start until 2020-01-06)  
data.loc[:'2020-01-06', ['close', 'open']]
```

```
Out[20]:
```

Date	close	open
2020-01-01	605.529358	604.345388
2020-01-02	609.389099	605.718794
2020-01-03	600.698792	607.234269
2020-01-06	587.698792	596.720663

```
In [21]: #Slice a single year  
data.loc['2021']
```

```
Out[21]:
```

Date	close	high	low	open	volume
2021-01-01	674.886292	683.387169	672.778791	681.966406	8810938
2021-01-04	670.600342	681.019274	662.549349	681.019274	15740192
2021-01-05	675.667786	677.585840	667.285304	672.115877	14386824
2021-01-06	672.755188	681.966448	669.226924	679.598509	22134050
2021-01-07	670.718689	678.461838	668.966437	678.414490	19894842
...
2021-12-27	690.054932	691.576948	676.427905	679.638458	4705098
2021-12-28	694.811218	697.712595	691.291489	694.763608	5450678
2021-12-29	691.505615	694.906435	688.437776	692.552053	7668702
2021-12-30	695.144226	697.688875	687.296207	693.717314	7215918
2021-12-31	703.658142	706.226596	695.144219	695.144219	6325736

248 rows × 5 columns

```
In [22]: #Slice a specific month  
data.loc['2021-05']
```

```
Out[22]:
```

Date	close	high	low	open	volume
2021-05-03	669.866272	673.394536	652.272531	659.707836	22473700
2021-05-04	657.505615	673.915444	655.114032	667.735100	21486328
2021-05-05	664.254272	667.569387	654.356275	663.496544	14421612
2021-05-06	663.449219	668.137750	660.655039	666.622234	11477044
2021-05-07	670.008301	674.838873	667.877156	669.155820	12048334
2021-05-10	672.423706	677.230634	669.084946	675.809871	11060050
2021-05-11	664.704285	674.483828	660.678788	661.128673	14519034
2021-05-12	662.786255	667.095893	657.742532	662.904652	13774926
2021-05-14	656.795288	662.502045	654.664143	660.347197	10604142
2021-05-17	682.084900	683.197820	654.166919	660.726099	15120692
2021-05-18	699.347107	702.212336	689.070274	690.940923	22165280
2021-05-19	690.585754	700.365355	687.910029	696.268808	10130916
2021-05-20	678.556641	694.232386	676.520190	690.656775	10684184
2021-05-21	709.103027	711.281520	683.387187	683.387187	19341646
2021-05-24	715.093872	720.066544	709.671315	711.920857	22146102
2021-05-25	700.412598	716.893475	696.410804	715.354315	19236974
2021-05-26	699.512878	704.225054	696.174061	700.909939	12283990
2021-05-27	702.164978	705.172249	692.598470	697.642191	20439496
2021-05-28	712.015564	716.538351	700.317968	706.072072	17750020
2021-05-31	717.888062	719.616669	704.461859	710.381706	15661054

```
In [23]: pd.options.display.float_format = '{:.2f}'.format  
#Slice a specific day, output it as DataFrame  
data.loc['2021-05-10']
```

```
Out[23]:
```

close	672.42
high	677.23
low	669.08
open	675.81
volume	11060050.00
Name:	2021-05-10 00:00:00, dtype: float64

```
In [24]: #Slice a date range  
data.loc['2021-01-01' : '2021-03-31']
```

```
Out[24]:
```

	close	high	low	open	volume
Date					
2021-01-01	674.89	683.39	672.78	681.97	8810938
2021-01-04	670.60	681.02	662.55	681.02	15740192
2021-01-05	675.67	677.59	667.29	672.12	14386824
2021-01-06	672.76	681.97	669.23	679.60	22134050
2021-01-07	670.72	678.46	668.97	678.41	19894842
...
2021-03-24	700.34	713.44	696.65	706.07	16368900
2021-03-25	693.02	708.27	686.82	705.74	23964356
2021-03-26	706.26	709.91	698.07	707.54	12021258
2021-03-30	735.81	740.00	711.12	713.53	25607644
2021-03-31	707.37	733.11	704.70	733.11	30774506

61 rows × 5 columns

```
In [25]: #Slice only year and month  
data.loc['2020-07' : '2020-09']
```

```
Out[25]:
```

	close	high	low	open	volume
Date					
2020-07-01	513.65	519.05	502.62	504.77	34846718
2020-07-02	515.93	526.27	513.94	516.35	36954496
2020-07-03	508.61	518.96	506.74	517.63	27597254
2020-07-06	522.37	530.37	520.95	524.71	35558216
2020-07-07	523.39	526.49	517.66	525.40	24349850
...
2020-09-24	487.98	495.85	485.43	491.58	19809510
2020-09-25	494.43	498.19	485.74	497.27	20321544
2020-09-28	499.26	501.53	493.79	496.77	16152422
2020-09-29	503.21	506.69	497.74	501.06	12644446
2020-09-30	510.81	514.74	498.21	502.00	18813842

66 rows × 5 columns

```
In [26]: #Slice by multi-level condition (date + columns)  
data.loc['2021-01-01' : '2021-12-31', ['open', 'close']]
```

```
Out[26]:
```

	open	close
Date		
2021-01-01	681.97	674.89
2021-01-04	681.02	670.60
2021-01-05	672.12	675.67
2021-01-06	679.60	672.76
2021-01-07	678.41	670.72
...
2021-12-27	679.64	690.05
2021-12-28	694.76	694.81
2021-12-29	692.55	691.51
2021-12-30	693.72	695.14
2021-12-31	695.14	703.66

248 rows × 2 columns

```
In [27]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1237 entries, 2020-01-01 to 2024-12-30
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   close    1237 non-null   float64 
 1   high     1237 non-null   float64 
 2   low      1237 non-null   float64 
 3   open     1237 non-null   float64 
 4   volume   1237 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 90.3 KB
```

```
In [28]: #To check the start date
data.index.min()
```

```
Out[28]: Timestamp('2020-01-01 00:00:00')
```

```
In [29]: #To check the end date
data.index.max()
```

```
Out[29]: Timestamp('2024-12-30 00:00:00')
```

```
In [30]: data['sma200'] = data['close'].rolling(window=200).mean()
```

```
In [31]: data.tail()
```

```
Out[31]:
```

	close	high	low	open	volume	sma200
Date						
2024-12-23	888.49	890.96	878.77	879.12	11044592	800.76
2024-12-24	887.06	892.29	882.84	889.01	14485834	801.78
2024-12-26	883.43	893.92	878.50	887.28	10481678	802.72
2024-12-27	887.13	890.93	882.62	885.43	7259330	803.67
2024-12-30	877.09	895.40	873.69	884.15	22222218	804.55

```
In [32]: data.head()
```

```
Out[32]:
```

	close	high	low	open	volume	sma200
Date						
2020-01-01	605.53	606.19	601.74	604.35	3673698	NaN
2020-01-02	609.39	609.98	605.72	605.72	6137166	NaN
2020-01-03	600.70	608.56	598.43	607.23	10855550	NaN
2020-01-06	587.70	597.57	585.35	596.72	10890186	NaN
2020-01-07	597.00	602.14	593.05	596.20	14724494	NaN

Missing values

Missing values are values that are absent from the dataframe. Usually, all the dataframes that you would work on would be large and there will be a case of 'missing values' in most of them.

Hence, it becomes important for you to learn how to handle these missing values.

```
In [33]: ksm = pd.read_csv('stock_nan.csv')
```

In [34]: ksm

Out[34]:

	Symbol	Series	Date	Prev Close	Open Price	High Price	Low Price	Last Price	Close Price	Average Price	Total Traded Quantity	No. of Trades
0	KSM	EQ	02-Apr-18	1131.80	1141.00	1149.55	1121.30	1136.70	1137.15	1135.86	4036351.00	142078.00
1	KSM	EQ	03-Apr-18	1137.15	1134.70	1143.55	1128.10	1139.40	1140.45	1135.21	2038584.00	114034.00
2	KSM	EQ	04-Apr-18	1140.45	1144.00	NaN	NaN	NaN	NaN	NaN	NaN	137029.00
3	KSM	EQ	05-Apr-18	1124.20	NaN	1151.30	1129.10	1146.05	1147.55	1140.71	3881772.00	NaN
4	KSM	EQ	06-Apr-18	1147.55	1143.00	1146.00	1122.10	1127.55	NaN	1128.81	2968871.00	137277.00
5	KSM	EQ	09-Apr-18	NaN	NaN	NaN	1106.55	1111.45	1111.25	NaN	3601441.00	171118.00
6	KSM	EQ	10-Apr-18	1111.25	1112.00	1124.50	NaN	NaN	NaN	1115.77	NaN	NaN
7	KSM	EQ	11-Apr-18	1113.40	1118.00	1131.50	1116.50	1124.50	1124.25	1124.78	4512787.00	102586.00
8	KSM	EQ	12-Apr-18	NaN	1129.45	NaN	1125.00	1164.05	1162.60	1157.71	8522183.00	216130.00
9	KSM	EQ	13-Apr-18	1162.60	1174.00	1185.90	1150.25	1168.00	1171.45	1172.37	10613519.00	NaN

DataFrame.isnull()

This method returns a Boolean result.

It will return 'True' if the data point has a 'NaN' (Not a Number) value. Missing data is represented by a NaN value.

In [35]: # Understanding the 'NaN' values of the 'Close Price' column in the infy dataframe

```
print(ksm['Close Price'].isnull())
```

```
0    False
1    False
2     True
3    False
4    True
5    False
6    True
7    False
8    False
9    False
Name: Close Price, dtype: bool
```

```
In [36]: # Understanding the 'NaN' values of the entire dataframe
```

```
print(ksm.isnull())
```

```
Symbol Series Date Prev Close Open Price High Price Low Price \
0 False False False False False False False False False \
1 False False False False False False False False False \
2 False False False False False False True True \
3 False False False False False True False False \
4 False False False False False False False False \
5 False False False True True True False \
6 False False False False False False False True \
7 False False False False False False False False \
8 False False False True False True False \
9 False False False False False False False False

Last Price Close Price Average Price Total Traded Quantity \
0 False False False False False \
1 False False False False False \
2 True True True True \
3 False False False False False \
4 False True False False False \
5 False False True False False \
6 True False False False True \
7 False False False False False \
8 False False False False False \
9 False False False False False

No. of Trades
0 False
1 False
2 False
3 True
4 False
5 False
6 True
7 False
8 False
9 True
```

DataFrame.notnull()

This method returns a Boolean result.

It will return 'True' if the data point is not a 'NaN' (Not a Number) value. Missing data is represented by a NaN value.

```
In [37]: print(ksm['Close Price'].notnull())
```

```
0 True
1 True
2 False
3 True
4 False
5 True
6 False
7 True
8 True
9 True
Name: Close Price, dtype: bool
```

DataFrame.fillna()

The .fillna() method will fill all the 'NaN' values of the entire dataframe or of the requested columns with a scalar value of your choice. Scaler means a single fixed value, not an array, list, or series.

In [38]: # Replace NaN with a Scalar Value of 1100

```
print(ksm.fillna(1100))
```

	Symbol	Series	Date	Prev Close	Open Price	High Price	Low Price	\
0	KSM	EQ	02-Apr-18	1131.80	1141.00	1149.55	1121.30	
1	KSM	EQ	03-Apr-18	1137.15	1134.70	1143.55	1128.10	
2	KSM	EQ	04-Apr-18	1140.45	1144.00	1160.00	1100.00	
3	KSM	EQ	05-Apr-18	1124.20	1100.00	1151.30	1129.10	
4	KSM	EQ	06-Apr-18	1147.55	1143.00	1146.00	1122.10	
5	KSM	EQ	09-Apr-18	1100.00	1100.00	1100.00	1106.55	
6	KSM	EQ	10-Apr-18	1111.25	1112.00	1124.50	1100.00	
7	KSM	EQ	11-Apr-18	1113.40	1118.00	1131.50	1116.50	
8	KSM	EQ	12-Apr-18	1100.00	1129.45	1100.00	1125.00	
9	KSM	EQ	13-Apr-18	1162.60	1174.00	1185.90	1150.25	

	Last Price	Close Price	Average Price	Total Traded	Quantity \
0	1136.70	1137.15	1135.86	4036351.00	
1	1139.40	1140.45	1135.21	2038584.00	
2	1100.00	1100.00	1100.00	1100.00	
3	1146.05	1147.55	1140.71	3881772.00	
4	1127.55	1100.00	1128.81	2968871.00	
5	1111.45	1111.25	1100.00	3601441.00	
6	1100.00	1100.00	1115.77	1100.00	
7	1124.50	1124.25	1124.78	4512787.00	
8	1164.05	1162.60	1157.71	8522183.00	
9	1168.00	1171.45	1172.37	10613519.00	

	No. of Trades
0	142078.00
1	114034.00
2	137029.00
3	1100.00
4	137277.00
5	171118.00
6	1100.00
7	102586.00
8	216130.00
9	1100.00

In [39]: # This will fill the 'Close Price' column with the scalar value of 1000

#It only fills NaN values temporarily for display - it doesn't modify your actual dataset.

```
print(ksm['Close Price'].fillna(1000))
```

0	1137.15
1	1140.45
2	1000.00
3	1147.55
4	1000.00
5	1111.25
6	1000.00
7	1124.25
8	1162.60
9	1171.45

Name: Close Price, dtype: float64

```
In [40]: import warnings
warnings.filterwarnings("ignore")

# If we want to do 'fillna()' using the 'backfill' method,
# then backfill will take the value from the next row and fill the NaN value with that same value

print(ksm['Close Price'])
print(ksm['Close Price'].fillna(method='backfill'))

0    1137.15
1    1140.45
2      NaN
3    1147.55
4      NaN
5    1111.25
6      NaN
7    1124.25
8    1162.60
9    1171.45
Name: Close Price, dtype: float64
0    1137.15
1    1140.45
2    1147.55
3    1147.55
4    1111.25
5    1111.25
6    1124.25
7    1124.25
8    1162.60
9    1171.45
Name: Close Price, dtype: float64
```

```
In [41]: # It is even possible to do it for the entire dataframe with the 'backfill' values
print(ksm.fillna(method='backfill'))
#print(infy['Close Price'].fillna(method='bfill')) --> backfill or bfill does same
```

	Symbol	Series	Date	Prev Close	Open Price	High Price	Low Price	\
0	KSM	EQ	02-Apr-18	1131.80	1141.00	1149.55	1121.30	
1	KSM	EQ	03-Apr-18	1137.15	1134.70	1143.55	1128.10	
2	KSM	EQ	04-Apr-18	1140.45	1144.00	1151.30	1129.10	
3	KSM	EQ	05-Apr-18	1124.20	1143.00	1151.30	1129.10	
4	KSM	EQ	06-Apr-18	1147.55	1143.00	1146.00	1122.10	
5	KSM	EQ	09-Apr-18	1111.25	1112.00	1124.50	1106.55	
6	KSM	EQ	10-Apr-18	1111.25	1112.00	1124.50	1116.50	
7	KSM	EQ	11-Apr-18	1113.40	1118.00	1131.50	1116.50	
8	KSM	EQ	12-Apr-18	1162.60	1129.45	1185.90	1125.00	
9	KSM	EQ	13-Apr-18	1162.60	1174.00	1185.90	1150.25	

	Last Price	Close Price	Average Price	Total Traded Quantity	\
0	1136.70	1137.15	1135.86	4036351.00	
1	1139.40	1140.45	1135.21	2038584.00	
2	1146.05	1147.55	1140.71	3881772.00	
3	1146.05	1147.55	1140.71	3881772.00	
4	1127.55	1111.25	1128.81	2968871.00	
5	1111.45	1111.25	1115.77	3601441.00	
6	1124.50	1124.25	1115.77	4512787.00	
7	1124.50	1124.25	1124.78	4512787.00	
8	1164.05	1162.60	1157.71	8522183.00	
9	1168.00	1171.45	1172.37	10613519.00	

	No. of Trades
0	142078.00
1	114034.00
2	137029.00
3	137277.00
4	137277.00
5	171118.00
6	102586.00
7	102586.00
8	216130.00
9	NaN

```
In [42]: # If we want to do 'fillna()' using the 'ffill' method, then ffill will take the value from
# the previous row and fill the NaN value with that same value

print(ksm['Close Price'])
print("-----")
print(ksm['Close Price'].fillna(method='ffill'))
# 'pad' does the same thing as 'ffill'
# print(infy['Close Price'].fillna(method='pad'))

0    1137.15
1    1140.45
2      NaN
3    1147.55
4      NaN
5    1111.25
6      NaN
7    1124.25
8    1162.60
9    1171.45
Name: Close Price, dtype: float64
-----
0    1137.15
1    1140.45
2    1140.45
3    1147.55
4    1147.55
5    1111.25
6    1111.25
7    1124.25
8    1162.60
9    1171.45
Name: Close Price, dtype: float64
```

DataFrame.dropna()

This method will drop the entire 'row' which has even a single 'NaN' value present, as per the request.

```
In [43]: # By default, dropna() will exclude or drop all the rows which have even one NaN value in it

print(ksm.dropna())

  Symbol Series      Date  Prev Close  Open Price  High Price  Low Price \
0     KSM     EQ  02-Apr-18     1131.80     1141.00     1149.55    1121.30
1     KSM     EQ  03-Apr-18     1137.15     1134.70     1143.55    1128.10
7     KSM     EQ  11-Apr-18     1113.40     1118.00     1131.50    1116.50

  Last Price  Close Price  Average Price  Total Traded Quantity \
0     1136.70      1137.15        1135.86           4036351.00
1     1139.40      1140.45        1135.21           2038584.00
7     1124.50      1124.25        1124.78           4512787.00

  No. of Trades
0            142078.00
1            114034.00
7            102586.00
```

```
In [44]: # If we specify the axis = 1, it will exclude or drop all the columns which has even one NaN value in it

print(ksm.dropna(axis=1))
```

	Symbol	Series	Date
0	KSM	EQ	02-Apr-18
1	KSM	EQ	03-Apr-18
2	KSM	EQ	04-Apr-18
3	KSM	EQ	05-Apr-18
4	KSM	EQ	06-Apr-18
5	KSM	EQ	09-Apr-18
6	KSM	EQ	10-Apr-18
7	KSM	EQ	11-Apr-18
8	KSM	EQ	12-Apr-18
9	KSM	EQ	13-Apr-18

Replacing values

Replacing helps us to select any data point in the entire dataframe and replace it with the value of our choice.

```
In [45]: import pandas as pd  
import numpy as np
```

```
# We will create a dataframe using the 'pd.DataFrame' constructor
```

```
df = pd.DataFrame({'one': [100, 200, 300, 400, 500, 2000],  
                   'two': [1000, 0, 30, 40, 50, 60]})
```

```
print(df)
```

```
   one  two  
0   100 1000  
1   200    0  
2   300   30  
3   400   40  
4   500   50  
5  2000   60
```

```
In [46]: # .replace() will first find the value which you want to replace and replace it the value you have given.  
# NaN values cannot be replaced as they are not defined  
# Example: In the below '1000' is the value it will find and replace it with '10'
```

```
print(df.replace({1000: 10, 2000: 60}))
```

```
   one  two  
0   100  10  
1   200    0  
2   300   30  
3   400   40  
4   500   50  
5    60   60
```

```
In [47]: print(ksm['Close Price'])
```

```
0    1137.15  
1    1140.45  
2      NaN  
3    1147.55  
4      NaN  
5    1111.25  
6      NaN  
7    1124.25  
8    1162.60  
9    1171.45  
Name: Close Price, dtype: float64
```

```
In [48]: ksm
```

	Symbol	Series	Date	Prev Close	Open Price	High Price	Low Price	Last Price	Close Price	Average Price	Total Traded Quantity	No. of Trades
0	KSM	EQ	02-Apr-18	1131.80	1141.00	1149.55	1121.30	1136.70	1137.15	1135.86	4036351.00	142078.00
1	KSM	EQ	03-Apr-18	1137.15	1134.70	1143.55	1128.10	1139.40	1140.45	1135.21	2038584.00	114034.00
2	KSM	EQ	04-Apr-18	1140.45	1144.00	NaN	NaN	NaN	NaN	NaN	NaN	137029.00
3	KSM	EQ	05-Apr-18	1124.20	NaN	1151.30	1129.10	1146.05	1147.55	1140.71	3881772.00	NaN
4	KSM	EQ	06-Apr-18	1147.55	1143.00	1146.00	1122.10	1127.55	NaN	1128.81	2968871.00	137277.00
5	KSM	EQ	09-Apr-18	NaN	NaN	NaN	1106.55	1111.45	1111.25	NaN	3601441.00	171118.00
6	KSM	EQ	10-Apr-18	1111.25	1112.00	1124.50	NaN	NaN	NaN	1115.77	NaN	NaN
7	KSM	EQ	11-Apr-18	1113.40	1118.00	1131.50	1116.50	1124.50	1124.25	1124.78	4512787.00	102586.00
8	KSM	EQ	12-Apr-18	NaN	1129.45	NaN	1125.00	1164.05	1162.60	1157.71	8522183.00	216130.00
9	KSM	EQ	13-Apr-18	1162.60	1174.00	1185.90	1150.25	1168.00	1171.45	1172.37	10613519.00	NaN

```
In [49]: print(ksm['Close Price'].replace({1147.55: 3000}))
```

```
0    1137.15
1    1140.45
2      NaN
3    3000.00
4      NaN
5    1111.25
6      NaN
7    1124.25
8    1162.60
9    1171.45
Name: Close Price, dtype: float64
```

Reindexing

Reindexing changes the row labels and column labels of a dataframe.

To reindex means to confirm the data to match a given set of labels along a particular axis.

```
In [50]: ksm.head()
```

```
Out[50]:
```

	Symbol	Series	Date	Prev Close	Open Price	High Price	Low Price	Last Price	Close Price	Average Price	Total Traded Quantity	No. of Trades
0	KSM	EQ	02-Apr-18	1131.80	1141.00	1149.55	1121.30	1136.70	1137.15	1135.86	4036351.00	142078.00
1	KSM	EQ	03-Apr-18	1137.15	1134.70	1143.55	1128.10	1139.40	1140.45	1135.21	2038584.00	114034.00
2	KSM	EQ	04-Apr-18	1140.45	1144.00	NaN	NaN	NaN	NaN	NaN	NaN	137029.00
3	KSM	EQ	05-Apr-18	1124.20	NaN	1151.30	1129.10	1146.05	1147.55	1140.71	3881772.00	NaN
4	KSM	EQ	06-Apr-18	1147.55	1143.00	1146.00	1122.10	1127.55	NaN	1128.81	2968871.00	137277.00

```
In [51]: #Reindexing adjusts the row and column layout as specified, producing a reshaped DataFrame
```

```
ksm_reindexed = ksm.reindex(index=[0, 2, 4, 6, 8], columns=[
```

```
    'Open Price', 'High Price', 'Low Price', 'Close Price'])
```

```
print(ksm_reindexed)
```

	Open Price	High Price	Low Price	Close Price
0	1141.00	1149.55	1121.30	1137.15
2	1144.00	NaN	NaN	NaN
4	1143.00	1146.00	1122.10	NaN
6	1112.00	1124.50	NaN	NaN
8	1129.45	NaN	1125.00	1162.60

```
In [52]: data.head()
```

```
Out[52]:
```

	close	high	low	open	volume	sma200
--	-------	------	-----	------	--------	--------

Date	close	high	low	open	volume	sma200
2020-01-01	605.53	606.19	601.74	604.35	3673698	NaN
2020-01-02	609.39	609.98	605.72	605.72	6137166	NaN
2020-01-03	600.70	608.56	598.43	607.23	10855550	NaN
2020-01-06	587.70	597.57	585.35	596.72	10890186	NaN
2020-01-07	597.00	602.14	593.05	596.20	14724494	NaN

```
In [53]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1237 entries, 2020-01-01 to 2024-12-30
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   close    1237 non-null   float64
 1   high     1237 non-null   float64
 2   low      1237 non-null   float64
 3   open     1237 non-null   float64
 4   volume   1237 non-null   int64  
 5   sma200  1038 non-null   float64
dtypes: float64(5), int64(1)
memory usage: 99.9 KB
```

Class Exercises: Using .loc on a Price DataFrame

1. Select all rows for the year 2021 using .loc .
2. Select close prices between 2022-01-01 and 2022-03-31 .
3. Select the rows where high > 500 using .loc with a condition.
4. Select both close and volume columns for the date range 2023-05-01 to 2023-05-31 .
5. Select the row for a single date (e.g., 2020-06-15) and print all columns.
6. Select rows where volume is above its 95th percentile using .loc .
7. Select rows where close > open for the entire year 2024.
8. Create a new df using .loc that contains only rows from 2021 and columns ['close', 'open'] .
9. Select the last 10 dates using .loc with index slicing.
10. Select rows where (high – low) > 10 and display close, high, low columns only.
11. Extract all rows where close crossed above 200-day MA

```
In [54]: df2 = data.copy()
```

```
In [55]: #Select all rows for the year 2021 using .Loc.  
df2.loc['2021'][:]
```

```
Out[55]:
```

	close	high	low	open	volume	sma200
Date						
2021-01-01	674.89	683.39	672.78	681.97	8810938	523.63
2021-01-04	670.60	681.02	662.55	681.02	15740192	524.68
2021-01-05	675.67	677.59	667.29	672.12	14386824	525.98
2021-01-06	672.76	681.97	669.23	679.60	22134050	527.22
2021-01-07	670.72	678.46	668.97	678.41	19894842	528.48
...
2021-12-27	690.05	691.58	676.43	679.64	4705098	718.17
2021-12-28	694.81	697.71	691.29	694.76	5450678	718.02
2021-12-29	691.51	694.91	688.44	692.55	7668702	717.88
2021-12-30	695.14	697.69	687.30	693.72	7215918	717.66
2021-12-31	703.66	706.23	695.14	695.14	6325736	717.49

248 rows × 6 columns

```
In [56]: #Select close prices between 2022-01-01 and 2022-03-31.  
df2.loc['2022-01-01' : '2022-03-31', 'close']
```

```
Out[56]:
```

Date	close
2022-01-03	722.80
2022-01-04	727.04
2022-01-05	744.30
2022-01-06	732.36
2022-01-07	737.50
...	
2022-03-25	680.59
2022-03-28	681.49
2022-03-29	690.53
2022-03-30	702.49
2022-03-31	699.35

Name: close, Length: 61, dtype: float64

```
In [57]: #Now the output is a DataFrame (because Lists preserve 2D structure).
df2.loc['2022-01-01' : '2022-03-31', ['close']]
```

Out[57]: close

Date	close
2022-01-03	722.80
2022-01-04	727.04
2022-01-05	744.30
2022-01-06	732.36
2022-01-07	737.50
...	...
2022-03-25	680.59
2022-03-28	681.49
2022-03-29	690.53
2022-03-30	702.49
2022-03-31	699.35

61 rows × 1 columns

```
In [58]: #Select the rows where high > 500 using .loc with a condition.
df2.loc[df2['high'] > 500]
```

Out[58]: close high low open volume sma200

Date	close	high	low	open	volume	sma200
2020-01-01	605.53	606.19	601.74	604.35	3673698	NaN
2020-01-02	609.39	609.98	605.72	605.72	6137166	NaN
2020-01-03	600.70	608.56	598.43	607.23	10855550	NaN
2020-01-06	587.70	597.57	585.35	596.72	10890186	NaN
2020-01-07	597.00	602.14	593.05	596.20	14724494	NaN
...
2024-12-23	888.49	890.96	878.77	879.12	11044592	800.76
2024-12-24	887.06	892.29	882.84	889.01	14485834	801.78
2024-12-26	883.43	893.92	878.50	887.28	10481678	802.72
2024-12-27	887.13	890.93	882.62	885.43	7259330	803.67
2024-12-30	877.09	895.40	873.69	884.15	22222218	804.55

1164 rows × 6 columns

```
In [59]: #Select both close and volume columns for the date range 2023-05-01 to 2023-05-31.  
df2.loc['2023-05-01' : '2023-05-31', ['close', 'volume']]
```

```
Out[59]:
```

Date	close	volume
2023-05-02	811.85	32221184
2023-05-03	814.69	29407270
2023-05-04	831.36	56854638
2023-05-05	782.21	62770416
2023-05-08	791.26	37244524
2023-05-09	791.28	43688938
2023-05-10	794.94	46340550
2023-05-11	795.47	35734618
2023-05-12	802.49	22478760
2023-05-15	806.34	18512260
2023-05-16	801.72	32722086
2023-05-17	797.41	30466868
2023-05-18	800.60	49646942
2023-05-19	801.52	30460332
2023-05-22	798.68	28334032
2023-05-23	796.80	22470074
2023-05-24	786.39	33588112
2023-05-25	783.37	29776930
2023-05-26	786.39	26635066
2023-05-29	795.97	21816292
2023-05-30	796.58	37125928
2023-05-31	783.98	43334270

```
In [60]: #Select the row for a single date (e.g., 2020-06-15) and print all columns.
```

```
df2.loc['2020-06-15']
```

```
Out[60]:
```

close	449.84
high	461.27
low	446.59
open	458.43
volume	32009936.00
sma200	NaN
Name:	2020-06-15 00:00:00, dtype: float64

```
In [61]: #Select rows where volume is above its 95th percentile using .loc  
df2.loc[df2['volume'] > df2['volume'].quantile(0.95)]
```

```
Out[61]:
```

Date	close	high	low	open	volume	sma200
2020-03-12	483.68	511.47	475.22	509.11	59000770	NaN
2020-03-13	506.64	512.16	435.32	464.14	66342364	NaN
2020-03-18	415.29	470.27	409.65	466.48	61181406	NaN
2020-03-19	424.12	435.68	376.50	401.13	67220048	NaN
2020-03-20	418.11	433.14	390.50	414.39	88637464	NaN
...
2024-07-24	791.33	799.69	783.44	793.35	61728206	750.73
2024-08-30	807.53	819.92	799.76	816.96	445342100	759.52
2024-09-20	858.99	860.84	842.39	846.80	60623386	766.12
2024-10-07	798.11	818.44	795.74	815.08	94458296	769.51
2024-11-25	880.89	889.75	870.68	880.62	427737274	779.68

62 rows × 6 columns

```
In [62]: #Select rows where close > open for the entire year 2024.  
df2.loc['2024'][df2['close'] > df2['open']]
```

```
Out[62]:
```

Date	close	high	low	open	volume	sma200
2024-01-02	826.93	828.73	821.87	826.78	29242092	780.04
2024-01-04	822.91	824.91	813.11	816.66	26734056	780.63
2024-01-10	805.93	807.87	798.85	799.62	16115824	781.83
2024-01-15	814.13	818.07	799.92	801.91	28320356	782.56
2024-01-16	817.22	819.41	806.97	814.23	25322500	782.88
...
2024-12-12	917.23	921.86	912.02	913.16	17293790	793.62
2024-12-13	923.39	925.00	902.43	918.83	19009002	794.70
2024-12-16	920.16	922.51	915.25	920.11	13107082	795.80
2024-12-23	888.49	890.96	878.77	879.12	11044592	800.76
2024-12-27	887.13	890.93	882.62	885.43	7259330	803.67

126 rows × 6 columns

```
In [63]: #Create a new df using .loc that contains only rows from 2021 and columns ['close', 'open'].
```

```
df3 = df2.loc['2021', ['close', 'open']]
```

```
In [64]: df3.tail()
```

```
Out[64]:
```

Date	close	open
2021-12-27	690.05	679.64
2021-12-28	694.81	694.76
2021-12-29	691.51	692.55
2021-12-30	695.14	693.72
2021-12-31	703.66	695.14

```
In [65]: #Select the last 10 dates using .loc with index slicing.  
df2.loc[df2.index[-10:]]
```

```
Out[65]:
```

Date	close	high	low	open	volume	sma200
2024-12-16	920.16	922.51	915.25	920.11	13107082	795.80
2024-12-17	904.40	918.76	901.07	915.80	21079532	796.86
2024-12-18	893.28	905.76	891.52	902.72	23422312	797.87
2024-12-19	884.79	888.54	877.64	887.80	25380770	798.84
2024-12-20	873.94	886.79	871.74	879.22	25692348	799.75
2024-12-23	888.49	890.96	878.77	879.12	11044592	800.76
2024-12-24	887.06	892.29	882.84	889.01	14485834	801.78
2024-12-26	883.43	893.92	878.50	887.28	10481678	802.72
2024-12-27	887.13	890.93	882.62	885.43	7259330	803.67
2024-12-30	877.09	895.40	873.69	884.15	22222218	804.55

```
In [66]: #Select rows where (high - low) > 10 and display close, high, low columns only.  
sel = df2['high'] - df2['low'] > 10  
df2.loc[sel, ['close', 'high', 'low']]
```

Out[66]:

	close	high	low
Date			
2020-01-03	600.70	608.56	598.43
2020-01-06	587.70	597.57	585.35
2020-01-08	595.44	597.74	587.27
2020-01-20	594.31	617.96	593.17
2020-01-27	574.56	584.88	573.87
...
2024-12-19	884.79	888.54	877.64
2024-12-20	873.94	886.79	871.74
2024-12-23	888.49	890.96	878.77
2024-12-26	883.43	893.92	878.50
2024-12-30	877.09	895.40	873.69

810 rows × 3 columns

```
In [67]: #Extract all rows where close price crossed above 200-MA  
df2.head()
```

Out[67]:

	close	high	low	open	volume	sma200
Date						
2020-01-01	605.53	606.19	601.74	604.35	3673698	NaN
2020-01-02	609.39	609.98	605.72	605.72	6137166	NaN
2020-01-03	600.70	608.56	598.43	607.23	10855550	NaN
2020-01-06	587.70	597.57	585.35	596.72	10890186	NaN
2020-01-07	597.00	602.14	593.05	596.20	14724494	NaN

```
In [68]: condition1 = df2['close'] > df2['sma200']  
condition2 = df2['close'].shift(1) <= df2['sma200'].shift(1)  
df2.loc[condition1 & condition2]
```

Out[68]:

	close	high	low	open	volume	sma200
Date						
2021-08-04	696.95	701.33	684.92	685.39	22053948	685.10
2021-11-25	725.80	729.29	716.79	720.50	10251652	722.39
2021-12-02	725.70	727.15	713.46	715.60	11203110	721.39
2021-12-07	725.68	728.68	718.17	720.09	12427534	720.98
2022-01-03	722.80	724.40	704.18	706.32	9069184	717.43
2022-01-21	723.73	727.63	706.61	713.46	11537694	719.14
2022-02-02	728.30	730.10	716.12	719.14	13969290	720.40
2022-02-10	725.39	730.34	714.24	720.12	14314530	721.72
2022-02-21	723.97	728.06	711.34	715.36	7468066	723.15
2022-04-04	788.04	819.10	743.21	751.51	97450970	721.73
2022-08-01	695.84	696.95	688.09	692.40	13193036	693.67
2022-08-08	703.49	704.67	686.67	687.11	15357736	691.07
2022-10-04	699.14	701.54	686.22	687.83	11538526	685.33
2022-10-14	692.40	696.25	680.97	681.81	12849920	685.03
2023-09-08	790.09	794.71	782.40	782.74	34502210	787.60
2023-12-04	783.27	784.54	765.87	772.86	50662456	777.50
2024-05-24	748.48	749.81	733.09	733.83	31060206	747.27
2024-05-30	747.32	751.64	738.84	739.60	33755350	746.21
2024-06-05	765.55	769.40	731.29	737.06	81631650	745.34

TTP Assignment 14

Pandas Indexing & Generating Trading Signals

Question #1 – Download Market Data Using yfinance

Download daily price data for the ticker LT.NS using yfinance for the period **2020-01-01 to 2024-12-31**.

- Store the data in a DataFrame named df
- Ensure the index is a **DateTimeIndex**

```
In [1]: import numpy as np
import pandas as pd
import yfinance as yf
import warnings
warnings.filterwarnings("ignore")
```

```
In [2]: start_date = "2020-01-01"
end_date = "2024-12-31"
df = yf.download("LT.NS", start_date, end_date)
```

[*****100%*****] 1 of 1 completed

```
In [3]: df.head(2)
```

```
Out[3]:      Price      Close      High      Low      Open    Volume
           Ticker       LT.NS       LT.NS       LT.NS       LT.NS       LT.NS
          Date
2020-01-01  1192.206543  1200.352149  1185.881281  1190.79593  3123998
2020-01-02  1224.379395  1226.836664  1193.162364  1194.07248  4335359
```

```
In [4]: df.columns = ['close', 'high', 'low', 'open', 'volume']
```

```
In [5]: df.head(2)
```

```
Out[5]:      close      high      low      open    volume
           Date
2020-01-01  1192.206543  1200.352149  1185.881281  1190.79593  3123998
2020-01-02  1224.379395  1226.836664  1193.162364  1194.07248  4335359
```

```
In [6]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1237 entries, 2020-01-01 to 2024-12-30
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   close    1237 non-null   float64
 1   high     1237 non-null   float64
 2   low      1237 non-null   float64
 3   open     1237 non-null   float64
 4   volume   1237 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 58.0 KB
```

Question #2 – Date-Based Row Selection with .loc

Using .loc , extract all rows belonging to the calendar year 2021.

- Print the first and last 5 rows of the result

```
In [7]: df_year = df.loc['2021']
df_year.head(5)
```

Out[7]:

Date	close	high	low	open	volume
2021-01-01	1230.778198	1233.577532	1217.493006	1217.493006	2058419
2021-01-04	1247.479736	1255.023898	1235.523084	1238.369912	4014636
2021-01-05	1239.603516	1245.961383	1228.927914	1240.267729	2833637
2021-01-06	1246.910400	1262.093480	1233.625206	1241.216746	4615963
2021-01-07	1270.586548	1278.035839	1257.348846	1262.093559	4270598

```
In [8]: df_year.tail(5)
```

Out[8]:

Date	close	high	low	open	volume
2021-12-27	1790.961304	1800.030383	1771.383854	1784.435526	1139744
2021-12-28	1822.918823	1832.035817	1796.623472	1796.623472	1916098
2021-12-29	1818.456177	1831.987690	1813.849766	1818.600153	1287946
2021-12-30	1809.675171	1825.030107	1805.068760	1816.009176	1805349
2021-12-31	1819.463867	1830.212275	1811.210613	1811.258527	1145546

Question #3 – Range Selection on Dates

Use .loc to select only the Close price between 2022-01-01 and 2022-03-31 .

- Verify the number of rows selected

```
In [12]: close_2022 = df.loc["2022-01-01":"2022-03-31", "close"]
```

```
print(close_2022.head(5))
```

```
Date  
2022-01-03    1845.327271  
2022-01-04    1859.434814  
2022-01-05    1870.039185  
2022-01-06    1846.910889  
2022-01-07    1828.101074  
Name: close, dtype: float64
```

```
In [11]: print("Number of rows:", close_2022.shape[0])
```

```
Number of rows: 61
```

Question #4 – Conditional Row Filtering with .loc

Using .loc , filter and display all rows where the **High** price is greater than **3800**.

- Display only Open, High, Low, Close columns

```
In [14]: high_gt_3800 = df.loc[df["high"] > 3800, ["open", "high", "low", "close"]]  
print(high_gt_3800.head(5))
```

	open	high	low	close
Date				
2024-06-03	3789.982814	3853.388400	3736.604349	3831.024414
2024-06-04	3786.640799	3831.860289	3263.667563	3345.455811
2024-08-01	3774.794940	3803.477389	3731.052991	3744.378662
2024-09-23	3764.887322	3803.328851	3714.358571	3752.700928
2024-12-05	3774.795073	3848.111302	3725.257080	3796.145996

Question #5 – Date Range + Column Subset

Select data using .loc for the date range 2023-05-01 to 2023-05-31 , but display only:

- **Close**
- **Volume**

```
In [18]: range_2023 = df.loc["2023-05-01":"2023-05-31", ["close", "volume"]]
```

```
print(range_2023.head(5))
```

	close	volume
Date		
2023-05-02	2317.621338	1840153
2023-05-03	2290.260498	998235
2023-05-04	2290.843506	1230912
2023-05-05	2310.865967	2425866
2023-05-08	2299.105225	1583576

Question #6 – Single Date Extraction

Using .loc , extract the row corresponding to the date 2020-06-15 .

- Print **all available columns** for that date

```
In [19]: date_row = df.loc["2020-06-15"]
```

```
print(date_row)
```

```
close      8.304836e+02
high       8.577541e+02
low        8.251309e+02
open       8.506942e+02
volume     4.982324e+06
Name: 2020-06-15 00:00:00, dtype: float64
```

Question #7 – Quantile-Based Filtering

Calculate the **95th percentile of Volume**.

Using `.loc`, select all rows where volume exceeds this threshold.

- Print the total number of such days

```
In [20]: volume_95 = df["volume"].quantile(0.95)
```

```
high_volume_days = df.loc[df["volume"] > volume_95]
```

```
print("Total days:", high_volume_days.shape[0])
```

Total days: 62

Question #8 – Combined Date + Condition Filter

Using `.loc`, select all rows from the year **2024** where:

- **Close > Open**
- Display only Open and Close columns

```
In [22]: year_2024 = df.loc[
    (df.index.year == 2024) & (df["close"] > df["open"]),
    ["open", "close"]
]

print(year_2024.head(5))
```

Date	open	close
2024-01-03	3374.750202	3381.975586
2024-01-04	3382.663491	3400.013916
2024-01-05	3406.157716	3462.141357
2024-01-09	3467.597590	3495.614014
2024-01-12	3445.577538	3508.245850

Question #9 – Creating a New DataFrame with `.loc`

Create a new DataFrame using `.loc` that contains:

- Only rows from the year **2021**
- Only the columns `['Open', 'Close']`

```
In [24]: df_2021 = df.loc["2021", ["open", "close"]]
```

```
print(df_2021.head(5))
```

	open	close
Date		
2021-01-01	1217.493006	1230.778198
2021-01-04	1238.369912	1247.479736
2021-01-05	1240.267729	1239.603516
2021-01-06	1241.216746	1246.910400
2021-01-07	1262.093559	1270.586548

Question #10 – Advanced Condition: Range & Candlestick Logic

Using .loc :

- Select the **last 10 trading dates** from the DataFrame
- From the full dataset, identify rows where $(\text{High} - \text{Low}) > 10$
- Display only Close, High, and Low columns

Identify **Bullish Engulfing** candles using .loc , where:

- Current candle is bullish
- Current candle's body completely engulfs the previous candle's body

```
In [25]: last_10_days = df.loc[df.index[-10:]]
```

```
print(last_10_days)
```

	close	high	low	open	volume
Date					
2024-12-16	3842.018066	3878.230144	3821.608317	3846.129623	918660
2024-12-17	3772.020752	3839.590524	3761.915050	3827.354785	1690180
2024-12-18	3723.424072	3783.711907	3710.395677	3772.020892	1318341
2024-12-19	3682.010498	3727.436692	3665.811478	3667.941515	1856838
2024-12-20	3596.309814	3690.580519	3573.770027	3682.010543	2796593
2024-12-23	3606.861328	3665.811541	3583.628057	3636.088744	1217612
2024-12-24	3606.118164	3645.005488	3598.835982	3614.242346	627922
2024-12-26	3595.962891	3631.283383	3576.989936	3624.199546	1124705
2024-12-27	3574.760742	3609.982062	3570.203150	3598.836013	1309122
2024-12-30	3545.879883	3589.522902	3533.049591	3574.562332	1591690

```
In [26]: range_gt_10 = df.loc[(df["high"] - df["low"]) > 10, ["close", "high", "low"]]
```

```
print(range_gt_10.head())
```

	close	high	low
Date			
2020-01-01	1192.206543	1200.352149	1185.881281
2020-01-02	1224.379395	1226.836664	1193.162364
2020-01-03	1215.050781	1224.060844	1210.591189
2020-01-06	1198.395508	1212.274780	1195.983678
2020-01-07	1201.808472	1219.100680	1195.665187

```
In [ ]: bullish_engulfing = df.loc[
    (df["Close"] > df["Open"]) &
    (df["Close"].shift(1) < df["Open"].shift(1)) &
    (df["Open"] < df["Close"].shift(1)) &
    (df["Close"] > df["Open"].shift(1))
]

print(bullish_engulfing[["Open", "High", "Low", "Close"]].head())
```

Question #11 – Long-Term Trend Filter Using MA200

Calculate the **200-day Simple Moving Average (MA200)** of the Close price.

Using `.loc` :

- Identify all dates where **Close < MA200**
- Count how many such days occurred
- Display Close and MA200 columns only

```
In [27]: df["MA200"] = df["close"].rolling(200).mean()

below_ma200 = df.loc[df["close"] < df["MA200"], ["close", "MA200"]]

print("Total days below MA200:", below_ma200.shape[0])
print(below_ma200.head())
```

```
Total days below MA200: 112
      close      MA200
Date
2020-10-19  840.527466  916.568068
2020-10-20  853.932678  914.876698
2020-10-21  860.588867  913.057746
2020-10-22  871.620239  911.340593
2020-10-23  876.228210  909.729756
```

Question #12 – Moving Average Relationship (MA50 vs MA200)

Calculate both **MA50** and **MA200**.

Using `.loc` :

- Select all rows where **MA50 > MA200**
- Interpret this condition as a market regime filter
- Display Close, MA50, and MA200 columns

```
In [30]: df["MA50"] = df["close"].rolling(50).mean()

bull_market = df.loc[df["MA50"] > df["MA200"], ["close", "MA50", "MA200"]]

print(bull_market.head())
```

```
      close      MA50      MA200
Date
2020-11-20  1074.487549  884.510509  884.435793
2020-11-23  1068.366821  888.797328  883.899414
2020-11-24  1077.524170  893.552156  883.325546
2020-11-25  1059.019775  897.790741  882.624799
2020-11-26  1061.724243  901.862788  882.022215
```

Question #13 – Inside Bar Detection

An **Inside Bar** is defined as a candle where:

- Current High < Previous High
- Current Low > Previous Low

Using `.loc`:

- Detect all Inside Bar candles
- Display Date, Open, High, Low, Close for those rows

```
In [31]: inside_bar = df.loc[
    (df["high"] < df["high"].shift(1)) &
    (df["low"] > df["low"].shift(1))
]

inside_bar_result = inside_bar[["open", "high", "low", "close"]]
print(inside_bar_result.head())
```

Date	open	high	low	close
2020-01-03	1224.060844	1224.060844	1210.591189	1215.050781
2020-01-20	1195.847167	1201.262425	1189.066846	1191.888184
2020-01-27	1219.555713	1242.217629	1219.555713	1227.382690
2020-02-11	1175.870086	1182.195349	1168.589157	1171.228516
2020-03-03	1061.195538	1080.126001	1057.600624	1075.120361

Question #14 – Outside Bar Detection

An **Outside Bar** is defined as a candle where:

- Current High > Previous High
- Current Low < Previous Low

Using `.loc`:

- Identify all Outside Bar candles
- Display only High, Low, and Close columns

```
In [32]: outside_bar = df.loc[
    (df["high"] > df["high"].shift(1)) &
    (df["low"] < df["low"].shift(1)),
    ["high", "low", "close"]
]

print(outside_bar.head())
```

Date	high	low	close
2020-01-07	1219.100680	1195.665187	1201.808472
2020-02-12	1183.014455	1164.948672	1180.693726
2020-03-02	1103.379391	1047.134274	1056.553955
2020-03-04	1081.081440	1053.914495	1070.888184
2020-03-25	715.218177	610.011890	706.681702

Question #15 – Bearish Engulfing Pattern

A **Bearish Engulfing** pattern is defined as:

- Previous candle is bullish (Close > Open)
- Current candle is bearish (Close < Open)
- Current candle's body completely engulfs the previous candle's body

Using .loc :

- Detect all Bearish Engulfing occurrences

```
In [33]: bearish_engulfing = df.loc[
    (df["close"].shift(1) > df["open"].shift(1)) &
    (df["close"] < df["open"]) &
    (df["open"] > df["close"].shift(1)) &
    (df["close"] < df["open"].shift(1))
]

result = bearish_engulfing[["open", "high", "low", "close"]]
print(result.head())
```

Date	open	high	low	close
2020-02-17	1180.420611	1181.194187	1162.218288	1165.130615
2020-02-19	1181.330930	1182.241047	1161.854420	1166.223022
2020-04-20	868.228608	870.258916	840.727330	844.972473
2020-07-20	857.338805	860.015127	844.418737	848.663879
2020-07-22	866.752073	868.413214	839.804489	849.863708

```
In [ ]:
```

Pandas groupby()

groupby() works through three steps:

Split the data into groups → **Apply** an operation → **Combine** the results.

Operations include:

- **Aggregation** (mean, sum, count)
- **Transformation** (group-wise z-score, scaling)
- **Filtration** (remove groups based on a rule)

This pattern makes groupby() essential for multi-stock analysis, sector studies, time-based analysis and portfolio grouping.

In [1]: # Creating a DataFrame

```
import pandas as pd

my_portfolio = {
    'Sector': [
        'IT', 'FMCG', 'Banking', 'Pharma', 'Auto',
        'FMCG', 'Oil & Gas', 'IT', 'Banking', 'Real Estate'],
    'Company': [
        'Infosys', 'Hindustan Unilever', 'HDFC Bank', 'Sun Pharma', 'Tata Motors',
        'Britannia', 'Oil India', 'Tata Elxsi', 'ICICI Bank', 'Godrej Properties'],
    'MarketCap': [
        'Large Cap', 'Large Cap', 'Large Cap', 'Large Cap', 'Mid Cap',
        'Large Cap', 'Mid Cap', 'Small Cap', 'Large Cap', 'Small Cap'],
    'Share Price': [
        1600, 2450, 1650, 1450, 980,
        5250, 510, 8800, 1120, 1900 ],
    'Amount Invested': [
        40000, 25000, 30000, 20000, 15000,
        35000, 18000, 25000, 20000, 12000] }

tpp = pd.DataFrame(my_portfolio)
tpp
```

Out[1]:

	Sector	Company	MarketCap	Share Price	Amount Invested
0	IT	Infosys	Large Cap	1600	40000
1	FMCG	Hindustan Unilever	Large Cap	2450	25000
2	Banking	HDFC Bank	Large Cap	1650	30000
3	Pharma	Sun Pharma	Large Cap	1450	20000
4	Auto	Tata Motors	Mid Cap	980	15000
5	FMCG	Britannia	Large Cap	5250	35000
6	Oil & Gas	Oil India	Mid Cap	510	18000
7	IT	Tata Elxsi	Small Cap	8800	25000
8	Banking	ICICI Bank	Large Cap	1120	20000
9	Real Estate	Godrej Properties	Small Cap	1900	12000

View groups

In [2]: #Output is keys='MarketCap' and values= the index labels/row numbers
tpp.groupby('MarketCap').groups

Out[2]: {'Large Cap': [0, 1, 2, 3, 5, 8], 'Mid Cap': [4, 6], 'Small Cap': [7, 9]}

In [3]: tpp.groupby('Sector').groups

Out[3]: {'Auto': [4], 'Banking': [2, 8], 'FMCG': [1, 5], 'IT': [0, 7], 'Oil & Gas': [6], 'Pharma': [3], 'Real Estate': [9]}

```
In [4]: # Groupby with multiple columns
```

```
ttt.groupby(['MarketCap', 'Sector']).groups
```

```
Out[4]: {('Large Cap', 'Banking'): [2, 8], ('Large Cap', 'FMCG'): [1, 5], ('Large Cap', 'IT'): [0], ('Large Cap', 'Pharma'): [3], ('Mid Cap', 'Auto'): [4], ('Mid Cap', 'Oil & Gas'): [6], ('Small Cap', 'IT'): [7], ('Small Cap', 'Real Estate'): [9]}
```

Iterating through groups (a better way to see)

```
In [5]:
```

```
# A better way to visualise  
# Create separate object(mini-dataframes container) for grouping each sector  
grouped = ttt.groupby('Sector')  
  
# The Loop goes through each (key, subset) pair in the grouped object.  
# name → the group name (e.g., 'IT', 'Banking', 'FMCG')  
# group → the mini-DataFrame containing only the rows belonging to that sector.  
# This prints the sector name (the group key).  
  
for name, group in grouped:  
    print(name)  
    print(group)
```

```
Auto
```

```
    Sector      Company MarketCap Share Price Amount Invested  
4   Auto  Tata Motors   Mid Cap        980          15000  
Banking  
    Sector      Company MarketCap Share Price Amount Invested  
2  Banking  HDFC Bank  Large Cap       1650          30000  
8  Banking  ICICI Bank  Large Cap       1120          20000  
FMCG  
    Sector      Company MarketCap Share Price Amount Invested  
1   FMCG  Hindustan Unilever  Large Cap       2450          25000  
5   FMCG        Britannia  Large Cap       5250          35000  
IT  
    Sector      Company MarketCap Share Price Amount Invested  
0     IT      Infosys  Large Cap       1600          40000  
7     IT  Tata Elxsi  Small Cap       8800          25000  
Oil & Gas  
    Sector      Company MarketCap Share Price Amount Invested  
6  Oil & Gas  Oil India   Mid Cap        510          18000  
Pharma  
    Sector      Company MarketCap Share Price Amount Invested  
3  Pharma  Sun Pharma  Large Cap       1450          20000  
Real Estate  
    Sector      Company MarketCap Share Price Amount Invested  
9 Real Estate  Godrej Properties  Small Cap       1900          12000
```

```
In [6]: # Lets try with MarketCap now
```

```
grouped = ttt.groupby('MarketCap')  
  
for name, group in grouped: # We will Learn 'for' Loop in further sections. It is usually used for iterations  
    print(name)  
    print(group)
```

```
Large Cap
```

```
    Sector      Company MarketCap Share Price Amount Invested  
0     IT      Infosys  Large Cap       1600          40000  
1   FMCG  Hindustan Unilever  Large Cap       2450          25000  
2  Banking  HDFC Bank  Large Cap       1650          30000  
3  Pharma        Sun Pharma  Large Cap       1450          20000  
5   FMCG        Britannia  Large Cap       5250          35000  
8  Banking  ICICI Bank  Large Cap       1120          20000  
Mid Cap  
    Sector      Company MarketCap Share Price Amount Invested  
4     Auto  Tata Motors   Mid Cap        980          15000  
6  Oil & Gas  Oil India   Mid Cap        510          18000  
Small Cap  
    Sector      Company MarketCap Share Price Amount Invested
```

```
7     IT        Tata Elxsi  Small Cap       8800          25000  
9 Real Estate  Godrej Properties  Small Cap       1900          12000
```

Select a group or to get one group

```
In [7]: grouped = ttp.groupby('MarketCap')

print(grouped.get_group('Mid Cap'))
print("-----")
print(grouped.get_group('Large Cap'))

      Sector      Company MarketCap Share Price  Amount Invested
4       Auto   Tata Motors   Mid Cap        980     15000
6    Oil & Gas      Oil India   Mid Cap        510     18000
-----
      Sector      Company MarketCap Share Price  Amount Invested
0        IT      Infosys  Large Cap       1600     40000
1    FMCG  Hindustan Unilever  Large Cap       2450     25000
2  Banking      HDFC Bank  Large Cap       1650     30000
3  Pharma      Sun Pharma  Large Cap       1450     20000
5    FMCG      Britannia  Large Cap       5250     35000
8  Banking     ICICI Bank  Large Cap       1120     20000
```

Aggregations

```
In [8]: import numpy as np
import warnings
warnings.filterwarnings("ignore")

grouped = ttp.groupby('MarketCap')

print(grouped['Amount Invested'].agg(np.mean))

MarketCap
Large Cap    28333.333333
Mid Cap     16500.000000
Small Cap    18500.000000
Name: Amount Invested, dtype: float64
```

This means that on an average, we have invested Rs. 28333 per script in Large Cap, Rs. 16500 per script in Mid Cap and Rs. 18500 per script in Small Cap

```
In [10]: # Applying multiple aggregation functions at once

grouped = ttp.groupby('MarketCap')

print(grouped['Amount Invested'].agg([np.sum, np.mean]))
```

	sum	mean
MarketCap		
Large Cap	170000	28333.333333
Mid Cap	33000	16500.000000
Small Cap	37000	18500.000000

```
In [11]: print(np.round(grouped['Amount Invested'].agg([np.sum, np.mean]),2))

      sum      mean
MarketCap
Large Cap  170000  28333.33
Mid Cap    33000  16500.00
Small Cap  37000  18500.00
```

```
In [12]: grouped = ttp.groupby('MarketCap')
```

Transformations

In [14]: #This function takes a Series x and standardizes it

```
def z_score(x): return (x - x.mean()) / x.std()

print(grouped[['Share Price', 'Amount Invested']].transform(z_score))
```

	Share Price	Amount Invested
0	-0.426381	1.428869
1	0.128349	-0.408248
2	-0.393750	0.204124
3	-0.524275	-1.020621
4	0.707107	-0.707107
5	1.955698	0.816497
6	-0.707107	0.707107
7	0.707107	0.707107
8	-0.739641	-1.020621
9	-0.707107	-0.707107

Applies the z_score function to each numeric column

Filteration

In [16]: print(ttp.groupby('MarketCap').filter(lambda x: len(x) <= 3))

	Sector	Company	MarketCap	Share Price	Amount Invested
4	Auto	Tata Motors	Mid Cap	980	15000
6	Oil & Gas	Oil India	Mid Cap	510	18000
7	IT	Tata Elxsi	Small Cap	8800	25000
9	Real Estate	Godrej Properties	Small Cap	1900	12000

It will filter out the Groups that have less than 3 companies in that particular group.

Merging/Joining

In [17]: import pandas as pd

```
left_df = pd.DataFrame({
    'id': [1, 2, 3, 4, 5],
    'Company': ['HCL Tech', 'Axis Bank', 'Nestle India', 'Eicher Motors', 'Cipla'],
    'Sector': ['IT', 'Banking', 'FMCG', 'Auto', 'Pharma']
})

right_df = pd.DataFrame(
    {'id': [1, 2, 3, 4, 5],
     'Company': ['Power Grid', 'Tech Mahindra', 'Dr Reddy's Labs', 'IndusInd Bank', 'Hero MotoCorp'],
     'Sector': ['Power', 'IT', 'Pharma', 'Banking', 'Auto']})
```

In [18]: left_df

Out[18]:

	id	Company	Sector
0	1	HCL Tech	IT
1	2	Axis Bank	Banking
2	3	Nestle India	FMCG
3	4	Eicher Motors	Auto
4	5	Cipla	Pharma

In [19]: right_df

Out[19]:

	id	Company	Sector
0	1	Power Grid	Power
1	2	Tech Mahindra	IT
2	3	Dr Reddy's Labs	Pharma
3	4	IndusInd Bank	Banking
4	5	Hero MotoCorp	Auto

Syntax

```
pd.merge(left_df, right_df, on='id')
```

Explanation

- Performs an **inner join** (default join type)
- Joins rows **only where id matches** in both dataframes
- Keeps **all columns** from both dataframes
- Since both dataframes have columns with the same names (Company , Sector), pandas automatically adds suffixes:
 - Company_x → from left_df
 - Company_y → from right_df
 - Sector_x → from left_df
 - Sector_y → from right_df

```
In [20]: # Merge 2 DFs on a key
```

```
print(pd.merge(left_df, right_df, on='id'))
```

	id	Company_x	Sector_x	Company_y	Sector_y
0	1	HCL Tech	IT	Power Grid	Power
1	2	Axis Bank	Banking	Tech Mahindra	IT
2	3	Nestle India	FMCG	Dr Reddy's Labs	Pharma
3	4	Eicher Motors	Auto	IndusInd Bank	Banking
4	5	Cipla	Pharma	Hero MotoCorp	Auto

Merge on Sector

- Merges rows where **Sector values match** in both dataframes.
- Duplicate column names get suffixes _x and _y to indicate their source.

```
In [21]: print(pd.merge(left_df, right_df, on='Sector'))
```

```
print(pd.merge(left_df, right_df, on='Sector'))
```

	id_x	Company_x	Sector	id_y	Company_y
0	1	HCL Tech	IT	2	Tech Mahindra
1	2	Axis Bank	Banking	4	IndusInd Bank
2	4	Eicher Motors	Auto	5	Hero MotoCorp
3	5	Cipla	Pharma	3	Dr Reddy's Labs

```
In [22]: # Merge 2 DFs on multiple keys
```

```
print(pd.merge(left_df, right_df, on=['Sector', 'Company']))
```

Empty DataFrame
Columns: [id_x, Company, Sector, id_y]
Index: []

Output above is empty as there are no rows where BOTH sector and company match.

Left Join on Sector

- Keeps all rows from left_df , and matches data from right_df where Sector names are the same.
- If no match is found, columns from right_df appear as **NaN**.

Explanation

- Every row from left_df is preserved.
- Example: For FMCG , there is no matching Sector in right_df , so merged values become **NaN**.

```
In [23]: # Merge using 'how' argument
```

```
# Left join
```

```
print(pd.merge(left_df, right_df, on='Sector', how='left'))
```

	id_x	Company_x	Sector	id_y	Company_y
0	1	HCL Tech	IT	2.0	Tech Mahindra
1	2	Axis Bank	Banking	4.0	IndusInd Bank
2	3	Nestle India	FMCG	NaN	NaN
3	4	Eicher Motors	Auto	5.0	Hero MotoCorp
4	5	Cipla	Pharma	3.0	Dr Reddy's Labs

Right Join on Sector

- Keeps all rows from `right_df`, and matches data from `left_df` where `Sector` values are the same.
- If no match is found, columns from `left_df` appear as `NaN`.

Explanation

- Every row from `right_df` is preserved.
- Example: If a Sector exists only in `right_df`, its matching values from `left_df` become `NaN`.

```
In [24]: # Right join
```

```
print(pd.merge(left_df, right_df, on='Sector', how='right'))
```

	id_x	Company_x	Sector	id_y	Company_y
0	NaN	NaN	Power	1	Power Grid
1	1.0	HCL Tech	IT	2	Tech Mahindra
2	5.0	Cipla	Pharma	3	Dr Reddy's Labs
3	2.0	Axis Bank	Banking	4	IndusInd Bank
4	4.0	Eicher Motors	Auto	5	Hero MotoCorp

Outer Join on Sector

- Keeps **all rows from both DataFrames**. → If a value exists only in one DataFrame, the other side is filled with `NaN`.

Explanation

- Includes all unique Sectors from both tables: IT, Banking, FMCG, Auto, Pharma, Power.
- Where a Sector matches → data is merged.
- Where no match exists → the unmatched side becomes `NaN`.

```
In [26]: # Outer join
```

```
print(pd.merge(left_df, right_df, on='Sector', how='outer'))
```

	id_x	Company_x	Sector	id_y	Company_y
0	4.0	Eicher Motors	Auto	5.0	Hero MotoCorp
1	2.0	Axis Bank	Banking	4.0	IndusInd Bank
2	3.0	Nestle India	FMCG	NaN	NaN
3	1.0	HCL Tech	IT	2.0	Tech Mahindra
4	5.0	Cipla	Pharma	3.0	Dr Reddy's Labs
5	NaN	NaN	Power	1.0	Power Grid

Inner Join on Sector

- Keeps only rows where `Sector` matches in both DataFrames; all non-matching sectors are dropped.

Explanation

- Only sectors common to both tables remain: **IT, Banking, Auto, Pharma**.
- Sectors like **FMCG** (left only) and **Power** (right only) are removed.

```
In [27]: # Inner join
```

```
print(pd.merge(left_df, right_df, on='Sector', how='inner'))
```

```
   id_x      Company_x  Sector  id_y      Company_y
0    1        HCL Tech     IT      2  Tech Mahindra
1    2        Axis Bank  Banking    4  IndusInd Bank
2    4  Eicher Motors    Auto      5  Hero MotoCorp
3    5         Cipla    Pharma      3  Dr Reddy's Labs
```

Join Type	Rows Kept From	Non-Matching Rows Show As	Common Sectors
left	left_df	NaN on right side	All from left
right	right_df	NaN on left side	All from right
outer	both	NaN where missing	All from both
inner	intersection only	none	Only matching ones

Concatenation (Row-wise)

- Stacks `left_df` and `right_df` **vertically**, one below the other.
- Does **not** match on any column; it simply appends rows and keeps all original columns.

Explanation

- Resulting DataFrame has **10 rows** (5 from each).
- Columns from both DataFrames appear together, and non-overlapping data becomes **NaN** where missing, this only happens when the two DataFrames do NOT have the same set of columns.

```
In [28]: print(left_df)
```

```
   id      Company  Sector
0   1        HCL Tech    IT
1   2        Axis Bank  Banking
2   3  Nestle India    FMCG
3   4  Eicher Motors    Auto
4   5         Cipla    Pharma
```

```
In [29]: print(right_df)
```

```
   id      Company  Sector
0   1        Power Grid  Power
1   2  Tech Mahindra     IT
2   3  Dr Reddy's Labs  Pharma
3   4  IndusInd Bank  Banking
4   5  Hero MotoCorp    Auto
```

```
In [30]: print(pd.concat([left_df, right_df]))
```

```
   id      Company  Sector
0   1        HCL Tech    IT
1   2        Axis Bank  Banking
2   3  Nestle India    FMCG
3   4  Eicher Motors    Auto
4   5         Cipla    Pharma
0   1        Power Grid  Power
1   2  Tech Mahindra     IT
2   3  Dr Reddy's Labs  Pharma
3   4  IndusInd Bank  Banking
4   5  Hero MotoCorp    Auto
```

Concatenation with Keys (Hierarchical Index)

- Stacks `right_df` below `left_df`, but assigns a **multi-level index** using the keys `x` and `y`. → Using `ignore_index=True` removes the hierarchical index and gives a simple continuous index.

Explanation

- With keys: Index shows `x` for rows from `left_df` and `y` for rows from `right_df`.
- With `ignore_index=True`: Hierarchical index is dropped and rows are renumbered from 0 onward.

```
In [31]: print(pd.concat([left_df, right_df], keys=['x', 'y']))
```

```
    id      Company Sector
x 0   1      HCL Tech IT
      2      Axis Bank Banking
      3      Nestle India FMCG
      4      Eicher Motors Auto
      5      Cipla Pharma
y 0   1      Power Grid Power
      2      Tech Mahindra IT
      3  Dr Reddy's Labs Pharma
      4  IndusInd Bank Banking
      5  Hero MotoCorp Auto
```

```
In [33]: print(pd.concat([left_df, right_df], keys=['x', 'y'], ignore_index=True))
```

```
    id      Company Sector
0   1      HCL Tech IT
1   2      Axis Bank Banking
2   3      Nestle India FMCG
3   4      Eicher Motors Auto
4   5      Cipla Pharma
5   1      Power Grid Power
6   2      Tech Mahindra IT
7   3  Dr Reddy's Labs Pharma
8   4  IndusInd Bank Banking
9   5  Hero MotoCorp Auto
```

Concatenation Column-wise (axis=1)

→ Joins the two DataFrames **side by side** instead of stacking vertically.

Explanation

- Columns from both DataFrames are combined.
- Row 1 of `left_df` aligns with Row 1 of `right_df`, Row 2 with Row 2, etc.
- Duplicate column names receive suffixes `_x` and `_y` to avoid conflicts.
- Works like placing two Excel tables next to each other.
- It simply places the two DataFrames side-by-side, matching rows by index, Column names do NOT matter here

```
In [35]: print(pd.concat([left_df, right_df], axis=1))
```

```
    id      Company Sector  id      Company Sector
0   1      HCL Tech IT     1      Power Grid Power
1   2      Axis Bank Banking  2      Tech Mahindra IT
2   3      Nestle India FMCG  3  Dr Reddy's Labs Pharma
3   4      Eicher Motors Auto  4  IndusInd Bank Banking
4   5      Cipla Pharma      5  Hero MotoCorp Auto
```

Concatenating Using append (Vertical)

→ Stacks `right_df` below `left_df` in a **vertical concatenation**.

Explanation

- Adds all rows from both DataFrames sequentially.
- Keeps the original index values, so indices like **0–4 repeat**.
- Columns align automatically because both DataFrames have the same structure.
- NaN appear when the two DataFrames have different column names

Concatenating Multiple DataFrames

→ Stacks four DataFrames sequentially: first `left_df`, then `right_df`, then `left_df` again, and finally `right_df`.

Explanation

- Concatenates all DataFrames from the list `[left_df, right_df, left_df, right_df]`.
- Total rows = **5 + 5 + 5 + 5 = 20**.
- Same columns across all, so the index values repeat.
- You can reset the index afterwards if needed.
- Think of it like copying and pasting the same two tables one below the other, twice

```
In [36]: print(pd.concat([left_df, right_df, left_df, right_df]))
```

	id	Company	Sector
0	1	HCL Tech	IT
1	2	Axis Bank	Banking
2	3	Nestle India	FMCG
3	4	Eicher Motors	Auto
4	5	Cipla	Pharma
0	1	Power Grid	Power
1	2	Tech Mahindra	IT
2	3	Dr Reddy's Labs	Pharma
3	4	IndusInd Bank	Banking
4	5	Hero MotoCorp	Auto
0	1	HCL Tech	IT
1	2	Axis Bank	Banking
2	3	Nestle India	FMCG
3	4	Eicher Motors	Auto
4	5	Cipla	Pharma
0	1	Power Grid	Power
1	2	Tech Mahindra	IT
2	3	Dr Reddy's Labs	Pharma
3	4	IndusInd Bank	Banking
4	5	Hero MotoCorp	Auto

```
In [ ]:
```

TTP Assignment 13

Pandas GroupBy

Use this dataset for All Questions

```
new_portfolio = {
    'Sector': [
        'Energy', 'Telecom', 'Retail', 'Chemicals', 'Logistics',
        'Telecom', 'Energy', 'Retail', 'Logistics', 'Infrastructure'],
    'Company': [
        'Reliance Energy', 'Bharti Airtel', 'DMart', 'Pidilite Industries',
        'Blue Dart',
        'Vodafone Idea', 'ONGC', 'Trent Retail', 'Delhivery', 'Larsen & Toubro'],
    'MarketCap': [
        'Large Cap', 'Large Cap', 'Large Cap', 'Mid Cap', 'Mid Cap',
        'Small Cap', 'Large Cap', 'Mid Cap', 'Mid Cap', 'Large Cap'],
    'Share Price': [
        1320, 920, 4100, 2700, 620,
        14, 205, 1800, 560, 3550],
    'Amount Invested': [
        50000, 30000, 38000, 25000, 16000,
        7000, 22000, 29000, 18000, 40000]
}
```

Question #1 – Create DataFrame From Dictionary

Convert the above dictionary `new_portfolio` into a Pandas DataFrame and name it `df`.

Print:

- The full DataFrame
- The first 5 rows using `df.head()`
- The DataFrame info using `df.info()`

```
In [1]: #The full DataFrame
```

```
import pandas as pd

new_portfolio = {
    'Sector': [
        'Energy', 'Telecom', 'Retail', 'Chemicals', 'Logistics',
        'Telecom', 'Energy', 'Retail', 'Logistics', 'Infrastructure'],
    'Company': [
        'Reliance Energy', 'Bharti Airtel', 'DMart', 'Pidilite Industries',
        'Vodafone Idea', 'ONGC', 'Trent Retail', 'Delhivery', 'Larsen & Toubro'],
    'MarketCap': [
        'Large Cap', 'Large Cap', 'Large Cap', 'Mid Cap', 'Mid Cap',
        'Small Cap', 'Large Cap', 'Mid Cap', 'Mid Cap', 'Large Cap'],
    'Share Price': [
        1320, 920, 4100, 2700, 620,
        14, 205, 1800, 560, 3550],
    'Amount Invested': [
        50000, 30000, 38000, 25000, 16000,
        7000, 22000, 29000, 18000, 40000]
}

df = pd.DataFrame(new_portfolio)
df
```

Out[1]:

	Sector	Company	MarketCap	Share Price	Amount Invested
0	Energy	Reliance Energy	Large Cap	1320	50000
1	Telecom	Bharti Airtel	Large Cap	920	30000
2	Retail	DMart	Large Cap	4100	38000
3	Chemicals	Pidilite Industries	Mid Cap	2700	25000
4	Logistics	Blue Dart	Mid Cap	620	16000
5	Telecom	Vodafone Idea	Small Cap	14	7000
6	Energy	ONGC	Large Cap	205	22000
7	Retail	Trent Retail	Mid Cap	1800	29000
8	Logistics	Delhivery	Mid Cap	560	18000
9	Infrastructure	Larsen & Toubro	Large Cap	3550	40000

```
In [2]: #The first 5 rows using df.head()
```

```
print(df.head(5))
```

	Sector	Company	MarketCap	Share Price	Amount Invested
0	Energy	Reliance Energy	Large Cap	1320	50000
1	Telecom	Bharti Airtel	Large Cap	920	30000
2	Retail	DMart	Large Cap	4100	38000
3	Chemicals	Pidilite Industries	Mid Cap	2700	25000
4	Logistics	Blue Dart	Mid Cap	620	16000

```
In [3]: #The DataFrame info using df.info()
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Sector            10 non-null    object  
 1   Company           10 non-null    object  
 2   MarketCap         10 non-null    object  
 3   Share Price       10 non-null    int64  
 4   Amount Invested  10 non-null    int64  
dtypes: int64(2), object(3)
memory usage: 532.0+ bytes
None
```

Question #2 – GroupBy Keys and Values

Group the DataFrame by **MarketCap** and print the `.groups` dictionary.

Interpret which sectors fall under each MarketCap.

```
In [4]: df.groupby(['MarketCap', 'Sector']).groups
```

```
Out[4]: {('Large Cap', 'Energy'): [0, 6], ('Large Cap', 'Infrastructure'): [9],
          ('Large Cap', 'Retail'): [2], ('Large Cap', 'Telecom'): [1], ('Mid Cap',
          'Chemicals'): [3], ('Mid Cap', 'Logistics'): [4, 8], ('Mid Cap', 'Retail'):
          [7], ('Small Cap', 'Telecom'): [5]}
```

Question #3 – GroupBy Sector and Inspect Mini-DataFrames

Group by **Sector** and loop through each group. Print the sector name and the mini-DataFrame belonging to that sector.

```
In [5]: grouped = df.groupby('Sector')

for name, group in grouped:
    print(name)
    print(group)

Chemicals
      Sector          Company MarketCap Share Price  Amount Invested
3  Chemicals  Pidilite Industries   Mid Cap        2700       25000
Energy
      Sector          Company MarketCap Share Price  Amount Invested
0  Energy   Reliance Energy  Large Cap       1320      50000
6  Energy           ONGC  Large Cap        205       22000
Infrastructure
      Sector          Company MarketCap Share Price  Amount Investe
d
9  Infrastructure  Larsen & Toubro  Large Cap        3550       4000
0
Logistics
      Sector          Company MarketCap Share Price  Amount Invested
4  Logistics     Blue Dart   Mid Cap        620       16000
8  Logistics    Delhivery   Mid Cap        560       18000
Retail
      Sector          Company MarketCap Share Price  Amount Invested
2  Retail         DMart  Large Cap       4100      38000
7  Retail    Trent Retail   Mid Cap       1800      29000
Telecom
      Sector          Company MarketCap Share Price  Amount Invested
1  Telecom    Bharti Airtel  Large Cap       920      30000
5  Telecom  Vodafone Idea  Small Cap        14       7000
```

Question #4 – GroupBy Multiple Columns

Group the dataset by both **MarketCap** and **Sector**.

Print the `.groups` output and count how many combinations exist.

```
In [6]: grouped = df.groupby(["MarketCap", "Sector"])
print(grouped.groups)
df.groupby(["MarketCap", "Sector"]).ngroups
```

```
{('Large Cap', 'Energy'): [0, 6], ('Large Cap', 'Infrastructure'): [9],
('Large Cap', 'Retail'): [2], ('Large Cap', 'Telecom'): [1], ('Mid Cap',
'Chemicals'): [3], ('Mid Cap', 'Logistics'): [4, 8], ('Mid Cap', 'Retai
l'): [7], ('Small Cap', 'Telecom'): [5]}
```

Out[6]: 8

Question #5 – Extract Groups Using `get_group()`

Use `df.groupby('MarketCap')` to:

- Print all rows where MarketCap is **Mid Cap**
- Print all rows where MarketCap is **Large Cap**

```
In [7]: grouped = df.groupby('MarketCap')

print(grouped.get_group('Mid Cap'))
print("-----")
print(grouped.get_group('Large Cap'))

      Sector           Company MarketCap  Share Price  Amount Invested
3  Chemicals  Pidilite Industries   Mid Cap     2700      25000
4  Logistics          Blue Dart   Mid Cap      620      16000
7    Retail        Trent Retail   Mid Cap     1800      29000
8  Logistics       Delhivery   Mid Cap      560      18000
-----
      Sector           Company MarketCap  Share Price  Amount Investe
d
0      Energy  Reliance Energy  Large Cap     1320      5000
0
1    Telecom      Bharti Airtel  Large Cap      920      3000
0
2      Retail         DMart  Large Cap     4100      3800
0
6      Energy          ONGC  Large Cap      205      2200
0
9  Infrastructure  Larsen & Toubro  Large Cap     3550      4000
0
```

Question #6 – Aggregation: Mean & Sum

Group by **Sector** and compute the following for **Amount Invested**:

- Sum using `np.sum`
- Mean using `np.mean`

Print a nicely formatted table of both.

```
In [8]: import numpy as np
import warnings
warnings.filterwarnings("ignore")

grouped = df.groupby('Sector')

print(grouped['Amount Invested'].agg([np.sum, np.mean]))
```

Sector	sum	mean
Chemicals	25000	25000.0
Energy	72000	36000.0
Infrastructure	40000	40000.0
Logistics	34000	17000.0
Retail	67000	33500.0
Telecom	37000	18500.0

Question #7 – Apply Multiple Aggregations Together

Group by **MarketCap** and apply aggregations: `[np.sum, np.mean]` on **Share Price**
 Round results to 2 decimals and print.

```
In [9]: grouped = df.groupby('MarketCap')
print(np.round(grouped['Share Price'].agg([np.sum, np.mean]),2))
```

	sum	mean
MarketCap		
Large Cap	10095	2019.0
Mid Cap	5680	1420.0
Small Cap	14	14.0

Question #8 – Apply Custom Function Using transform()

Write a **z-score** function that standardizes values. Apply it on the columns **Share Price** and

Amount Invested using `.transform()` inside a MarketCap groupby.

Print the resulting standardized DataFrame.

```
In [10]: def z_score(x): return (x - x.mean()) / x.std()
df[["SharePrice_z", "AmountInvested_z"]] = (
    df.groupby("MarketCap")[["Share Price", "Amount Invested"]]
    .transform(z_score)
)
print(df)
```

	Sector	Company	MarketCap	Share Price	\
0	Energy	Reliance Energy	Large Cap	1320	
1	Telecom	Bharti Airtel	Large Cap	920	
2	Retail	DMart	Large Cap	4100	
3	Chemicals	Pidilite Industries	Mid Cap	2700	
4	Logistics	Blue Dart	Mid Cap	620	
5	Telecom	Vodafone Idea	Small Cap	14	
6	Energy	ONGC	Large Cap	205	
7	Retail	Trent Retail	Mid Cap	1800	
8	Logistics	Delhivery	Mid Cap	560	
9	Infrastructure	Larsen & Toubro	Large Cap	3550	

	Amount Invested	SharePrice_z	AmountInvested_z
0	50000	-0.409383	1.322876
1	30000	-0.643650	-0.566947
2	38000	1.218778	0.188982
3	25000	1.246701	0.495434
4	16000	-0.779188	-0.990867
5	7000	NaN	NaN
6	22000	-1.062404	-1.322876
7	29000	0.370114	1.156012
8	18000	-0.837627	-0.660578
9	40000	0.896660	0.377964

Question #9 – Filter Groups by Size

Use `.filter()` to keep only those MarketCap groups that have **2 or fewer** companies.

Print the filtered DataFrame.

```
In [17]: filtered_df=df.groupby('MarketCap').filter(lambda x: len(x) <= 2)
print(filtered_df)
```

```
      Sector          Company MarketCap Share Price Amount Invested \
5  Telecom    Vodafone Idea  Small Cap           14        7000

   SharePrice_z  AmountInvested_z
5             NaN                 NaN
```

Question #10 – Merge Two Synthetic DataFrames

Create a second small DataFrame with columns **Sector** and **Rating** (assign any rating values). Merge it with the main DataFrame on **Sector** using:

- Inner join
- Left join

Print both merged outputs.

```
In [45]: rating_df = pd.DataFrame({
    "Sector": ["IT", "Banking", "Pharma"],
    "Rating": ["A", "B", "A+"]
})
print(rating_df)
```

```
      Sector Rating
0       IT      A
1  Banking      B
2    Pharma     A+
```

```
In [46]: inner_merged = pd.merge(
    df,
    rating_df,
    on="Sector",
    how="inner"
)
print("INNER JOIN RESULT")
print(inner_merged)
```

```
INNER JOIN RESULT
Empty DataFrame
Columns: [Sector, Company, MarketCap, Share Price, Amount Invested, ShareP
rice_z, AmountInvested_z, Rating]
Index: []
```

```
In [44]: left_merged = pd.merge(  
    df,  
    rating_df,  
    on="Sector",  
    how="left"  
)  
  
print("\nLEFT JOIN RESULT")  
print(left_merged)
```

LEFT JOIN RESULT

	Sector	Company	MarketCap	Share Price	\
0	Energy	Reliance Energy	Large Cap	1320	
1	Telecom	Bharti Airtel	Large Cap	920	
2	Retail	DMart	Large Cap	4100	
3	Chemicals	Pidilite Industries	Mid Cap	2700	
4	Logistics	Blue Dart	Mid Cap	620	
5	Telecom	Vodafone Idea	Small Cap	14	
6	Energy	ONGC	Large Cap	205	
7	Retail	Trent Retail	Mid Cap	1800	
8	Logistics	Delhivery	Mid Cap	560	
9	Infrastructure	Larsen & Toubro	Large Cap	3550	

	Amount Invested	SharePrice_z	AmountInvested_z	Rating
0	50000	-0.409383	1.322876	NaN
1	30000	-0.643650	-0.566947	NaN
2	38000	1.218778	0.188982	NaN
3	25000	1.246701	0.495434	NaN
4	16000	-0.779188	-0.990867	NaN
5	7000	NaN	NaN	NaN
6	22000	-1.062404	-1.322876	NaN
7	29000	0.370114	1.156012	NaN
8	18000	-0.837627	-0.660578	NaN
9	40000	0.896660	0.377964	NaN

```
In [ ]:
```

Understand Market Trends and Relationships:

Analyze historical data to identify patterns, trends, and relationships between assets for better decision-making.

Test and Improve Strategies:

Use past data to check if a trading idea works and refine strategies for better performance.

Make Data-Driven Decisions:

Replace guesswork with factual insights to ensure informed and reliable trading actions.

Manage Risk and Spot Opportunities:

Measure potential losses and detect unusual price movements to safeguard and capitalize on trades.

Simplify Complex Data:

Turn large datasets into simple metrics like averages and ranges for quick understanding and execution.

courses.tradingheads.com
kedar.shah@immediaspacem.com

Descriptive Statistics : is about describing or summarizing large datasets using quantitative or visual tools. You can think of stock market indexes as a descriptive statistics of sort for market as a whole.

There are 2 major classes of Descriptive Statistics: [measures of central tendency](#) and [measures of variability / dispersion](#). Measures of central tendency look at where the bulk of the data lies. In other words, averages. They are the key building block of many technical indicators, most notably moving averages. [MEAN – MEDIAN – MODE](#) are the most common measures of central tendency.

[Measures of Variability](#) describe how spread out or scattered the values in a dataset are around the central tendency.

Inferential Statistics : builds on descriptive statistics to draw conclusions or inferences based on that data.

- MEAN / ARITHMETIC MEAN
 - MEDIAN
 - MODE
- GEOMETRIC MEAN

Year	Annual Return (%)
2014	7.46
2015	18.58
2016	14.78
2017	11.84
2018	-0.77
2019	8.32
2020	-3.82
2021	18.87
2022	-8.51
2023	11.58

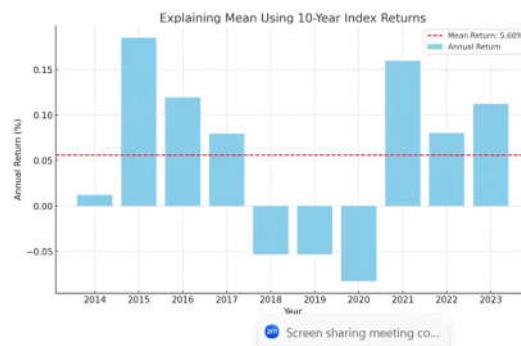
$$\text{Mean} = \frac{\sum_{i=1}^n x_i}{n}$$

Where:

- x_i = each individual value in the dataset.
- n = total number of values in the dataset.
- \sum = summation symbol, meaning "add all values."

- Step-by-Step Mean Calculation:
- Sum of Annual Returns:
 $7.46 + 18.58 + 14.78 + 11.84 - 0.77 + 8.32 - 3.82 + 18.87 - 8.51 + 11.58 = 56$
 - Number of Years (n): 10
 - Mean Calculation:

$$\text{Mean} = \frac{\text{Sum of Returns}}{\text{Number of Years}} = \frac{56.04}{10} = 5.60\%$$



The **median** is a measure of central tendency that represents the **middle value** of a dataset when it is ordered from smallest to largest. It divides the data into two equal halves, with 50% of the values below it and 50% above.

Example 1: Odd Number of Values

Dataset: [3, 7, 10, 15, 20] ($n = 5$).

1. Sorted Dataset: Already sorted.
2. Position of the Median:

$$\text{Median} = \text{Value at position } \frac{5+1}{2} = \text{Value at position 3}$$

3. Median:

$$\text{Median} = 10$$

Example 2: Even Number of Values

Dataset: [3, 7, 10, 15] ($n = 4$).

1. Sorted Dataset: Already sorted.
2. Median:

$$\text{Median} = \frac{\text{Value at position } \frac{4}{2} + \text{Value at position } (\frac{4}{2} + 1)}{2}$$

$$\text{Median} = \frac{\text{Value at position 2} + \text{Value at position 3}}{2} = \frac{7 + 10}{2} = 8.5$$

Mode Calculation:

The mode is the most frequently occurring value in the dataset.

In this dataset, the value **20** appears twice, while all other values appear once.

Mode: 20

Dataset:

Value
10
20
20
30
40

Most Frequent Value:

The mode identifies the value with the highest frequency of occurrence.

Applicable to All Data Types:

Works for numerical, categorical, and ordinal data.

Uniqueness:

A dataset can have:

- **One mode:** Unimodal dataset.
- **Two modes:** Bimodal dataset.
- **More than two modes:** Multimodal dataset.
- **No mode:** All values occur with the same frequency.

courses.tradingheads.com
kedar.shah@nimldatacomm.com

Categorical / Qualitative Trading Data:

- The mode identifies the most common trading category. Example: Candlestick outcomes like Green, Green, Red, Green → **Mode: Green**.

courses.tradingheads.com
kedar.shah@nimldatacomm.com

Most Frequent Value:

- Useful for spotting the most repeated price level or return class (e.g., consolidation zones, frequent daily return buckets).

Non-Numerical Market Data:

- Mode works even when mean/median cannot be used (e.g., trend labels, chart patterns, signal types).

Strengths of the Mode:

- Highlights the most frequently occurring market behavior.
- Easy to compute and interpret.
- Effective for large datasets or when mean/median offer little insight (e.g., categorical signals).

Limitations of the Mode:

- May not exist if market data has no repeated values.
- Not always representative when multiple dominant zones or regimes exist.

- The **geometric mean** is a measure of central tendency that calculates the average rate of return or growth for a dataset, especially when values are compounded or multiplicative. Unlike the arithmetic mean, it considers the cumulative effect of changes, making it ideal for analyzing percentages, ratios, and rates over time.

Geometric Mean Formula

For a dataset with n values x_1, x_2, \dots, x_n :

$$\text{Geometric Mean (GM)} = \sqrt[n]{x_1 \cdot x_2 \cdot \dots \cdot x_n}$$

Alternatively, for growth rates or percentages:

$$GM = \left(\prod_{i=1}^n (1 + r_i) \right)^{\frac{1}{n}} - 1$$

Where r_i is the return in decimal form (e.g., 5% = 0.05).

Steps to Calculate Geometric Mean

- Convert Data:**
 - If working with percentages or returns, express them as decimals.
- Multiply All Values:**
 - Compute the product of all data points.
- Take the n -th Root:**
 - Raise the product to the power of $\frac{1}{n}$ (or use the exponential/logarithm approach for ease).
- Interpret:**
 - The result represents the average growth rate over the dataset.

The geometric mean of a portfolio's various returns over a set number of periods gives us the rate that portfolio would need to return to produce the same final value if each return was the same.

Or, put more simply, it is the compound rate of return.

Time series datasets ruin the interpretability of the arithmetic mean.

Imagine a stock whose return is -50% the first year and 75% the second year. In this situation, the arithmetic mean of the two returns is simply $((-50\% / 100) + (75\% / 100)) / 2 = .125$, or 12.5%. Clearly, a 12.5% return is not close to being representative of the investment's actual performance.

Worse, if you had invested \$100 in that stock, you would end up with \$50 after year one, and \$87.50 after year two — ultimately a losing investment, despite the positive return the simple average gave us!

This equation is similar to the arithmetic mean, but instead of summing the observations and dividing by the number of observations, find the product of the observations by multiplying them together and then raising that product to the $1/n$ th root.

Because the observations in the dataset are really scaling factors, zeros and negative numbers, both common results when dealing with investment returns, are problematic. To cope with this, add 1 to each of the returns before plugging them into the equation, and then subtract 1 at the end.

For our imaginary stock example, we can calculate our geometric mean as follows:

$$GM = (((1 + -50\%) \times (1 + 75\%))^{1/2}) - 1$$

$$GM = ((.5 \times 1.75)^{1/2}) - 1$$

$$GM = (.93541) - 1$$

$$GM = -.06458$$

$$GM \approx -6.46\%$$

Computing the geometric mean returns a compound annual return of -6.46%. That would turn our \$100 investment into \$87.50 over two years. In other words, using the geometric mean preserves the compound nature of investment returns.

C	D	E	F	G	H	I	J	K	L	M	N
		4	1	3	0.25						
		Return in %	P&L	Capital							
				1000							
Year 1		-50	-500	500	-500	-50%	-0.5	0.50		0.5	
Year 2		75	375	875	375	75%	0.75	1.75		1.75	
Arithmetic Mean		12.5		-125							
		#REF!									
			4								
Arithmetic Mean		#REF!									
						0.875					
						0.935414					
						-0.06459					
						-6.45857					



Importance Of Geometric Mean



Key Uses of Geometric Mean

1. Finance:
 - Calculate average annual returns for investments or portfolios.
 - Measure compound interest over multiple periods.
2. Growth Rates:
 - Determine average growth in populations, sales, or production over time.
3. Proportional Comparisons:
 - Analyze relative changes in datasets (e.g., comparing indices or prices).

Why Geometric Mean Is Important ?

Handles Compounding:

- Accounts for cumulative effects of changes, unlike the arithmetic mean.



Prevents Distortion:

- Avoids overestimating average growth when there are fluctuations.

Ideal for Percentages/Ratios:

- Works well when dealing with rates of change over time.



Key Differences



Difference Between Geometric and Arithmetic Mean

Aspect	Arithmetic Mean	Geometric Mean
Definition	Simple average of values	Average rate of change or growth
Use Case	Additive data (e.g., scores)	Multiplicative data (e.g., returns)
Effect of Outliers	Sensitive to outliers	Less sensitive to outliers

Just like measures of central tendency correspond to expected returns in academic finance, there is a similar mapping for dispersion: volatility or risk. In fact, the notion that statistical dispersion equals risk is so deeply ingrained in the investment community that we actually use dispersion measures like standard deviation as a stand-in for risk itself.

This should make sense intuitively. If the Dow's mean is an approximation of its typical return in a given year in our dataset, dispersion can help us measure the risk that it missed that mark for any given year.

In other words, dispersion measures the volatility of the return series around its average.

Variance and standard deviation are the two most common measures of dispersion. It makes sense to talk about them in tandem because they are very closely related.

Sample variance is defined as the average squared deviation from the mean. Symbolically, variance is defined as:

$$s^2 = \frac{\sum (x_i - \bar{x})^2}{n-1}$$

1	2007	6%	0.0036
2	2008	-34%	0.1156
3	2009	19%	0.0361
4	2010	11%	0.0121
5	2011	6%	0.0036
6	2012	7%	0.0049
7	2013	27%	0.0729
8	2014	8%	0.0064
9	2015	-2%	0.0004
10	2016	13%	0.0169
11	2017	25%	0.0625
		7.82%	0.335 Sum of Sq SD

$$s^2 = \frac{.3350}{11-1} \approx .0335 \text{ or } 3.35\%$$

Finding the sum of the average square deviation from the mean

The square root of variance is how to calculate standard deviation. Symbolically, standard deviation is defined as:

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n-1}}$$

Taking the square root of the variance .0335 equals .1830 or 18.3%.

This result explains that for the 11 years from 2007 through 2017, the Dow Jones Industrial Average's standard deviation of annual returns was roughly 18.3%.

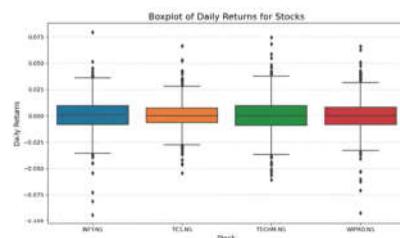
Said another way, roughly 68% of times, the annual return will fall somewhere between roughly 18.3% above and below the average annual return of 7.8%. That is, somewhere between -10.5% and 26.1%.

One note about standard deviation. Just as the mean is highly influenced by outliers in the data, so too is standard deviation. To overcome this, the median is usually used as the measure of central tendency and the interquartile range is used for measures of dispersion.



BOX & WHISKER PLOTS

BOX PLOT

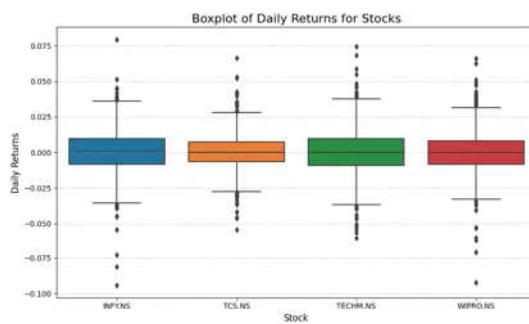


A boxplot is a statistical visualization used to display the distribution of a dataset, highlighting its central tendency, variability, and outliers. It is particularly useful for comparing distributions across multiple categories. Here's a detailed explanation of how to read and analyze a boxplot:

Common Applications of Boxplots

- Outlier Detection: Quickly identify abnormal values.
- Comparison Across Groups: Compare performance, variability, or distributions of multiple datasets.
- Data Quality Check: Detect anomalies or inconsistencies in data collection.

WHISKER PLOTS



- WIDER BOX OF TECH MAHINDRA & INFOSYS SUGGEST THEY HAVE WIDER RANGE OF DAILY RETURNS WHEN COMPARED TO TCS & WIPRO, THIS SUGGEST THEY BOTH ARE MORE VOLATILE
- THEY ARE ALL APPROXIMATELY CENTERED AROUND THE SAME SLIGHTLY +VE RETURNS INDICATING THAT NONE SYSTEMATICALLY OUTPERFORMED THE OTHERS

A **scatterplot** is a type of graph that displays individual data points on a two-dimensional plane, showing the relationship between two numerical variables. Each point represents one observation in the dataset, with its position determined by the values of the variables.

Key Components of a Scatterplot:

1. X-Axis:

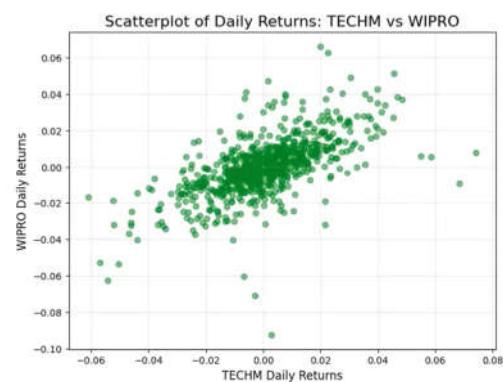
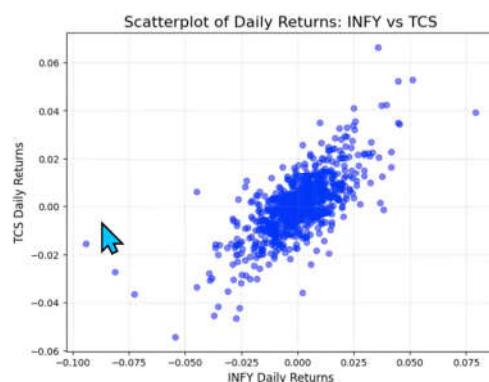
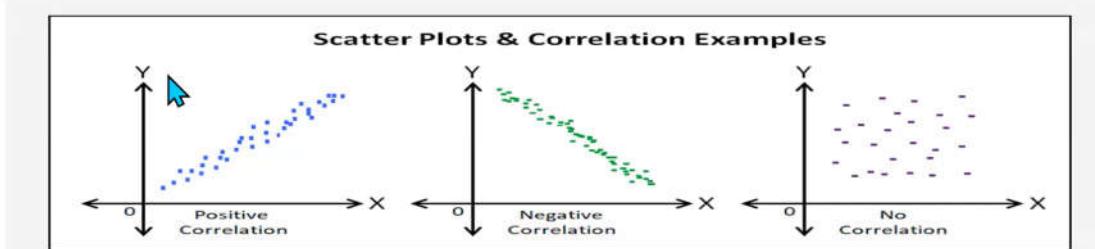
- Represents the independent variable (predictor).

2. Y-Axis:

- Represents the dependent variable (outcome).

3. Data Points:

- Each dot represents a single observation with its x and y coordinates corresponding to its values.



A **scatterplot** is a type of graph that displays individual data points on a two-dimensional plane, showing the relationship between two numerical variables. Each point represents one observation in the dataset, with its position determined by the values of the variables.

Key Components of a Scatterplot:

1. X-Axis:

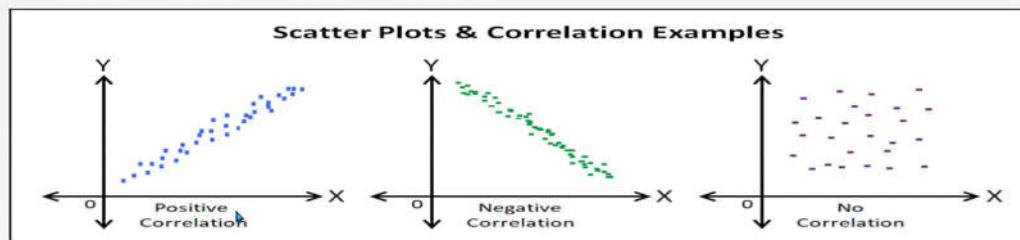
- Represents the independent variable (predictor).

2. Y-Axis:

- Represents the dependent variable (outcome).

3. Data Points:

- Each dot represents a single observation with its x and y coordinates corresponding to its values.



1. Positive Relationship:

- Points trend upward as X increases, Y also increases.
- Example: Higher education levels (X) often correspond to higher incomes (Y).

2. Negative Relationship:

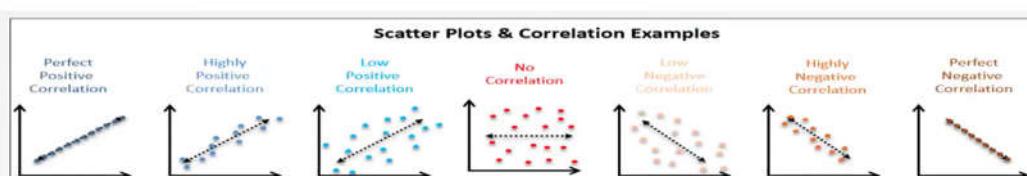
- Points trend downward as X increases, Y decreases.
- Example: Increase in distance from the city center (X) may lead to lower property prices (Y).

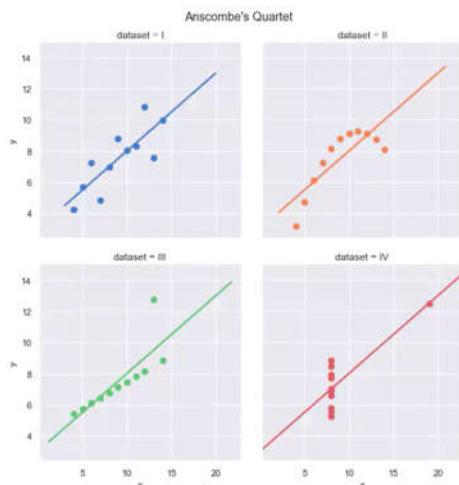
3. No Relationship:

- Points are randomly scattered with no visible trend.
- Example: Shoe size (X) and IQ (Y) have no connection.

4. Outliers:

- Points far from the general trend or cluster may indicate unusual observations or errors.





[Anscombe's Quartet](#) is a collection of four data sets with nearly exactly the same summary statistics. It was developed by statistician Francis Anscombe in 1973 to illustrate the importance of plotting data and the impact of outliers on statistical analysis.

The quartet is still often used to illustrate the importance of looking at a set of data graphically before starting to analyze according to a particular type of relationship, and the inadequacy of basic statistic properties for describing realistic datasets

A **histogram** is a graphical representation of the distribution of a dataset used in continuous numerical values. It uses bars to show the frequency of data points within specified ranges (called bins or intervals). It helps visualize the shape, spread, and central tendency of the data.

Key Features of a Histogram:

1. Bars Represent Frequency:

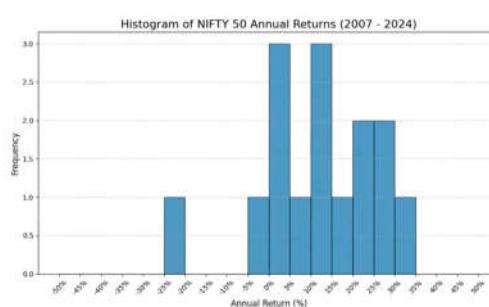
1. Each bar's height shows how many data points fall within the corresponding interval (bin).

2. X-Axis (Bins):

1. Represents the range of data values, divided into intervals.

3. Y-Axis (Frequency):

1. Represents the count of data points in each bin.



1. Shape:

1. **Symmetrical:** Data is evenly spread around the center.
2. **Skewed Left:** Longer tail on the left; data concentrated on the right.
3. **Skewed Right:** Longer tail on the right; data concentrated on the left.

2. Spread:

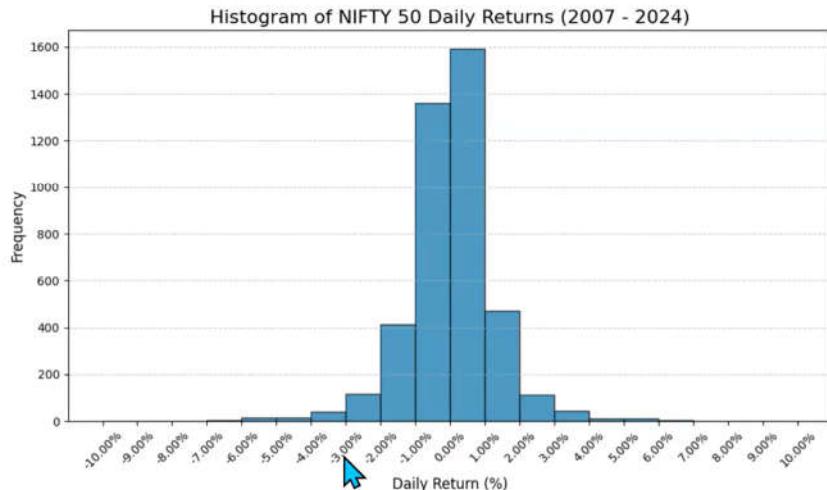
1. Look at the width of the range to understand variability.

3. Outliers:

1. Unusually tall or isolated bars indicate possible outliers.

Conclusion:

- A histogram is a powerful tool for understanding data distribution. It provides insights into the central tendency, variability, and overall shape of the data, making it indispensable in data analysis and decision-making.



PROBABILITY

Probability measures the extent to which an event is likely to occur

It is measured in a scale of 0 and 1. A probability of 0 indicates that an event is more or less impossible, while a probability of 1 indicates proximate certainty. So, probability can never be > 1

The probability of an event E is calculated as:

$$P(E) = \frac{\text{Number of Favorable Outcomes}}{\text{Total Number of Possible Outcomes}}$$

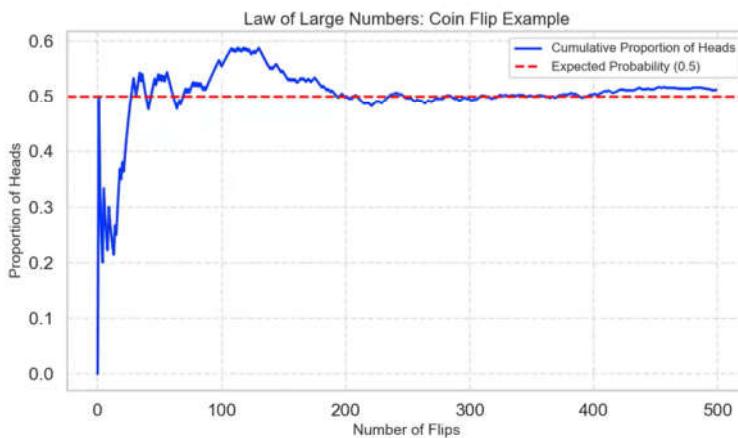
Example: Probability of Getting Heads in a Coin Flip

- Favorable Outcome: Heads (1 outcome).
- Total Possible Outcomes: Heads or Tails (2 outcomes).

$$P(\text{Heads}) = \frac{\text{Favorable Outcomes (Heads)}}{\text{Total Outcomes (Heads, Tails)}} = \frac{1}{2} = 0.5$$

So, the probability of getting heads is 0.5.

The Law of Large Numbers (LLN) states that as the number of trials or observations increases, the average of the observed outcomes will converge to the expected value or true probability.



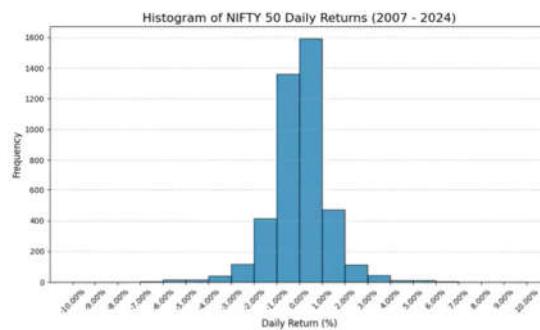
The outcome of our coin toss is an example of a random variable because the result is dictated by a random phenomenon. While we know that our coin flip will end up resulting in either heads or tails, the result we get on any given flip is random. Over many coin flips, though, the law of large numbers dictates that the average of the results will end up close to the expected probability of 0.5. The last slide figure shows the occurrence of heads in a simulation of 500 flips of a fair coin. While the outcome of any given flip is random, the occurrence of heads over the entire experiment (the empirical probability of flipping a head) ultimately converges with the theoretical probability of flipping a head as the number of coin flips increases.

i.i.d. (independent and identically distributed)

One important assumption in our coin toss example is that our random variables are independent and identically distributed (i.i.d.)

Independence means that the occurrence of one event does not affect the probability of the occurrence of the other. The event of flipping heads is not impacted by how many heads were flipped in prior coin tosses. So even say, after flipping nine heads, the probability of flipping heads on the 10th toss is still 0.5. The i.i.d. assumption is a key assumption of many statistical calculations. A large body of academic research has shown that financial return series data is not an i.i.d. process.

This histogram of returns is an example of a probability distribution, a function that provides the probabilities of occurrence of different possible outcomes for a random variable. As the number of observations increases the histogram of returns begins to look more like a “bell curve”.



Key Properties

1. Symmetry:

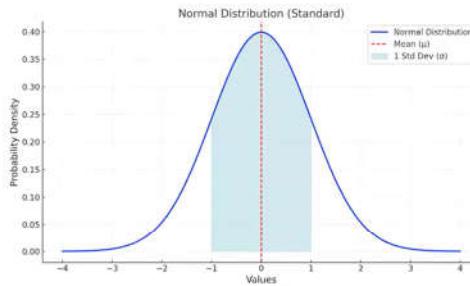
- The distribution is symmetric around the mean (μ).
- The left and right sides of the curve are mirror images.

2. Bell Shape:

- Most values cluster around the mean, and probabilities decrease as you move away from the mean.

3. 68-95-99.7 Rule:

- Approximately:



Normal Distribution

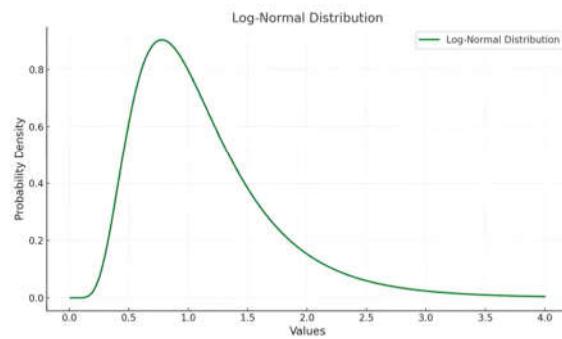
The **normal distribution**, also known as the **Gaussian distribution**, is a symmetric, bell-shaped probability distribution that describes many natural phenomena. It is defined by two parameters:

- Mean (μ):** Determines the center of the distribution.
- Standard Deviation (σ):** Determines the spread or width of the distribution.

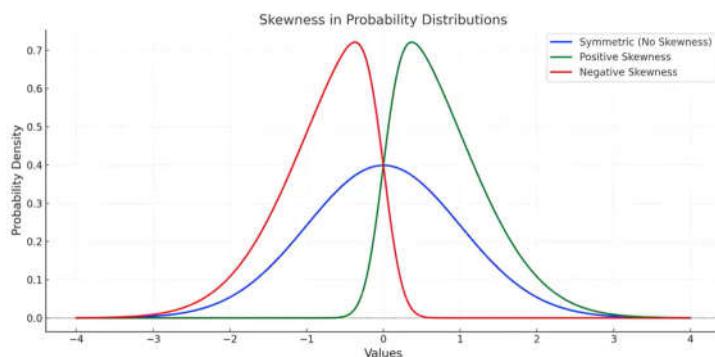
The log-normal distribution describes a random variable whose logarithm is normally distributed. Unlike the normal distribution, the log-normal distribution is positively skewed and defined only for positive values.

Skewed to the right (long tail towards larger values)

Often used to model stock prices, income distributions, and natural phenomena where values cannot be negative.



Skewness and kurtosis are statistics that describe the way a probability distribution looks. Skewness is the degree to which returns are asymmetric around the mean. The skewness is positive when the right tail of the distribution is larger than the left side (that is, it is stretched to the right). A Normal distribution has a skewness of zero. A standard normal distribution has a mean = 0, standard deviation = 1 and Kurtosis = 3



1. No Skewness (Symmetric):

1. The left and right sides of the distribution are mirror images.
2. The mean, median, and mode are all equal.
3. Example: Normal distribution.

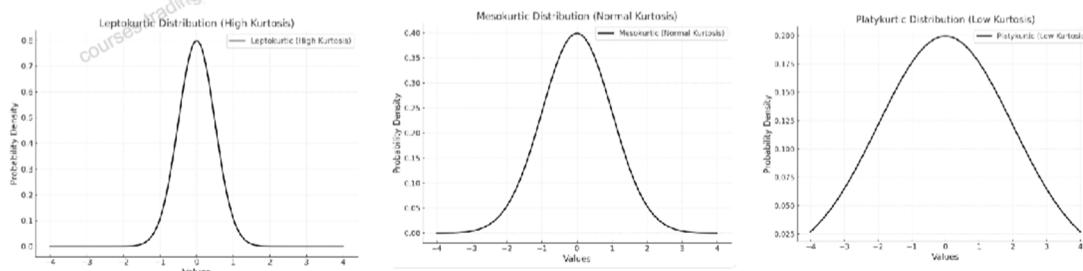
2. Positive Skewness:

1. The tail is longer on the right (values are stretched toward higher numbers).
2. The mean is greater than the median.
3. Example: Income distribution where a few high earners pull the average up.

3. Negative Skewness:

1. The tail is longer on the left (values are stretched toward lower numbers).
2. The mean is less than the median.
3. Example: Exam scores where most students score high but a few score very low.

Kurtosis measures the degree to which returns show up in the tails of the distributions. A normal distribution has a kurtosis of 3, distributions with higher Kurtosis have more returns out in the tails, Kurtosis is often measured relative to the Normal distribution – in this case, a distribution with excess kurtosis of 1 would actually have Kurtosis of 4. (deduct 3 from it). In general a Lower Kurtosis is seen as a good thing because the size and occurrence of tail events are lower.



1. Mesokurtic:

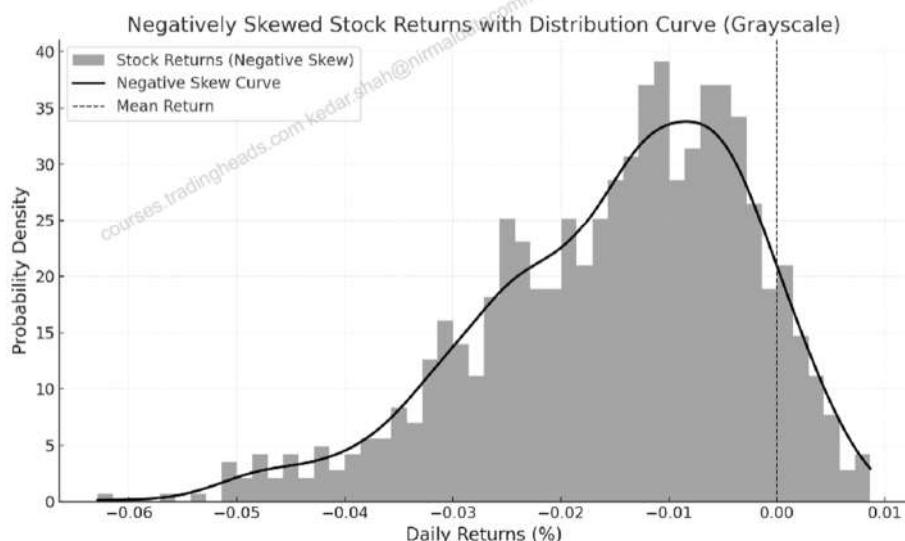
1. The distribution has a kurtosis close to that of a normal distribution (kurtosis = 3 in statistical terms but often reported as "excess kurtosis = 0").
2. Tails and peak are of average size.
3. Example: Normal distribution.

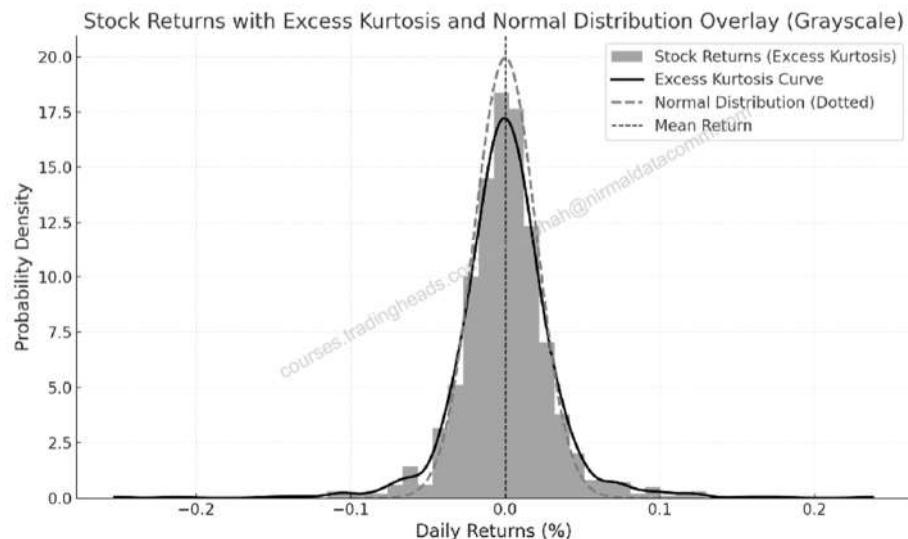
2. Leptokurtic (High Kurtosis):

1. The distribution has a sharper peak and fatter tails.
2. Indicates frequent extreme values (more outliers).
3. Example: Stock market returns during volatile periods.

3. Platykurtic (Low Kurtosis):

1. The distribution is flatter with thinner tails.
2. Indicates fewer extreme values compared to a normal distribution.
3. Example: Uniform distribution.





- Technical Analysis (TA) divides into two broad categories: objective and subjective. Subjective TA is comprised of analysis methods and patterns that are not precisely defined. As a consequence, a conclusion derived from a subjective method reflects the private interpretations of the analyst applying the method can differ. Therefore, subjective methods are untestable.
- In contrast, objective methods are clearly defined and its signals are unambiguous when applied to market data. This makes it possible to simulate the rule/method to be tested on historical data. This is called Back testing. This makes it possible to find out which objective methods are effective and which are not.
- The acid test for distinguishing an objective from a subjective method is the programmability criterion: it is objective if it can be implemented as a computer program that produces clear market positions (long, short or neutral)

A rule is a function that transforms one or more items of information, referred to as the rule's input, into the rule's output, which is a recommended market position (e.g., long, short, neutral). Input(s) consists of one or more financial market time series. The rule is defined by one or more mathematical and logical operators that convert the input time series into a new time series that consists of the sequence of recommended market position (long, short, out-of-the-market). The output is typically represented by a signed number (e.g., +1 or -1). This book adopts the convention of assigning positive values to indicate long positions and negative values to indicate shorts position. The process by which a rule transforms one or more input series into an output series is illustrated in Figure 1.1.

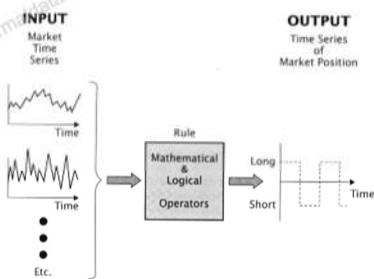


FIGURE 1.1 TA rule transforms input time series into a time series of market position.

A rule is said to generate a *signal* when the value of the output series changes. A signal calls for a change in a previously recommended market position. For example a change in output from +1 to -1 would call for closing a previously held long position and the initiation of a new short position. Output values need not be confined to {+1, -1}. A complex rule, whose output spans the range {+10, -10}, is able to recommend positions that vary in size. For example, an output of +10 might indicate that 10 long positions are warranted, such as long 10 contracts of copper. A change in the output from +10 to +5 would call for a reduction in the long position from 10 contracts to 5 (i.e., sell 5).

1. Trend Following

Trade in the direction of the prevailing trend. Buy strength, sell weakness. Let winners run and cut losers quickly.

2. Mean Reversion

Price tends to return to its average. Buy when price is too low, sell when price is too high. Works best in range-bound markets.

3. Counter-Trend Trading

Take trades *against* the current move, expecting exhaustion or reversal. Sell into sharp rallies, buy into sharp drops.

Criteria	High-Frequency Trading (HFT)	Medium-Frequency Trading (MFT)	Low-Frequency Trading (LFT)
Trade Frequency	Thousands per second	Few trades per hour or day	Trades last for days to months
Speed	Microseconds to milliseconds	Seconds to hours	Hours to days
Holding Period	Seconds to minutes	Minutes to hours	Days, weeks, or months
Data Requirement	Tick-by-tick market data	Real-time intraday data	Daily or weekly price data
Infrastructure	Co-location, low-latency network	Low-latency systems and APIs	Basic retail trading platforms
Strategies Used	Arbitrage, market making	Momentum, mean reversion	Swing trading, trend following
Market Suitability	Volatile, liquid markets	Range-bound, mean-reverting markets	Trending markets
Risk Exposure	Very low (few seconds)	Medium (minutes to hours)	High (overnight and weekend risk)
Operating Cost	Very high (co-location fees)	Medium (cloud-based servers)	Low (can be done on retail platforms)
Who Uses It?	Hedge funds, prop firms	Prop desks, advanced retail traders	Retail traders, hedge funds

Test metrics For Quantitative Strategies

- Backtest - the final step
- Signal Testing
- Avoid multiple rules test
- Start with the most prominent rule
- Reverse signal Exit
- Interdependency of rules
- Understanding grey shades
- Backtest is for validation not research

- Law of Large numbers (No. of Trades)
- Biases
- How the system reacts to price shocks
- Fewer rules and parameters
- Out of Sample Data Testing and Walk Forward Optimization

- Survivorship bias occurs when only the winners are considered while the losers that have disappeared are not considered. Survivorship bias skews the average results upward for the system, causing them to appear to perform better since underperformers have been overlooked. Selection should include dead and inactive Scrip's.
- Look-ahead bias occurs when the strategy has access to data that it would not have been able to see at the time of the historical trade.
- Meta-look-ahead bias comes from the trader's own preexisting knowledge of the dataset.

- Price shocks are large changes in price caused by unpredictable and significant events.
- No one can profit from a price shock by clever planning, but only by luck. A price shock that causes a windfall profit could just as easily have produced a devastating loss.
- By posting profitable results that should have been losses we have underestimated our risk. It is the erroneously low risk that is fatal to trading because it allows overleveraging and undercapitalization. This is the most common cause of catastrophic loss.

- **Out-of-sample (OOS) testing** is the testing the parameters derived through optimization on a period or market in which no optimization has been conducted.

- The comparisons between in-sample and out-of-sample results will differ in performance but should not materially differ in average duration of trades, average winner, average loser, Win Percentage, Profit Factor, etc. These performance expectations must be documented in order to measure comparative results

- Atleast 30 and 50 trades in the OOS data
- Use only the out-of-sample data in your final test, and not include the in-sample data.

- Simplest trading system is buy-and-hold
- Balance between Simple & Complex Systems
- Complex enough to capture enough profit opportunities
- Not so complex that it becomes overfit

- One of the most essential tools in strategy evaluation is the **Equity Plot**, also known as the **Cumulative Returns Plot or Equity Curve**.
- It visually tracks the **total portfolio value** — including both **cash** and **open positions** — throughout the test period.
- The plot helps us quickly assess whether the strategy is **consistently profitable**, **volatile**, or **unstable**.

- **How to Interpret the Equity Plot**
- A **smooth, upward-sloping curve** indicates consistent performance and effective risk control.
- **Sharp drops** reveal drawdowns — highlighting periods of loss or volatility.
- A **sideways or flat curve** may signal a need for better entries, exits, or position sizing.
- **Rising cash levels** can indicate conservative exposure, while **declining cash** may show full deployment of capital.

Key Takeaway

- The **Equity Plot** is the **heartbeat of a trading system** — it tells the story of capital growth, drawdowns, and recovery.
- A successful strategy doesn't just produce profits — it produces a **stable, upward-trending equity curve** that reflects both **profitability and discipline**.
- Ideally, the **equity curve should rise steadily**, confirming that the strategy produces **sustainable growth with controlled drawdowns**.

- Even strong strategies will **experience losses**
- When markets decline, our portfolio will typically decline as well
- A **drawdown plot** shows how far the equity curve falls from its **most recent peak**

- There are **endless ways** to combine rules, filters, and settings in a strategy
- Quantitative analysis allows us to:
 - **Test each idea objectively**
 - Compare alternatives and improvements
 - Build **confidence** before deploying real capital

How We Evaluate Performance

- No single metric tells the full story
- Each metric has **strengths and weaknesses**
- Best practice:
 - Use **multiple measures** to judge robustness and quality

- CAGR / CAR / Compounded Rate of Return
- Annualized Return
- CAR- Risk free Rate
- CAR- Benchmark Return
- CAR - minimum acceptable return (MAR)
- Total Profit / Gross Profit
- Average Profit
- Percentage of Profitable trades
- Length of Average Winning Trade
- Net Profit / Loss
- Jensen's Alpha
- Average net profit per trade

- Standard Deviation
- Maximum Drawdown
- Annualized Volatility
- Downside deviation
- Total Loss / Gross Loss
- Average Loss
- Beta
- Loss Percentage
- Length of Average losing Trade
- Maximum consecutive losses
- Average Annualized Max Drawdown
- Time to recovery from large drawdowns
- Exposure (Time in the market)

Metric	Numerator	Denominator
Sharpe Ratio	CAR- Risk free Rate	Standard Deviation
Treynor Ratio	CAR- Risk free Rate	Beta
Sortino ratio	CAR- Benchmark Return	Downside deviation
Profit Factor	Total Profit	Total Loss
Payoff Ratio	Average Profit	Average Loss
Efficiency Factor	Compounded Rate of Return	Annualized Volatility
Calmar / Mar Ratio	Annualized Return	Maximum Drawdown
Information Ratio	Annualized Return	Standard Deviation

Probabilistic Sharpe Ratio (PSR)

- **What it tells you:** Probability that the observed Sharpe Ratio is statistically greater than a chosen benchmark (usually 0).

How to interpret:

- > 95% → Sharpe is reliable, not luck
- < 80% → performance likely due to randomness
- Use this when sample size is small or returns are noisy.

Sortino / $\sqrt{2}$

- **What it tells you:** Downside-risk-adjusted return, scaled to be comparable with Sharpe.

How to interpret:

- **Higher than Sharpe** → downside risk is well controlled
- Penalizes only **negative volatility**, not upside
- Better than Sharpe for trend-following systems.

Gain / Pain Ratio

- **What it tells you:** Total gains divided by **total drawdowns**.

How to interpret:

- > 1 → profitable
- > 1.5 → good risk control Simple and intuitive
- Used widely by CTAs.

Gain / Pain (1M)

- **What it tells you:** Same as above but **aggregated monthly**.

How to interpret:

- Filters daily noise
- More relevant for **investor-level evaluation**
- Good for PMS or longer-horizon strategies.

Recovery Factor

- **What it tells you:** Net profit divided by **maximum drawdown**.

How to interpret:

- $> 1 \rightarrow$ strategy recovers well
- $> 2 \rightarrow$ strong capital efficiency
- Commonly used by prop desks.

Ulcer Index

- **What it tells you:** Depth and **duration** of drawdowns.

How to interpret:

- Lower is better
- Penalizes long, slow recoveries Better than max drawdown alone.

Tail Ratio

- **What it tells you:** Size of **right tail wins vs left tail losses**.

How to interpret:

- $> 1 \rightarrow$ winners dominate losers
- $< 1 \rightarrow$ dangerous fat-tail losses
- Critical for trend-following and breakout systems.

Common Sense Ratio

- **What it tells you:** Reward quality by combining **tail ratio** × **gain/pain**.

How to interpret:

- > 1 → logical, healthy strategy
- > 2 → strong asymmetric payoff Punishes fat left tails.
- “Does this strategy make sense?”

Outlier Win Ratio

- **What it tells you:** Dependence on **big winning trades**.

How to interpret:

- **High** → classic trend-following behavior
- Too high → few trades drive entire P&L
- Check robustness if these are removed

Omega Ratio

- **What it tells you:** Probability-weighted **gains vs losses** above a chosen threshold (often 0%).

How to interpret:

- > 1.5 → strong payoff asymmetry
- > 2.0 → very favorable distribution
- Captures skew and tail behavior better than Sharpe.

CPC Index (Consistency of Profits)

- **What it tells you:** How **consistent** profits are over time.

How to interpret:

- **High CPC** → smooth equity curve
- **Low CPC** → lumpy returns
- Very important for capital allocation decisions.

Serenity Index

- **What it tells you:** Returns adjusted for **drawdown severity and volatility**.

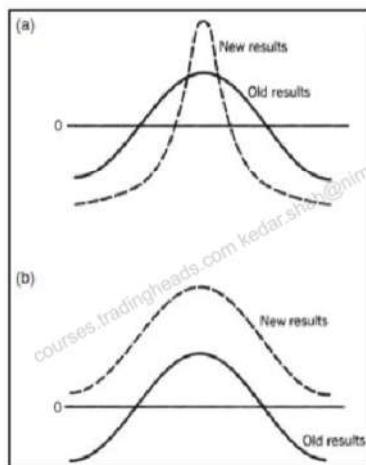
How to interpret:

- **High** → smooth, investor-friendly strategy
- Low → stressful equity curve
- Excellent for PMS suitability checks

- Too much testing.
- “If you torture the data long enough, it will confess to anything”.
Ronald H. Coase
- If you test all combinations of indicators, you will not know whether you have found a good idea or have simply overfit the data.
- Start process with a well-thought-out plan.
- Keep close track of tests performed.
- Definite Objective.
- Solution to the multiple testing fallacy — good experimental design and conscious effort to avoid it.

We do not know ahead of time whether the future will be similar, but we do know that there will be trends and trading ranges. Any system must be able to deal with both of these situations and have developed adjustable parameter sets or rules that will account for them. Also, the market may have high volatility / Low volatility / normal volatility.

1. Uptrend with High Volatility
2. Uptrend with Low Volatility
3. Uptrend with Normal Volatility
4. Downtrend with High Volatility
5. Downtrend with Low Volatility
6. Downtrend with Normal Volatility
7. Trading Range with High Volatility
8. Trading Range with Low Volatility
9. Trading Range with Normal Volatility



- Montecarlo & Stimulations
- No sudden dip in performance measures when parameters are changed slightly
- Percent of Profitable Tests
- Successful across multiple securities
- Bigger Data
- Virtual Data
- Successful in multiple timeframes
- Top ten winners and losers
- Ignoring a few large losses
- Failing to recognize an evolving marketplace
- Test prices for limit moves

- Rule must be based on a sound premise
- Liquidity
- Repainting Signals
- No Leverage in initial testing
- No Hedging in initial testing
- No Pyramiding in initial testing
- Estimation of Slippage and Transaction Costs
- Test for brittleness - the phenomenon when one or more of the rules are never triggered
- Visualizing with Scatter Diagrams
- Paper Trading

- More computer power does make processing faster, but is not essential to finding a successful system.
- Overfitting is useless and even dangerous because it gives us a false sense of confidence.
- Testing that follows a sound procedure and a test evaluation that looks for robust solutions will improve the chances for success. Bad testing can be tragic—it is a waste of time and resources, and leads to a system that has little chance of success.
- Begin simply, increase the complexity slowly, and stop as soon as possible.
- Simple and the most general rules may show lower returns and greater risk than the ideal optimized results, but they are much more likely to represent the future results.

The screenshot shows a Jupyter Notebook interface with the following details:

- Header:** jupyter TTP_Statistics1 (unsaved changes)
- Toolbar:** File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Run, Code, Logout, Trusted, Python 3 (ipykernel)
- Code Cells:**
 - In [1]: Import statements for pandas, numpy, datetime, yfinance, quantstats, matplotlib.pyplot, seaborn, and warnings. Includes `warnings.filterwarnings("ignore")` and `sns.set_style("whitegrid")`.
 - In [2]: Downloads NSEI and TCS.NS data from yfinance between 2015-01-01 and 2020-12-31, printing progress bars for each.
 - In [3]: Renames the 'close' column for both datasets.
 - In [4]: Prints the head of the Nifty and TCS datasets.
 - In [5]: Prints information about the Nifty dataset.
 - In [6]: Prints information about the TCS dataset.

Every Great Trader First Learns Humility

“They [great traders] have all been humbled by the market early on in their careers. Until one has this respect indelibly engraved in their (sic) makeup, the concept of money management and discipline will never be treated seriously.”

- The primary goal of money management is not maximizing returns, but **preventing financial ruin**.
- Ruin is an ever-present possibility. No system is 100% profitable, and losses are a certainty.
- The difference between success and failure is a disciplined framework for managing the risk side of the equation—an area often neglected in favor of finding the perfect entry signal. This framework is your defense against the certainty of losses.



- **Risk in trading is best understood as the combination of the probability of loss and the magnitude of that loss.** Trading systems operate under uncertainty and randomness. Losses are inevitable, even in profitable systems, because markets are probabilistic rather than deterministic.
- Risk management, therefore, does not aim to eliminate losses but to control them so that no single loss or series of losses can irreparably damage trading capital.
- A critical aspect of risk is capital asymmetry. Losses require larger percentage gains to recover. **For example, 20% loss → 25% gain | 50% loss → 100% gain** This nonlinearity makes capital preservation more important than return maximization.
- **Risk is often mistakenly equated with volatility. While volatility measures variability of returns, true trading risk is the possibility of permanent capital loss.**
- A volatile strategy can survive if losses are controlled, while a low-volatility strategy can fail catastrophically if exposed to rare but extreme losses

- **Financial markets reward participants for accepting risk.** Higher expected returns generally require accepting greater uncertainty, larger drawdowns, or longer periods of underperformance.
- Reward can be measured precisely using metrics such as **return on capital or CAGR**, but risk is probabilistic and can never be known with certainty in advance.
- Each trader must **define an acceptable reward-risk balance** based on psychological tolerance, capital size, time horizon, and financial goals.
- Two traders using the same system may experience very different outcomes because of differences in position sizing and risk tolerance. A system that is profitable for one may be untradeable for another due to excessive drawdowns.
- **Importantly, risk and reward are not linearly related. Doubling risk does not guarantee doubling returns**, but it almost always increases drawdowns and emotional stress. Long-term success depends on finding a balance where the trader can remain disciplined across both winning and losing periods.

What is 'risk'? The answer determines your strategy.



The Architect's View Risk as Volatility

The academic perspective defines risk as the statistical variance of returns. It is a predictable, modelable force. The primary goal is to optimize a portfolio's design for the best return at a given level of volatility.



The Practitioner's View Risk as Ruin

The trader's perspective defines risk as the tangible threat of catastrophic loss. It is about survival. The primary goal is to manage positions and capital to avoid drawdowns that permanently impair the ability to invest.

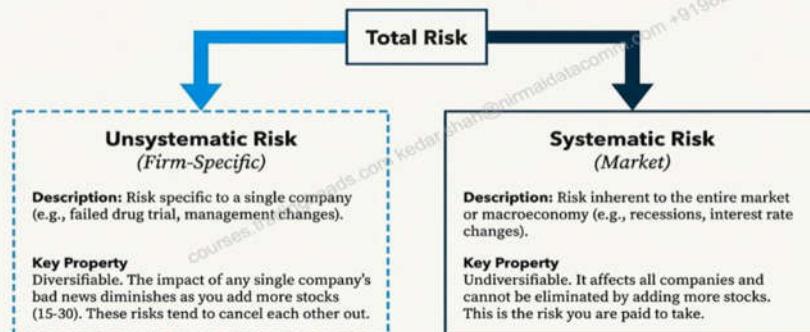
- The terms "risk" and "volatility" are often used interchangeably in finance, but they represent fundamentally different concepts.
- Volatility refers to the *daily fluctuations* in a portfolio's value—the normal, everyday noise of the market. Risk, on the other hand, is about *extreme outcomes*. It's the potential for a loss so significant that it leads to the *inability to stay invested*, forcing you out of the market at the worst possible time.
- A portfolio that fluctuates 1% daily has volatility. A portfolio that drops 50% in a month and forces liquidation has experienced true risk. This brings us to a critical distinction:

"Risk is not how much a portfolio moves. Risk is whether it survives."

- True diversification reduces risk only when the components are **uncorrelated or negatively correlated**.
- Correlation tends to increase during periods of market stress, which is precisely when diversification is needed most. This phenomenon explains why many portfolios fail during crises despite appearing diversified under normal conditions.
- Effective diversification spans markets, strategies, timeframes, and underlying risk drivers. Its objective is not to maximize returns but to reduce drawdowns and smooth equity curves.
- A diversified portfolio keeps capital productive even when individual systems or markets face adverse conditions.

An asset's total risk has two components: one you can eliminate, one you cannot.

Total Risk = Systematic Risk + Unsystematic Risk



During a market crisis, three key transformations occur simultaneously, feeding on each other to create a cascade of losses.

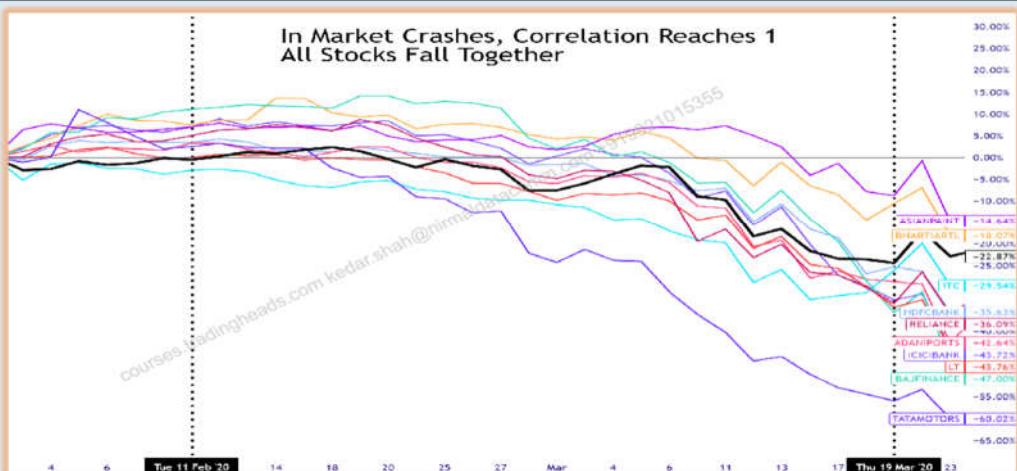
- 1. Volatility Spikes Suddenly** Volatility is not a constant. In a crisis, the measured volatility of the market can increase by multiples—4x, 5x, even 6x—in a matter of days or weeks. The range of potential daily outcomes explodes.
- 2. Correlations Move Toward One** Assets that appear uncorrelated or even negatively correlated during normal times suddenly move in lockstep. The benefits of diversification vanish as investors sell everything indiscriminately in a flight to safety.
- 3. Risk Increases Non-Linearly** The combination of spiking volatility and converging correlations means that risk accelerates. A 20% market drop is not twice as bad as a 10% drop; the damage to a portfolio is often exponentially worse as these dynamics compound.

These are not three separate events; they are a feedback loop. Spiking volatility forces leveraged investors to de-risk, their selling pressure drives correlations toward one, which in turn creates more panic and even greater volatility.

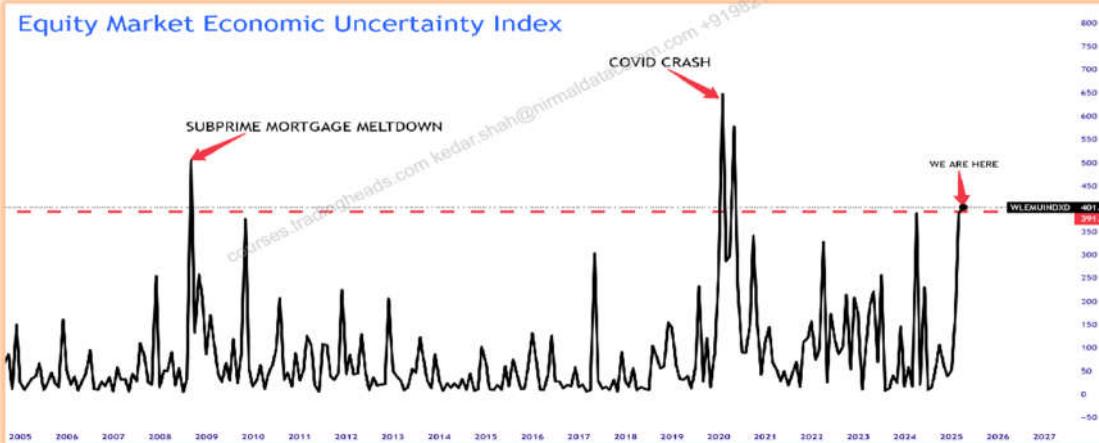
This phenomenon is best summarized by a simple, stark reality:

"In a crisis, diversification temporarily disappears."

In Market Crashes, Correlation Reaches 1
All Stocks Fall Together



Equity Market Economic Uncertainty Index



Lens 1: Modern Portfolio Theory treats risk as the standard deviation of returns.

Pioneer:

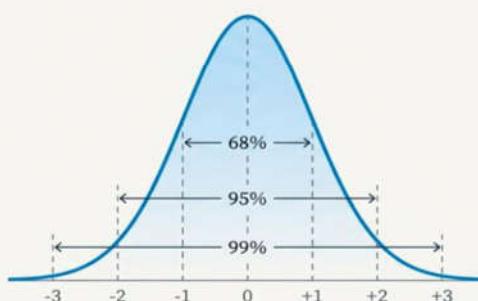
Harry Markowitz (1952). His key insight was to measure risk as the volatility (or standard deviation) of expected returns.

Core Assumption:

Investors are risk-averse, basing decisions on two factors: expected return and risk.

Objective:

The objective is to create a "mean-variance efficient" portfolio—one that offers the highest expected return for a given level of risk, or the lowest risk for a given level of return.



Sharpe and Treynor Ratios evaluate performance by measuring return per unit of risk.

Higher returns are only good if they don't come with excessive risk. These ratios provide an apples-to-apples comparison.

Sharpe Ratio

Am I getting enough return for the *total volatility* I'm taking on?

$$\frac{\text{Portfolio Return} - \text{Risk-Free Rate}}{\text{Standard Deviation}}$$

Best For: Evaluating an entire portfolio or a single, undiversified asset.

Treynor Ratio

Am I getting enough return for the *market risk* I'm taking on?

$$\frac{\text{Portfolio Return} - \text{Risk-Free Rate}}{\text{Beta}}$$

Best For: Evaluating a stock's contribution to a well-diversified portfolio.

Example: Improving Risk-Adjusted Returns

Portfolio	Expected Return	Std. Dev.	Sharpe Ratio
Portfolio 1 (2-Stock)	20.55%	12.88	1.43
Portfolio 2 (3-Stock)	19.70%	9.48	1.85

But is volatility the whole story?

A portfolio optimized for low volatility can still go to zero.
How do we prevent that?

Mean-variance analysis assumes normal distributions, but **real-world returns have "fat tails."** What happens during a crash?

An architect's blueprint is essential, but a building must also withstand unforeseen storms. What is the practitioner's plan for survival?

The focus shifts from optimization to survival.

- The vast majority of our financial models, from portfolio optimization to option pricing, quietly rely on standard deviation as the primary measure of risk.
- In doing so, they make two powerful assumptions: that asset returns are normally distributed—fitting neatly into a bell curve—and that volatility is stable over time.
- To be clear, standard deviation is an elegant and useful metric for describing *typical* market conditions. It effectively captures the expected range of outcomes on an average Tuesday. But its utility evaporates in a crisis.

This leads to the most important warning about this metric:

"Standard Deviation describes typical days, not crisis days."

Standard deviation is an inadequate measure of true risk for three fundamental reasons, especially during periods of extreme stress:

- **It is backward-looking:** SD is calculated using past data. A model based on the calm period of 2005-2006 was utterly unprepared for the dynamics of 2008. The past is often a poor guide for an unprecedented future.
- **Crashes are regime changes:** A crisis is not just an event at the far end of the bell curve. It represents a different state of the world, where the rules of the game have changed.
- **Extreme moves dominate outcomes:** Portfolio returns are not determined by the average of thousands of small daily moves. They are dominated by the handful of extreme days. A single -20% loss can erase years of +0.1% daily gains, rendering the "average" irrelevant.

This leads to a more sophisticated understanding of risk. It's not just about the final distribution of returns, but the path you take to get there. **"Risk is path-dependent, not just distribution-dependent."**

This means the sequence of returns matters immensely. A portfolio can withstand a series of small losses, but a single, sharp 50% drop at the start can trigger margin calls and force liquidation, an outcome the final distribution statistics alone would never reveal.

- To visualize this, imagine a chart of daily market returns during a calm period. You would see a series of small, manageable moves clustered around zero.
- Now, picture a crisis. That same chart would be characterized by huge, discontinuous gaps—days where the market opens dramatically lower and plunges further.
- This is the volatility explosion, and it reveals a critical flaw in our models. It's not just that the market is moving more; it means the yardstick we use to measure risk has just snapped in half.
- The **standard deviation itself becomes unstable**, rendering it an unreliable guide precisely when we need it most.

- Drawdown measures the **decline from an equity peak to a subsequent trough** and represents the true cost of risk.
- Maximum drawdown defines the worst historical loss** and has a direct impact on psychological tolerance and capital requirements. Under-capitalization is one of the most common causes of trading failure. **Initial Capital should be at least 2-3 times MDD**
- Traders often allocate capital based on average performance rather than worst-case scenarios. Proper capital planning must assume drawdowns worse than those observed in historical backtests.
- Adequate capital allows systems to endure adverse periods without forced liquidation or emotional abandonment.

Lens 2: Practitioners define risk as the potential for drawdown and ruin.

For active managers, the most important risk isn't abstract volatility but the real possibility of losing significant capital. This is about staying in the game.

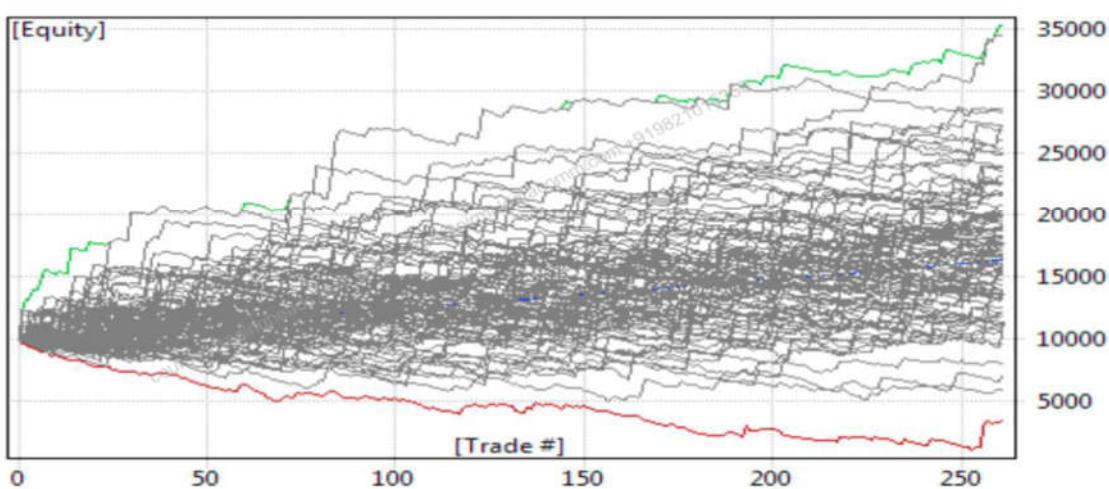
- Drawdown:** The percentage decline in capital from a peak to a trough.
- Maximum Drawdown (MDD):** The worst peak-to-trough decline. Often used as an estimate of the worst-case future loss.
- Risk of Ruin (ROR):** The probability that a strategy will lose so much capital it can no longer continue.

"Ruin is also very likely. Every day some traders and investors are wiped out, largely because they did not utilize a portfolio method that included an assessment and control of risk."



- Risk of ruin represents the probability that a trader will lose a substantial portion or all of trading capital.
- It is a function of three variables: the probability of winning trades, the payoff ratio between wins and losses, and the percentage of capital risked per trade.
- Even systems with positive expectancy can have an unacceptably high risk of ruin if position sizing is aggressive. This is why many traders with seemingly profitable strategies eventually fail.
- **Risk of ruin highlights the fact that survival is a mathematical requirement, not a matter of confidence or conviction.** Professional traders design systems with the explicit goal of minimizing the probability of ruin.

- **Monte Carlo simulation is a critical tool for realistic risk assessment.** By randomizing the sequence of trades, it reveals the full distribution of possible outcomes, including deeper drawdowns and longer losing streaks than those seen in a single backtest. Instead of one equity curve, you get a distribution of possible curves. This lets you see ranges for metrics like drawdown, CAGR, Sharpe ratio, or risk of ruin.
- It takes the list of trades you got from your backtest and shuffles or resamples them thousands of times. This simulates different possible paths of wins and losses, rather than just the one historical order.
- **Monte Carlo analysis exposes risks that traditional backtesting hides.** It allows traders to estimate capital requirements, drawdown probabilities, and **risk of ruin** under adverse conditions. Serious system development requires Monte Carlo testing to ensure that strategies are robust, not just historically profitable.



Percentile	Final Equity	Annual Return	Max. Drawdown \$	Max. Drawdown %	Lowest Eq.
1%	5706	-7.37%	-7685	-63.82%	3618
5%	7987	-3.02%	-5292	-45.47%	5853
10%	9706	-0.41%	-4626	-38.48%	6690
25%	12851	3.48%	-3563	-27.63%	8107
50%	16174	6.78%	-2747	-19.77%	9135
75%	19632	9.64%	-2136	-14.38%	9640
90%	23258	12.21%	-1726	-11.32%	9922
95%	25269	13.48%	-1549	-9.76%	10000
99%	29139	15.71%	-1302	-7.23%	10000

A 99% percentile value of -7.23% means that in 99% of cases, you will see drawdowns worse (more negative) than -7.23%. A 1% percentile value of -63.82% tells you that in 1% of cases, you would experience drawdowns equal to or worse (more negative) than -63.82%.

- Losses in trading do not occur in a neat alternating pattern. Instead, they cluster into sequences known as runs.
- The theory of runs explains that even with a high probability of success, long losing streaks are statistically inevitable.** Traders consistently underestimate the length and frequency of losing runs.
- This underestimation leads to overconfidence during winning periods and panic during drawdowns. **Many systems are abandoned at precisely the worst possible time, just before recovery occurs.**
- Understanding runs prepares traders psychologically and mathematically for extended periods of losses. **A system must be designed not only to be profitable but to be survivable during its worst historical and hypothetical runs.**

A series of probable small losses can be more dangerous than a single large one.

Theory of Runs

This theory quantifies the probability of consecutive losses. A system with a 60% win rate still has a 1.6% chance of five losses in a row. A long string of losses, even if small, can lead to premature abandonment of a profitable system.



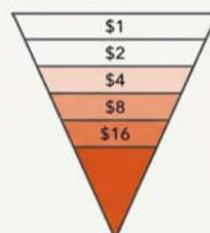
$$0.40 \times 0.40 \times 0.40 \times 0.40 \times 0.40 = 0.40^5 = 1.6\%$$

The Martingale System: A Cautionary Tale

How it works: Double your bet after every loss. A single win recovers all previous losses plus the original profit.

The Appeal: In theory, it has a high probability of a small gain.

The Flaw: It requires infinite capital to withstand an unexpectedly long series of losses. It ignores the risk of ruin, which approaches 100% as trials increase, because you will eventually hit table limits or run out of capital.



A strategy's average return is irrelevant if a string of bad luck can wipe you out first.

- **Position sizing determines how much capital is exposed on each trade and is the single most important component of risk management.**
- It controls drawdowns, volatility, and long-term survivability more than entries, indicators, or market selection. A strong trading edge combined with poor position sizing will fail, while a modest edge with disciplined risk control can compound successfully over time.
- Position sizing translates abstract strategy logic into real financial outcomes. **Effective position sizing ensures that no single trade or sequence of trades can materially damage the trading account.** It allows traders to remain emotionally stable and operationally consistent.

- Several position sizing models are commonly used in practice.
- **Fixed dollar risk limits the absolute loss per trade, while fixed percentage risk adjusts exposure dynamically as account equity changes.**
- **Volatility-based sizing** accounts for market conditions by scaling positions according to price variability.
- **Kelly and Optimal-f** methods aim to maximize long-term growth but are highly aggressive and lead to large drawdowns.
- As a result, most professional traders deliberately use a fraction of the Kelly size. The goal is not theoretical optimality but practical survivability and consistency.

- **Leverage magnifies both gains and losses and should be used cautiously. While it can enhance returns, it also increases drawdowns and reduces margin for error.**
- Leverage must be aligned with risk tolerance and capital strength. **Protective stops are essential for limiting losses and preventing catastrophic outcomes.**
- Stops define risk before entry and enforce discipline during adverse market movements.
- Psychological risk often exceeds market risk. Fear, greed, and overconfidence cause traders to deviate from systems, increase size after losses, or abandon strategies prematurely. Long-term success depends on discipline, consistency, and respect for risk

Money management is not about prediction; it's about controlling losses when you're wrong.

A protective stop is a pre-determined price – to exit a position and prevent further losses. It is an inviolable rule for risk control. It answers the question: "At what point is my initial thesis proven wrong?" Source Serif Pro

Protective Stop



A hard price level to limit the maximum loss on any single trade.

Trailing Stop



A stop that adjusts upwards as a trade becomes profitable, locking in gains while allowing room to grow. Can be based on price, volatility (ATR), or technical levels.

Managing Leverage



Leverage magnifies both gains and losses. It dramatically increases the risk of ruin and must be used with extreme caution, tied directly to the system's expected MDD.

Entry strategies carry no risk until executed. Once a position is on, an exit strategy is the most important decision you will make.



Pillar 2: The Art of the Exit—Managing Live Positions

Once you enter a position, your capital is at risk. An exit strategy is the most important action in any system. All entries must have a protective stop under all circumstances. This is your ultimate line of defense.

The Role of the Stop

- A stop represents the **maximum loss** the trader can afford to take on any single position.
- It is a pre-determined point that reflects that the potential for profit is no longer greater than the potential for further loss.
- It is an **inviolate rule**, removing emotion and panicked decision-making from a losing trade.

Two Families of Stops

- **Protective Stops:** A fixed level determined at entry to prevent catastrophic loss.
- **Trailing Stops:** A dynamic level that moves in your favor to lock in gains and protect profits.



The Protocol for "If Everything Goes Wrong"

Occasionally, a portfolio model breaks down completely. Murphy's Law takes hold, and everything that can go wrong does. At that point, the remedy is to close the system down and exit all positions. This is not a tactical adjustment; it is a strategic retreat to preserve capital.

The Ultimate Failsafe

- A pre-defined standard for closing the entire portfolio model must be established before you begin trading.
- This is your line in the sand—an objective rule that forces you to stop the bleeding.
- **Example:** A 20% drawdown on total portfolio equity from its peak.
- **Action:** If the portfolio stop is hit, all positions are liquidated without question. The system is then taken offline for re-evaluation and testing. No new trades are placed until the issues are identified and resolved.





Case Study 1: The 2008 Global Financial Crisis



The 2008 crisis remains the quintessential example of modern risk model failure. Over a period of weeks, equity market **volatility increased 4–6x**. Assets that were assumed to be uncorrelated, such as real estate, credit, and equities, all fell together in a correlated panic. For firms that employed it, **leverage amplified these losses** into catastrophic, firm-ending events.

The key takeaway from 2008 is a nuanced one. The sophisticated quantitative models used by banks and hedge funds didn't necessarily "break" in the sense of a calculation error. The problem was deeper.

"Models didn't break. Assumptions broke."



Case Study 2: The March 2020 COVID Crash



The 2020 crash was a different kind of crisis, notable for its breathtaking velocity. Global **markets fell ~30-40% in a matter of weeks**, not months. We witnessed **daily moves that were many multiples of the historical standard deviation**, making a mockery of any model based on normal distributions. In the initial panic, **even traditionally defensive assets like government bonds were sold off** as investors scrambled for cash.

The crucial insight from this event was that risk has multiple dimensions. It isn't just about how far the market falls, but how fast.

"Risk came from speed, not just magnitude."



Case Study 3: Long-Term Capital Management (1998)



The collapse of LTCM is the classic cautionary tale in quantitative finance. Here was a firm run by celebrated traders and guided by the insights of **Nobel Prize-winning models**. Their strategy was built on **high leverage** applied to trades that sought to exploit small, predictable pricing anomalies. The entire edifice was built on the **assumption of normal distributions** in market returns. When a Russian debt default triggered a global flight to quality, volatility and correlations moved in ways their models deemed impossible, and the fund imploded.

The lesson from LTCM is one that every manager of a seemingly "safe" strategy must remember:

"Low volatility strategies can hide catastrophic risk."

These three events, separated by years and driven by different catalysts, are a recurring lesson in the failure of our statistical imagination. In each case, the models worked perfectly, but they were modelling a world that no longer existed.

If standard deviation is not the answer, then what is the goal? **The true purpose of risk management is not to optimize for the 95% of normal days. It is to prepare for the 5% of days that threaten survival.** Its mandate can be defined by four clear objectives:

- 1. Survive extreme events.** This is the prime directive. All other goals are secondary.
- 2. Control drawdowns.** Keep losses within a tolerable threshold so that capital is preserved to invest another day.
- 3. Manage exposure during stress.** Have a plan to reduce risk *during* a crisis, not after the damage is done.
- 4. Stay in the game.** The ultimate goal is to avoid being forced out of the market, ensuring you can participate in the eventual recovery.

"Risk management exists for the days you didn't model."

- **Exposure control:** This counters the risk of a changing environment by actively managing how much capital is deployed, ensuring we aren't fully invested right before a volatility explosion.
- **Trend filters:** These serve as a defence against our own assumptions by **systematically reducing exposure when a market is in a clear downtrend**, preventing us from trying to 'catch a falling knife' based on a model that assumes a reversion to the mean.
- **Position sizing:** This directly addresses the danger of a single catastrophic event by ensuring no single position can ever be large enough to threaten the survival of the entire portfolio.
- **Drawdown limits:** These are pre-defined circuit breakers that counter our emotional biases during a panic, forcing a disciplined reduction in risk as losses accumulate, preserving capital for the eventual recovery.
- **Stress testing, not just backtesting:** This is an act of intellectual humility that forces us to confront model weakness by simulating performance under known historical or hypothetical crises, rather than just the comfortable past data used in a backtest.

A unified framework requires both the Architect and the Practitioner.

The Architect (MPT) Provides the Strategic Blueprint

- Focuses on: Asset Allocation, Portfolio Construction, Diversification.
- Asks: "How do I build a portfolio with the best long-term risk/return characteristics?"
- Tools: Mean-Variance Optimization, Correlation Analysis, Beta.



The Practitioner (Money Management) Provides the Tactical Rulebook

- Focuses on: Execution, Survival, Loss Control.
- Asks: "How do I manage capital and individual positions to avoid being forced out of the market?"
- Tools: Position Sizing, Stop-Loss Orders, Leverage Control.

A robust strategy needs both. MPT builds an efficient ship. Money Management ensures you are a skilled captain who can navigate it through storms without sinking.

Managing risk is a dual mandate: Seek efficiency, but demand survival.



Risk has two faces:
The **statistical volatility** you can model (MPT) and the **catastrophic ruin** you must avoid (Money Management).



Diversification is powerful but incomplete: It protects from firm-specific risk, but not from market risk or your own poor execution.



Your first decision is not *what* to buy, but *how much*. Position sizing and loss control are more critical to long-term success than asset selection.

A successful investor is both an **architect**, who designs a robust portfolio for the long term, and a **practitioner**, who navigates the present with disciplined execution to ensure they survive to see it.

Mastering Investment Risk: From Theory to Practice



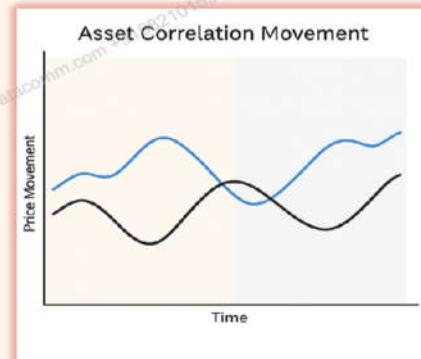
“WHAT IS CORRELATION”

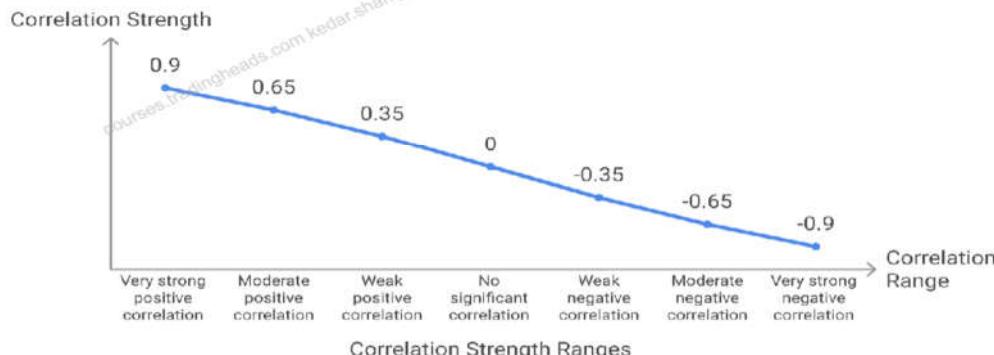
Correlation measures the **strength and direction** of the relationship between two assets.

It ranges from **-1 to +1**:

- **+1** → Move exactly together
- **0** → No relationship
- **-1** → Move exactly opposite

In investing, correlation helps understand how different assets move together — important for diversification and risk management.





Every Investment Decision Is a Trade-Off Between Risk and Return

To make intelligent decisions, we must first quantify these two dimensions. We use historical data to establish a baseline for what we can expect.

- Expected Return:** The average annual return of an asset, our best estimate for future performance.
- Standard Deviation (Volatility):** The dispersion of returns around the average. This is the most common measure of an asset's "total risk." Higher volatility means a wider range of potential outcomes.

Ticker	Expected Return	Std. Dev. (Total Risk)
CVX	17.87%	16.55%
YUM	23.24%	20.07%
JNJ	13.59%	15.43%
^SPX	17.81%	15.45%

Total Risk Can Be Decomposed into Two Distinct Components

A stock's total volatility is driven by two different types of factors. Understanding this distinction is the first step toward building a more resilient portfolio.



$$\text{Total Risk} = \text{Market Risk} + \text{Firm-Specific Risk}$$

$$\text{Total Risk} = \text{Systematic Risk} + \text{Unsystematic Risk}$$

$$\text{Total Risk} = \text{Undiversifiable Risk} + \text{Diversifiable Risk}$$

The crucial insight from Modern Portfolio Theory is that investors should **not expect to be compensated** for taking on **Firm-Specific risk**, precisely because it can be eliminated for free.

Diversification Is the Key to Eliminating Firm-Specific Risk

The power of diversification comes from combining assets that do not move in perfect unison. The statistical measure for this relationship is **correlation**.

- **Correlation (ρ)** ranges from +1.0 (perfectly synchronized) to -1.0 (perfectly opposite).
- Combining assets with low, or even negative, correlations is the most effective way to reduce portfolio volatility.

	CVX	YUM	JNJ	^SPX
CVX	1.000	-0.020	0.091	0.496
YUM	-0.020	1.000	-0.739	0.543
JNJ	0.091	-0.739	1.000	0.097
^SPX	0.496	0.543	0.097	1.000

Very low correlation. These two stocks have almost no relationship, making them excellent diversifiers for each other.

Moderate correlation. YUM tends to move with the market, but not perfectly.

A Diversified Portfolio Can Offer Higher Returns for Lower Risk

By combining two assets with low correlation (CVX and YUM, $\rho = -0.02$), we can create a portfolio that is more "mean-variance efficient" than either stock alone. This combination offers a better reward per unit of risk.

To quantify this, we use the **Sharpe Ratio**, which measures excess return (above the risk-free rate) per unit of total risk (standard deviation). A higher Sharpe Ratio is better.



The portfolio's Sharpe Ratio of 1.43 demonstrates it is significantly more efficient at generating returns for the risk taken.

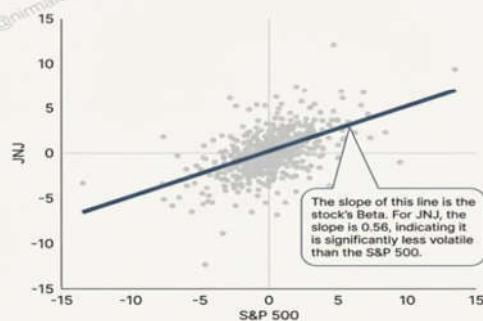
Beta Measures the Systematic Risk That Diversification Cannot Eliminate

Once a portfolio is well-diversified, its primary risk exposure is to broad market movements. Beta quantifies this sensitivity.

Definition: Beta measures a stock's or portfolio's volatility in relation to the overall market (e.g., the S&P 500).

Interpretation:

- **Beta = 1.0:** Moves in line with the market.
- **Beta > 1.0:** More volatile than the market ("high-beta").
- **Beta < 1.0:** Less volatile than the market ("low-beta").



- Owning a single stock exposes the portfolio to the highest level of risk
- Adding multiple stocks from the same sector can reduce risk to some extent, since their daily returns are not perfectly identical, even though they remain correlated.
- Adding stocks from different sectors reduces risk more effectively, as these stocks are influenced by different economic and business drivers.
- The most practical way to evaluate diversification is by analyzing the correlation of daily percentage returns between stocks.
- By selecting stocks that exhibit low or moderate correlation, we can construct a better-diversified portfolio.
- Even when returns do not improve, a meaningful reduction in risk alone is a significant achievement.
- According to **Modern Portfolio Theory (MPT)**, the benefit of diversification diminishes beyond roughly 20-30 stocks, as most diversifiable (idiosyncratic) risk has already been reduced.

Modern Portfolio Theory (MPT)

Modern Portfolio Theory (MPT) concluded that **adding more stocks to a portfolio reduces overall risk through diversification**, because the returns of different assets are not perfectly correlated. This allows investors to achieve the same expected return with lower volatility compared to holding individual stocks.

Core Insight of MPT

- Diversification lowers risk:** By combining assets with different risk-return profiles, investors can reduce the portfolio's overall variance without necessarily sacrificing returns.
- Risk vs. return trade-off:** MPT formalized the idea that investors should seek the *efficient frontier*—portfolios that maximize expected return for a given level of risk.
- Correlation matters:** The benefit of adding stocks depends on how correlated they are. If assets move differently under market conditions, diversification is more effective.

How Adding Stocks Reduces Risk

Aspect	Holding a Single Stock	Diversified Portfolio
Risk Exposure	Entirely tied to one company's performance	Spread across multiple companies
Volatility	High, due to company-specific events	Lower, as shocks in one stock may be offset by others
Expected Return	Depends solely on one firm	Balanced across industries/sectors
Systematic Risk	Still present (market-wide)	Cannot be eliminated, but unsystematic risk is reduced

Diversification : Elbow Curve

Important Caveats

- Diminishing returns to diversification:** After a certain number of stocks (often cited as 20–30), the marginal risk reduction becomes small because most unsystematic risk has already been diversified away.
- Systematic risk remains:** Diversification cannot eliminate risks tied to the entire market (e.g., recessions, interest rate changes).
- Quality matters:** Simply adding more stocks isn't enough—choosing assets with low correlation is key.



The **efficient frontier** in Modern Portfolio Theory represents the set of portfolios that deliver the **highest expected return for a given level of risk** or the **lowest risk for a given level of return**. Adding stocks and diversifying moves a portfolio closer to this frontier, where investors achieve optimal risk-return trade-offs.

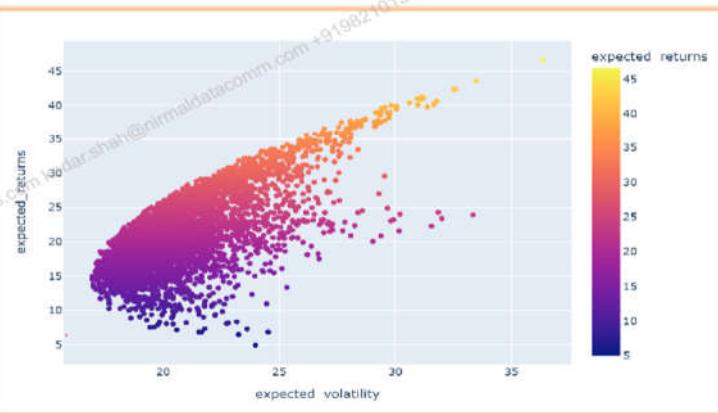
What the Efficient Frontier Shows

- **Optimal portfolios only:** Any portfolio on the frontier is considered efficient because no other portfolio offers a better return for the same risk.
- **Risk-return balance:** It visualizes the trade-off between risk (measured by volatility) and expected return.
- **Diversification impact:** By combining assets with different correlations, investors can push their portfolio toward the frontier, reducing risk without lowering returns.

Comparison: Inside vs. On the Frontier

Portfolio Type	Position	Characteristics	Investor Outcome
Inefficient Portfolio	Below the frontier	Higher risk for lower return	Suboptimal choice
Efficient Portfolio	On the frontier	Best possible return for given risk	Optimal allocation
Single Asset	Far below frontier	Exposed to unsystematic risk	Poor diversification

The efficient frontier is the **visual core of MPT**, guiding investors to construct portfolios that maximize efficiency. It shows that smart diversification isn't just about adding more assets—it's about combining them in the right proportions to reach the frontier.



TTP_Statistics1

```
In [1]: import pandas as pd
import numpy as np
import datetime
import yfinance as yf
import quantstats as qs
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")
sns.set_style("whitegrid")
```

```
In [2]: nifty = yf.download("^NSEI", start="2015-01-01", end="2020-12-31")['Close']
tcs = yf.download("TCS.NS", start="2015-01-01", end="2020-12-31")['Close']

[*****100%*****] 1 of 1 completed
[*****100%*****] 1 of 1 completed
```

```
In [3]: # Rename columns
nifty.columns = ['close']
tcs.columns = ['close']
```

```
In [4]: print(nifty.head())
print(tcs.head())

      close
Date
2015-01-02  8395.450195
2015-01-05  8378.400391
2015-01-06  8127.350098
2015-01-07  8102.100098
2015-01-08  8234.599609
      close
Date
2015-01-01  1006.856140
2015-01-02  1020.265137
2015-01-05  1004.759827
2015-01-06   967.718384
2015-01-07   956.286743
```

```
In [5]: nifty.info()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1471 entries, 2015-01-02 to 2020-12-30
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   close    1471 non-null   float64 
dtypes: float64(1)
memory usage: 23.0 KB
```

```
In [6]: tcs.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1479 entries, 2015-01-01 to 2020-12-30
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   close    1479 non-null   float64 
dtypes: float64(1)
memory usage: 23.1 KB
```

```
In [7]: nifty['returns'] = nifty['close'].pct_change()
tcs['returns'] = tcs['close'].pct_change()
```

```
In [8]: nifty = nifty.dropna()
tcs = tcs.dropna()
```

Daily Return Time Series (Nifty vs TCS)

Shows how daily returns evolve over time and helps compare volatility.

```
In [9]: # ---- Date Range ----
start = "2020-01-01"
end   = "2020-12-31"

nifty_plot = nifty.loc[start:end]
tcs_plot   = tcs.loc[start:end]

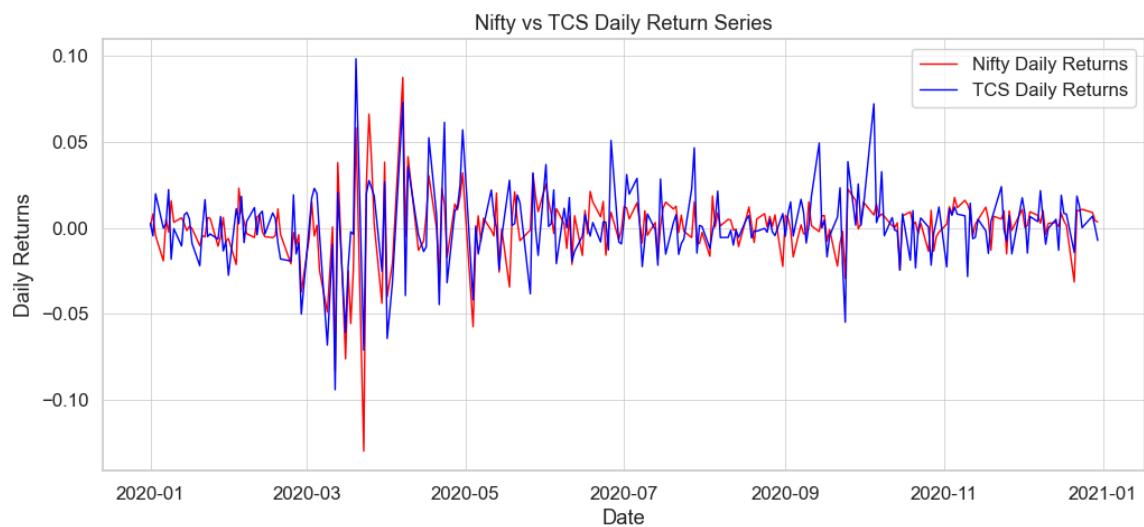
# ---- Plot ----
plt.figure(figsize=(12,5))

plt.plot(nifty_plot.index, nifty_plot['returns'],
          label="Nifty Daily Returns", linewidth=1, color="red") # Dark Blue

plt.plot(tcs_plot.index, tcs_plot['returns'],
          label="TCS Daily Returns", linewidth=1, color="blue")    # Dark Red

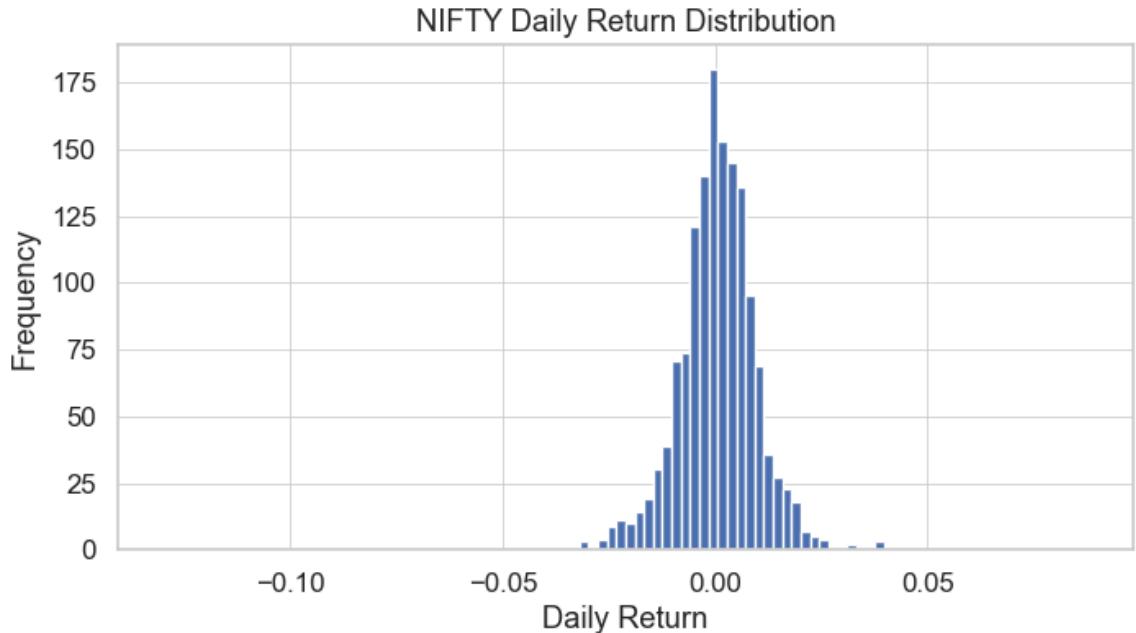
plt.title("Nifty vs TCS Daily Return Series")
plt.xlabel("Date")
plt.ylabel("Daily Returns")
plt.grid(True)
plt.legend()

plt.show()
```

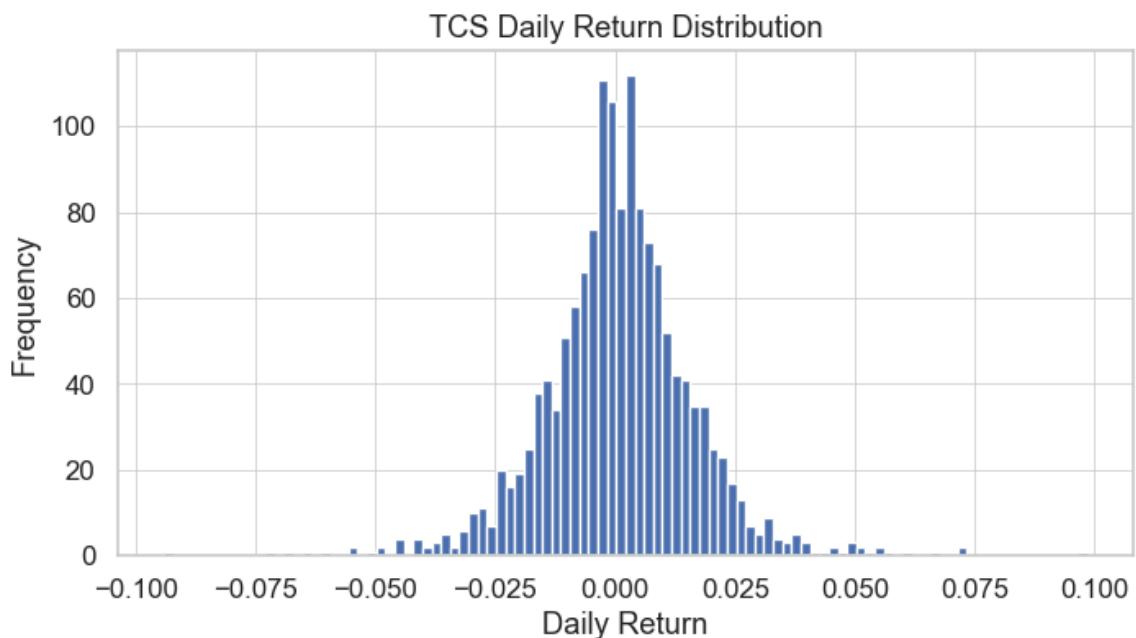


Histogram of daily returns

```
In [10]: nifty['returns'].hist(bins=100, figsize=(8,4))
plt.title("NIFTY Daily Return Distribution")
plt.xlabel("Daily Return")
plt.ylabel("Frequency")
plt.show()
```



```
In [11]: tcs['returns'].hist(bins=100, figsize=(8,4))
plt.title("TCS Daily Return Distribution")
plt.xlabel("Daily Return")
plt.ylabel("Frequency")
plt.show()
```



Descriptive Statistics: Nifty and TCS Daily Returns

```
In [12]: import numpy as np
from scipy import stats

# Annualization factor
trading_days = 252

# =====
# ----- NIFTY -----
# =====

n_mean    = nifty['returns'].mean()
n_median  = nifty['returns'].median()
n_mode    = stats.mode(nifty['returns'], keepdims=True)[0][0]
n_sd_d    = nifty['returns'].std()
n_sd_y    = n_sd_d * np.sqrt(trading_days)
n_var     = nifty['returns'].var()

print("---- NIFTY STATISTICS ----")
print(f"Mean (daily)      : {n_mean:.6f}")
print(f"Median (daily)    : {n_median:.6f}")
print(f"Mode (daily)      : {n_mode:.6f}")
print(f"Std Dev (daily)   : {n_sd_d:.6f}")
print(f"Std Dev (yearly)  : {n_sd_y:.6f}")
print(f"Variance (daily)  : {n_var:.6f}")
print()

# =====
# ----- TCS -----
# =====

t_mean    = tcs['returns'].mean()
t_median  = tcs['returns'].median()
t_mode    = stats.mode(tcs['returns'], keepdims=True)[0][0]
t_sd_d    = tcs['returns'].std()
t_sd_y    = t_sd_d * np.sqrt(trading_days)
t_var     = tcs['returns'].var()

print("---- TCS STATISTICS ----")
print(f"Mean (daily)      : {t_mean:.6f}")
print(f"Median (daily)    : {t_median:.6f}")
print(f"Mode (daily)      : {t_mode:.6f}")
print(f"Std Dev (daily)   : {t_sd_d:.6f}")
print(f"Std Dev (yearly)  : {t_sd_y:.6f}")
print(f"Variance (daily)  : {t_var:.6f}")
```

----- NIFTY STATISTICS -----
Mean (daily) : 0.000412
Median (daily) : 0.000624
Mode (daily) : 0.000000
Std Dev (daily) : 0.011302
Std Dev (yearly) : 0.179421
Variance (daily) : 0.000128

----- TCS STATISTICS -----
Mean (daily) : 0.000766
Median (daily) : 0.000510
Mode (daily) : 0.000000
Std Dev (daily) : 0.015798
Std Dev (yearly) : 0.250787
Variance (daily) : 0.000250

Scatter Plot of Nifty and TCS

```
In [13]: # Combine both return series on common dates
df = pd.concat([nifty['returns'], tcs['returns']], axis=1, join='inner')

df.columns = ['Nifty', 'Tcs']

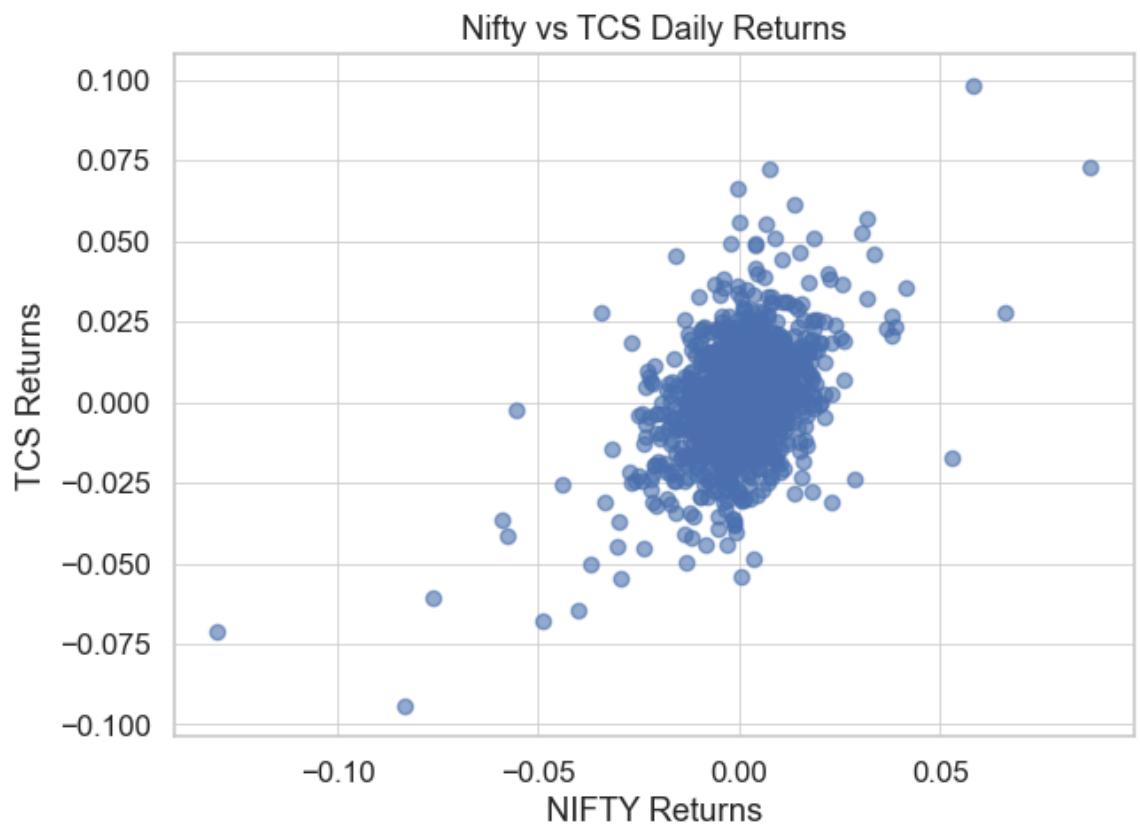
# Drop missing values
df = df.dropna()

# Ensure numeric
df = df.astype(float)

# Clean Scatter Plot
plt.figure(figsize=(7,5))
plt.scatter(df['Nifty'], df['Tcs'], alpha=0.6, marker='o')

plt.title("Nifty vs TCS Daily Returns")
plt.xlabel("NIFTY Returns")
plt.ylabel("TCS Returns")

plt.grid(True)
plt.show()
```



OLS(ordinary least squares) Regression Using NumPy

```
In [14]: # taking Nifty returns as x-axis data, TCS returns as y-axis data
#.values converts pandas Series --> numpy arrays
x = df['Nifty'].values
y = df['Tcs'].values

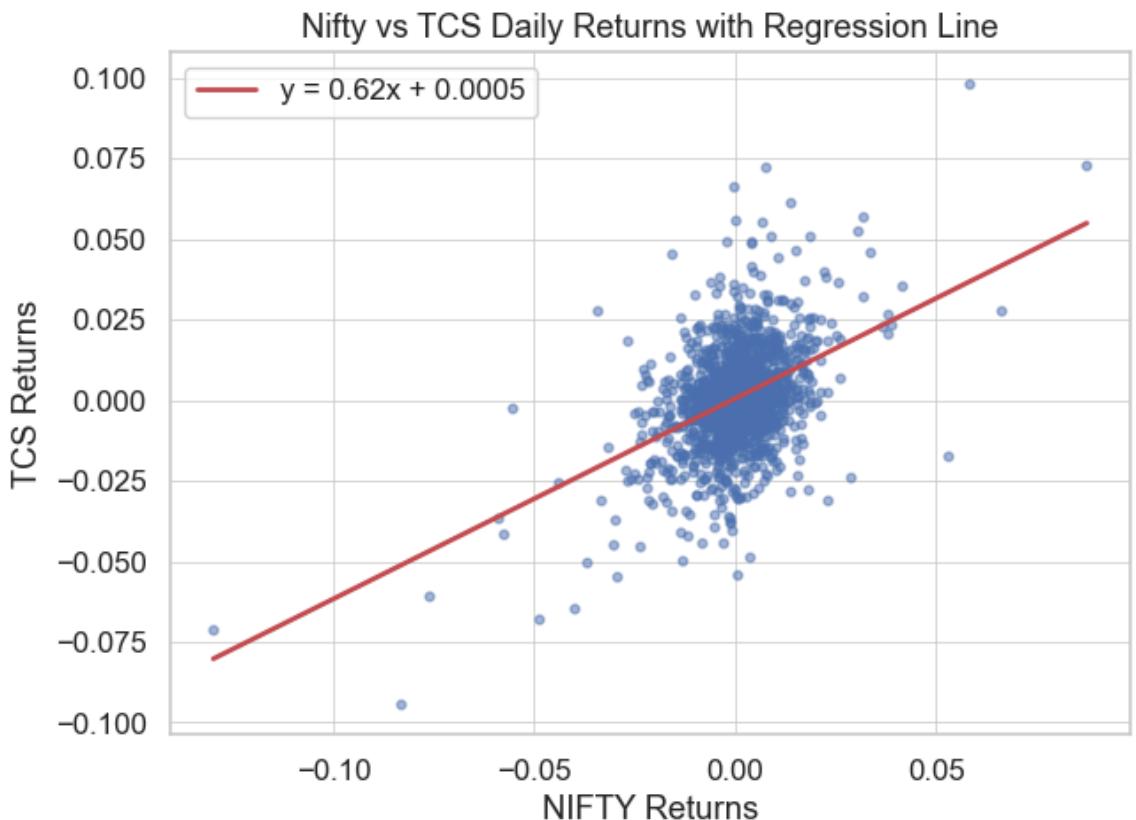
# Linear regression: y = m*x + c
#1 means we want a first-degree (linear) polynomial
#returns m --> slope and c --> intercept
m, c = np.polyfit(x, y, 1)

# Generates 100 evenly spaced x and y values to give a smooth line
x_line = np.linspace(x.min(), x.max(), 100)
y_line = m * x_line + c

# Plot
plt.figure(figsize=(7,5))

#Plots the regression line in red('r'), shows slope and intercept in the leg
plt.scatter(x, y, alpha=0.5, s=12)
plt.plot(x_line, y_line, 'r', linewidth=2, label=f"y = {m:.2f}x + {c:.4f}")

plt.title("Nifty vs TCS Daily Returns with Regression Line")
plt.xlabel("NIFTY Returns")
plt.ylabel("TCS Returns")
plt.grid(True)
plt.legend()
plt.show()
```



TCS shows a positive but moderate relationship with Nifty, where the slope of 0.62 means TCS moves in the same direction as Nifty but with only about 62 percent of the market's intensity. The tiny intercept value 0.0005 indicates almost no independent drift when Nifty's return is zero. Overall, this equation implies TCS behaves like a low-beta stock, rising and falling less sharply than Nifty.

Outliers Detection (Nifty)

```
In [15]: #Flag top 1 percent and bottom 1 percent returns
lower_n = df['Nifty'].quantile(0.01)
upper_n = df['Nifty'].quantile(0.99)

outliers_nifty = df[(df['Nifty'] < lower_n) | (df['Nifty'] > upper_n)]
print(outliers_nifty)
```

Date	Nifty	Tcs
2015-01-06	-0.029964	-0.036866
2015-01-15	0.026167	0.006801
2015-08-24	-0.059151	-0.036433
2016-02-11	-0.033171	-0.031228
2016-03-01	0.033669	0.045870
2019-05-20	0.036915	0.023145
2019-09-20	0.053191	-0.017365
2019-09-23	0.028916	-0.024038
2020-02-28	-0.037096	-0.050036
2020-03-09	-0.048956	-0.068086
2020-03-12	-0.083019	-0.094104
2020-03-13	0.038065	0.020595
2020-03-16	-0.076121	-0.060843
2020-03-18	-0.055565	-0.002171
2020-03-20	0.058329	0.098451
2020-03-23	-0.129805	-0.071073
2020-03-25	0.066247	0.027684
2020-03-26	0.038904	0.023225
2020-03-30	-0.043781	-0.025212
2020-03-31	0.038238	0.026764
2020-04-01	-0.040005	-0.064263
2020-04-07	0.087632	0.073147
2020-04-09	0.041509	0.035592
2020-04-17	0.030463	0.052533
2020-04-21	-0.030275	-0.044539
2020-04-30	0.032088	0.057093
2020-05-04	-0.057445	-0.041699
2020-05-18	-0.034323	0.027841
2020-05-27	0.031665	0.032064
2020-12-21	-0.031405	-0.014278

Outliers Detection (TCS)

```
In [16]: #Flag top 1 percent and bottom 1 percent returns
lower_t = df['Tcs'].quantile(0.01)
upper_t = df['Tcs'].quantile(0.99)

outliers_tcs = df[(df['Tcs'] < lower_t) | (df['Tcs'] > upper_t)]
print(outliers_tcs)
```

	Nifty	Tcs
Date		
2015-04-17	-0.011566	-0.041796
2015-10-14	-0.002927	-0.044198
2016-09-08	0.003874	-0.048534
2016-11-09	-0.013057	-0.049645
2016-11-25	0.018681	0.051132
2017-01-31	-0.008277	-0.044193
2018-01-22	0.006563	0.055584
2018-03-13	0.000523	-0.054175
2018-04-20	-0.000118	0.066214
2018-07-11	0.000096	0.055890
2018-10-04	-0.023853	-0.045334
2018-11-28	0.004048	0.049544
2019-04-15	0.004028	0.049045
2020-02-28	-0.037096	-0.050036
2020-03-09	-0.048956	-0.068086
2020-03-12	-0.083019	-0.094104
2020-03-16	-0.076121	-0.060843
2020-03-20	0.058329	0.098451
2020-03-23	-0.129805	-0.071073
2020-04-01	-0.040005	-0.064263
2020-04-07	0.087632	0.073147
2020-04-17	0.030463	0.052533
2020-04-21	-0.030275	-0.044539
2020-04-23	0.013780	0.061458
2020-04-30	0.032088	0.057093
2020-06-26	0.009146	0.050965
2020-07-28	0.015159	0.046651
2020-09-14	-0.002128	0.049408
2020-09-24	-0.029312	-0.054793
2020-10-05	0.007568	0.072262

Z-Score

A **z-score** tells you how many standard deviations a data point is away from the mean of the distribution.

$$z = \frac{x - \mu}{\sigma}$$

Where:

- x = the data point
- μ = mean of the data
- σ = standard deviation

How to interpret a z-score

Z-Score	Meaning
0	Exactly at the mean
+1	1 SD above mean
+2	2 SD above mean (unusually high)
-1	1 SD below mean
-2	2 SD below mean (unusually low)
**>	3

```
In [17]: from scipy.stats import zscore  
  
df['Nifty_zscore'] = zscore(df['Nifty'])  
print(df[['Nifty', 'Nifty_zscore']].head())
```

```
Nifty  Nifty_zscore  
Date  
2015-01-05 -0.002031 -0.216170  
2015-01-06 -0.029964 -2.688440  
2015-01-07 -0.003107 -0.311399  
2015-01-08  0.016354  1.410986  
2015-01-09  0.006060  0.499909
```

On 2015-01-06 is extreme because its Z-score **-2.688** is less than **-2**.

Correlation (R) and Coefficient of Determination (R^2)

Correlation (R):

Correlation measures the strength and direction of a linear relationship between two variables.

- R ranges from -1 to +1
- R > 0 → move together
- R < 0 → move opposite
- |R| close to 1 → strong relationship

Coefficient of Determination (R^2):

R^2 tells you how much of the variation in one variable is explained by the other.

- $R^2 = R \times R$
- R^2 ranges from 0 to 1
- Higher R^2 → stronger explanatory power

```
In [18]: # Correlation (r)
r = nifty['returns'].corr(tcs['returns'])
print("Correlation (r):", r)

# Coefficient of Determination (R²)
r_squared = r ** 2
print("Coefficient of Determination (R²):", r_squared)
```

Correlation (r): 0.44433638515383833
 Coefficient of Determination (R²): 0.19743482317158015

Correlation (r) is **0.44**, indicating a moderate positive relationship between Nifty and TCS.
 Coefficient of Determination (R²) is **0.19**, meaning only about **19 percent** of TCS's daily return variation is explained by Nifty.**

Skewness and Kurtosis

- **Normal Distribution:** Bell-shaped curve centered at the mean.
- **Skewness:** Measures symmetry of the distribution.
- **Kurtosis:** Measures extreme events and tail heaviness.

```
In [19]: from scipy.stats import skew, kurtosis

# Skewness
nifty_skew = skew(df['Nifty'])
tcs_skew = skew(df['Tcs'])

# Kurtosis (excess kurtosis = kurtosis - 3)
nifty_kurt = kurtosis(df['Nifty'], fisher=False) # normal = 3
tcs_kurt = kurtosis(df['Tcs'], fisher=False)

print("----- NIFTY -----")
print(f"Skewness : {nifty_skew:.4f}")
print(f"Kurtosis : {nifty_kurt:.4f}")

print("\n----- TCS -----")
print(f"Skewness : {tcs_skew:.4f}")
print(f"Kurtosis : {tcs_kurt:.4f}")

----- NIFTY -----
Skewness : -1.2985
Kurtosis : 23.4718

----- TCS -----
Skewness : 0.0496
Kurtosis : 6.8889
```

Nifty shows **strong negative skewness** and **extremely high kurtosis**, indicating more downside outliers and very heavy tails.

TCS shows **near-zero skewness** and **moderately high kurtosis**, meaning returns are mostly symmetric but still have some extreme moves.

KDE (Kernel Density Estimate)

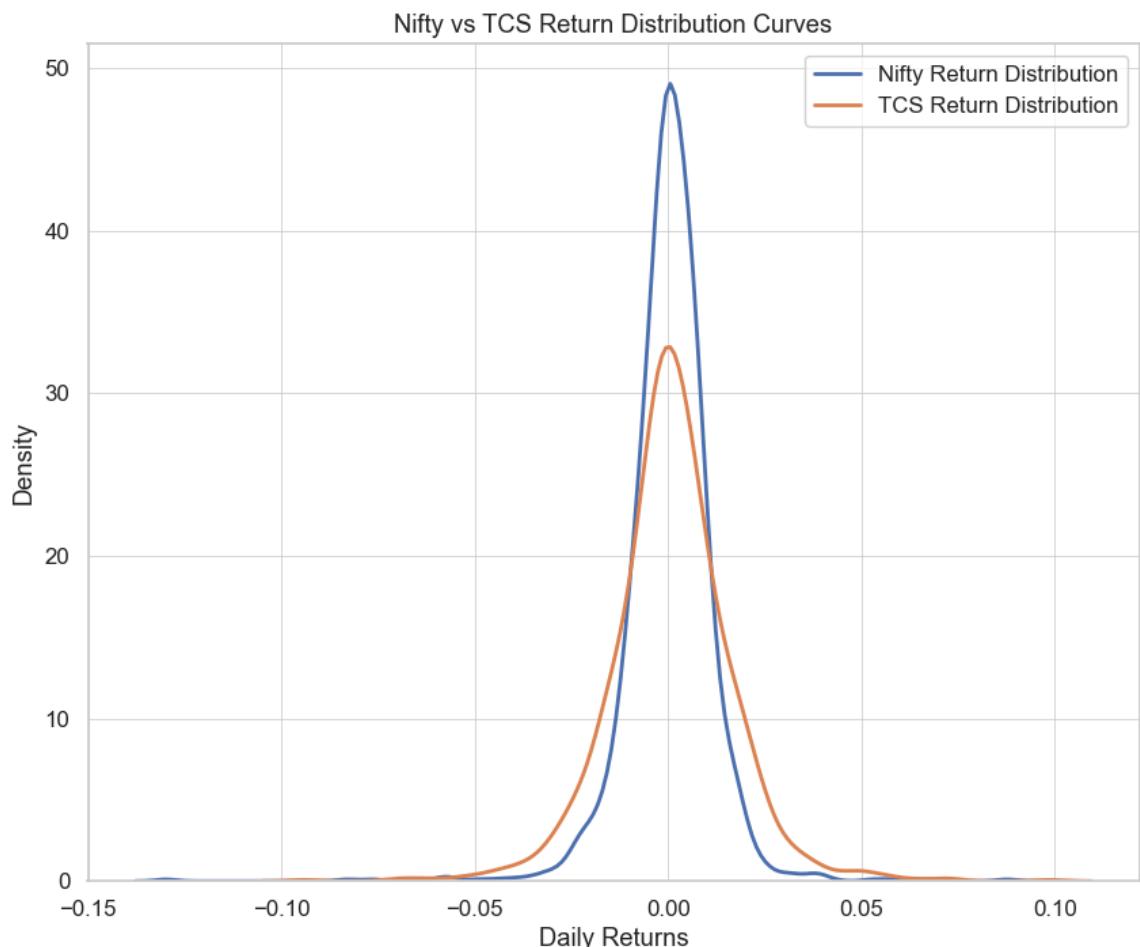
Return Distribution Plot (Nifty and TCS)

```
In [20]: plt.figure(figsize=(10,8))

sns.kdeplot(df['Nifty'], label="Nifty Return Distribution", linewidth=2)
sns.kdeplot(df['Tcs'], label="TCS Return Distribution", linewidth=2)

plt.title("Nifty vs TCS Return Distribution Curves")
plt.xlabel("Daily Returns")
plt.ylabel("Density")
plt.grid(True)
plt.legend()

plt.show()
```



Summary and Conclusion

The return distribution curves show that Nifty has a taller, narrower peak and heavier tails compared to TCS. This matches the earlier statistics where Nifty displayed strong negative skewness (-1.29) and extremely high kurtosis (23.47), indicating more frequent downside outliers and very fat tails. TCS, on the other hand, shows a more symmetric distribution (skew ~ 0) and moderately high kurtosis (6.89), which explains its comparatively smoother and wider density curve.

TTP_MACross_Metrics

```
In [72]: import pandas as pd
import numpy as np
import datetime
import yfinance as yf
import quantstats as qs
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
```

```
In [73]: df = yf.download("HINDALCO.NS", start="2010-01-01", end="2024-12-31")['Close']

[*****100%*****] 1 of 1 completed
```

```
In [74]: df.tail()
```

```
Out[74]:      Ticker HINDALCO.NS
              Date
2024-12-23    629.533325
2024-12-24    622.882080
2024-12-26    624.172607
2024-12-27    612.905273
2024-12-30    596.723877
```

```
In [75]: df.columns = ['Close']
```

```
In [76]: df.head()
```

```
Out[76]:      Close
              Date
2010-01-04  143.648712
2010-01-05  153.928131
2010-01-06  150.281998
2010-01-07  153.884216
2010-01-08  152.258835
```

```
In [77]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3699 entries, 2010-01-04 to 2024-12-30
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   Close    3699 non-null   float64 
dtypes: float64(1)
memory usage: 57.8 KB
```

```
In [78]: df1 = df.copy()
```

Work with Copy

```
In [79]: #Calculate Stock Returns  
df1['cc_returns'] = df1['Close'].pct_change()
```

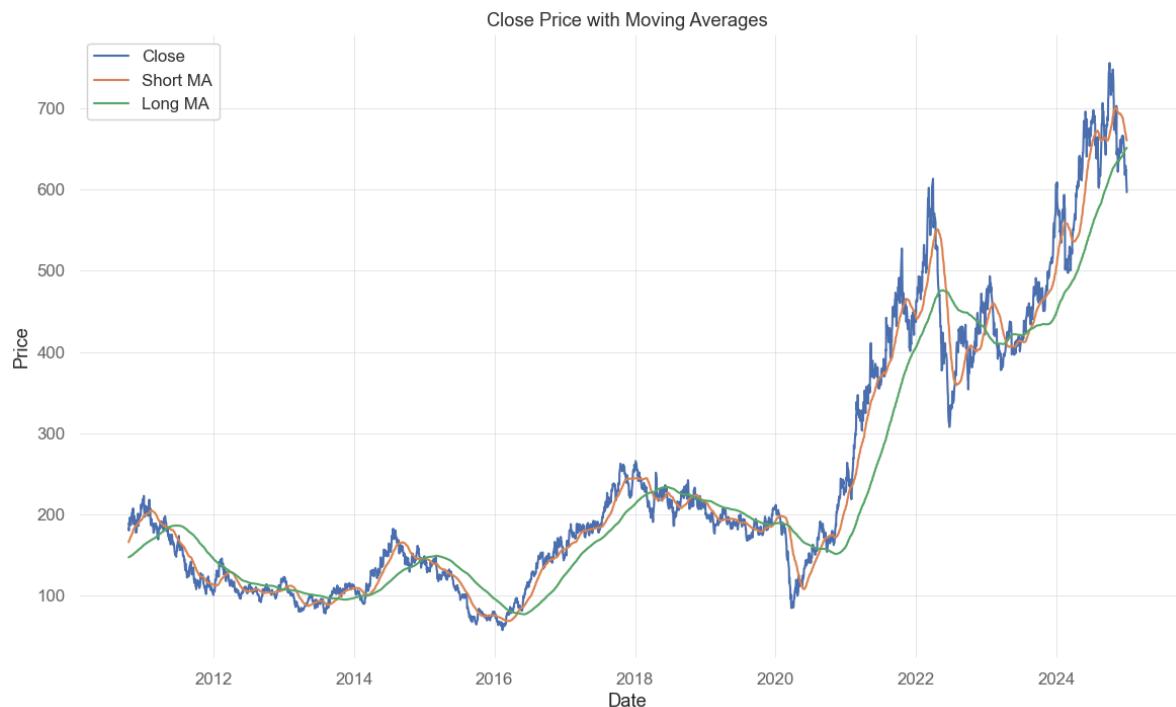
```
In [80]: short = 50  
long = 200
```

```
In [81]: df1['short_ma'] = df1['Close'].rolling(window = short).mean()
```

```
In [82]: df1['long_ma'] = df1['Close'].rolling(window = long).mean()
```

```
In [83]: df1 = df1.dropna()
```

```
In [84]: plt.figure(figsize=(14, 8))  
plt.plot(df1.index, df1['Close'], label='Close')  
plt.plot(df1.index, df1['short_ma'], label='Short MA')  
plt.plot(df1.index, df1['long_ma'], label='Long MA')  
plt.title('Close Price with Moving Averages')  
plt.xlabel('Date')  
plt.ylabel('Price')  
plt.legend()  
plt.grid(True)  
plt.show()
```



Strategy : Buy when MA 50 Crosses Above MA 200, else Exit

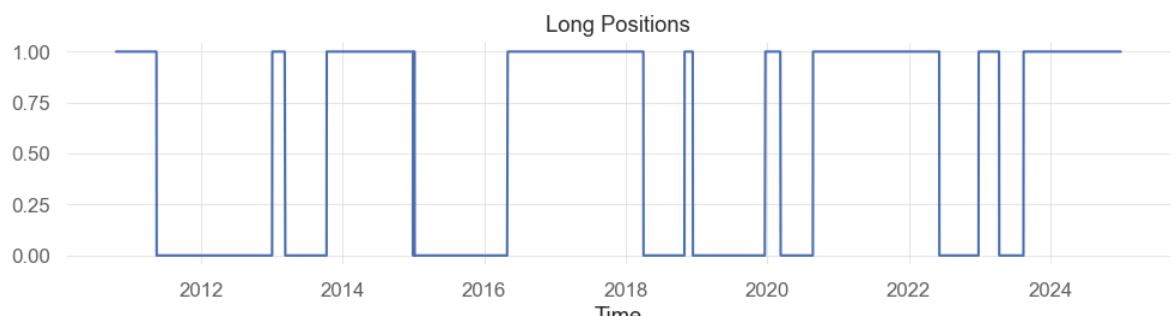
```
In [85]: df1['signal'] = np.where(df1['short_ma'] > df1['long_ma'], 1, 0)
df1['position'] = df1['signal'].shift()
```

```
In [86]: df1['position'].value_counts()
```

```
Out[86]: position
1.0    1898
0.0    1601
Name: count, dtype: int64
```

```
In [87]: plt.figure(figsize=(14, 6))
plt.plot(df1.index, df1['Close'], label='Close')
plt.plot(df1.index, df1['short_ma'], label='Short MA')
plt.plot(df1.index, df1['long_ma'], label='Long MA')
plt.title('Close Price with Moving Averages')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.show()

# Plot signals graphs
plt.figure(figsize=(10, 3))
plt.plot(df1['position'])
plt.title("Long Positions")
plt.xlabel('Time')
plt.tight_layout()
plt.show()
```



```
In [88]: df1['strategy_returns'] = df1['cc_returns'] * df1['position']
```

```
In [89]: df1['strategy_returns_cum'] = (1 + df1['strategy_returns']).cumprod()
```

```
In [90]: df1['cc_returns_cum'] = (1 + df1['cc_returns']).cumprod()
```

```
In [91]: df1[['strategy_returns_cum', 'cc_returns_cum']].plot(grid=True, figsize=(9, 5))  
plt.show()
```



```
In [92]: print('Buy and hold returns : ', np.round(df1['cc_returns_cum'][-1], 2))  
print('Strategy returns : ', np.round(df1['strategy_returns_cum'][-1], 2))
```

Buy and hold returns : 3.19

Strategy returns : 3.33

```
In [93]: returns = df1['strategy_returns'].dropna()  
returns_cc = df1['cc_returns'].dropna()
```

```
manual_cum = (1 + returns).cumprod() - 1
```

```
manual_cum.iloc[-1] * 100
```

```
print(f"Cumulative Return (%): {manual_cum.iloc[-1] * 100:.2f}%")
```

Cumulative Return (%): 232.89%

Performance Metrics

In [94]: `import pyfolio as pf`

```
pf.create_simple_tear_sheet(returns)
```

Start date 2010-10-20

End date 2024-12-30

Total months 166

Backtest

Annual return 9.048%

Cumulative returns 232.89%

Annual volatility 27.455%

Sharpe ratio 0.45

Calmar ratio 0.15

Stability 0.77

Max drawdown -58.441%

Omega ratio 1.11

Sortino ratio 0.67

In [95]: `qs.reports.basic(returns, benchmark=returns_cc)`

Performance Metrics

	Benchmark	Strategy
Start Period	2010-10-21	2010-10-21
End Period	2024-12-30	2024-12-30
Risk-Free Rate	0.0%	0.0%
Time in Market	100.0%	54.0%
Cumulative Return	231.76%	238.13%
CAGR %	9.02%	9.17%
Sharpe	0.42	0.46
Prob. Sharpe Ratio	93.96%	95.59%
Sortino	0.61	0.67
Sortino/V2	0.43	0.48
Omega	1.11	1.11
Max Drawdown	-74.32%	-58.44%

In []:

In []:

In []:

QUANTITATIVE TRADING STRATEGY

Hypothesis → Data Preparation → Rule Formalization → Backtesting → Evaluation

This section covers the complete workflow for building a quantitative trading strategy:

- **Hypothesis** – Formulating a testable market belief or trading edge.
- **Data Preparation** – Collecting, cleaning, and validating historical market data.
- **Rule Formalization** – Computing indicators and generating signals with precise entry and exit rules.
- **Backtesting** – Testing the rules on past data with realistic execution assumptions.
- **Evaluation** – Assessing performance, risk, drawdowns, and robustness.

CORE WORKFLOW :

Step 1: Strategy Setup

Determine the asset to build the strategy on

Define entry and exit rules

Step 2: Data Preparation

Import historical data

Perform data sanity checks

Step 3: Computing Indicators

Calculate technical indicators

Calculate asset daily % change

Step 4: Exploratory Visualization

Plot the asset price (line chart) and other indicators

Step 5: Signal Generation

Generate trading signals - Avoid lookahead bias

Step 6: Signal Visualization

Plot asset price with generated signals for visual confirmation

Step 7: Return Visualization

Plot strategy returns and asset returns on the same chart to compare the equity curves

Step 8: Strategy Computation

Calculate strategy returns vs asset returns

Step 9: Trading Strategy Statistical Evaluation

Generate performance statistics using PyFolio or QuantStats

Strategy 1: Trend Following | Moving Average (Short MA > Long MA)

Strategy Rules

- Compute **MA 50** and **MA 200**
- **Go Long** when MA 50 is **above** MA 200
- **Exit** when MA 50 is **below** MA 200

```
In [22]: import numpy as np
import pandas as pd
import yfinance as yf
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt
%matplotlib inline
```

Data Preparation

```
In [23]: start_date = "2010-01-01"
end_date = "2024-12-31"
data = yf.download("KPIYTECH.NS", start_date, end_date)

[*****100%*****] 1 of 1 completed
```

```
In [24]: data.head(2)
```

```
Out[24]:
```

	Price	Close	High	Low	Open	Volume
Ticker	KPIYTECH.NS	KPIYTECH.NS	KPIYTECH.NS	KPIYTECH.NS	KPIYTECH.NS	KPIYTECH.NS
Date						
2019-04-22		99.140793	99.140793	89.698818	94.419805	1066838
2019-04-23		104.052536	104.052536	100.666780	104.052536	746528

```
In [25]: data.columns = ['close', 'high', 'low', 'open', 'volume']
```

```
In [26]: data.head(2)
```

```
Out[26]:
```

	close	high	low	open	volume
Date					
2019-04-22	99.140793	99.140793	89.698818	94.419805	1066838
2019-04-23	104.052536	104.052536	100.666780	104.052536	746528

```
In [27]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1407 entries, 2019-04-22 to 2024-12-30
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   close    1407 non-null   float64
 1   high     1407 non-null   float64
 2   low      1407 non-null   float64
 3   open     1407 non-null   float64
 4   volume   1407 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 66.0 KB
```

```
In [28]: df = data.copy()
```

Compute MA50 and MA200

```
In [29]: #Compute MA 50 and MA 200
df['ma50'] = df['close'].rolling(window=50).mean()
df['ma200'] = df['close'].rolling(window=200).mean()
```

```
In [30]: df = df.dropna()
```

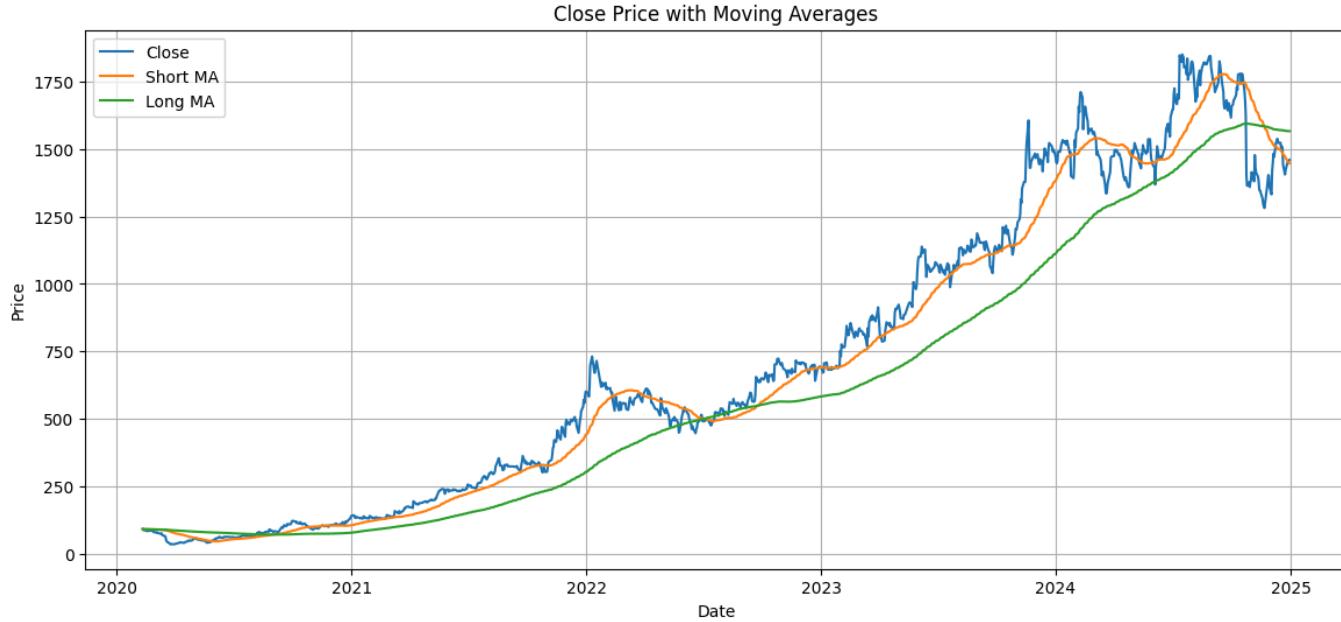
In [31]: df.info()

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1208 entries, 2020-02-11 to 2024-12-30
Data columns (total 7 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   close    1208 non-null   float64
 1   high     1208 non-null   float64
 2   low      1208 non-null   float64
 3   open     1208 non-null   float64
 4   volume   1208 non-null   int64  
 5   ma50    1208 non-null   float64
 6   ma200   1208 non-null   float64
dtypes: float64(6), int64(1)
memory usage: 75.5 KB
```

Plot the price chart and the indicators for Exploratory Visualization

In [32]:

```
plt.figure(figsize=(14, 6))
plt.plot(df.index, df['close'], label='Close')
plt.plot(df.index, df['ma50'], label='Short MA')
plt.plot(df.index, df['ma200'], label='Long MA')
plt.title('Close Price with Moving Averages')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.show()
```



Generate signals and final positions

In [33]:

```
#Go Long when MA 50 is above MA 200
#Exit when MA 50 is below MA 200
```

```
df['signal'] = np.where(df['ma50'] > df['ma200'], 1, 0)
df['position'] = df['signal'].shift(1)
```

In [34]: df.head()

Out[34]:

Date	close	high	low	open	volume	ma50	ma200	signal	position
------	-------	------	-----	------	--------	------	-------	--------	----------

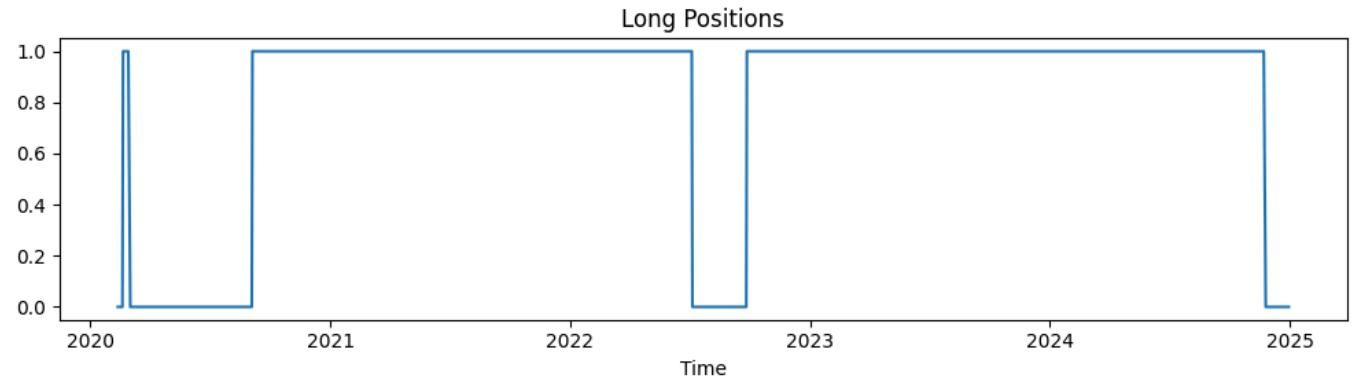
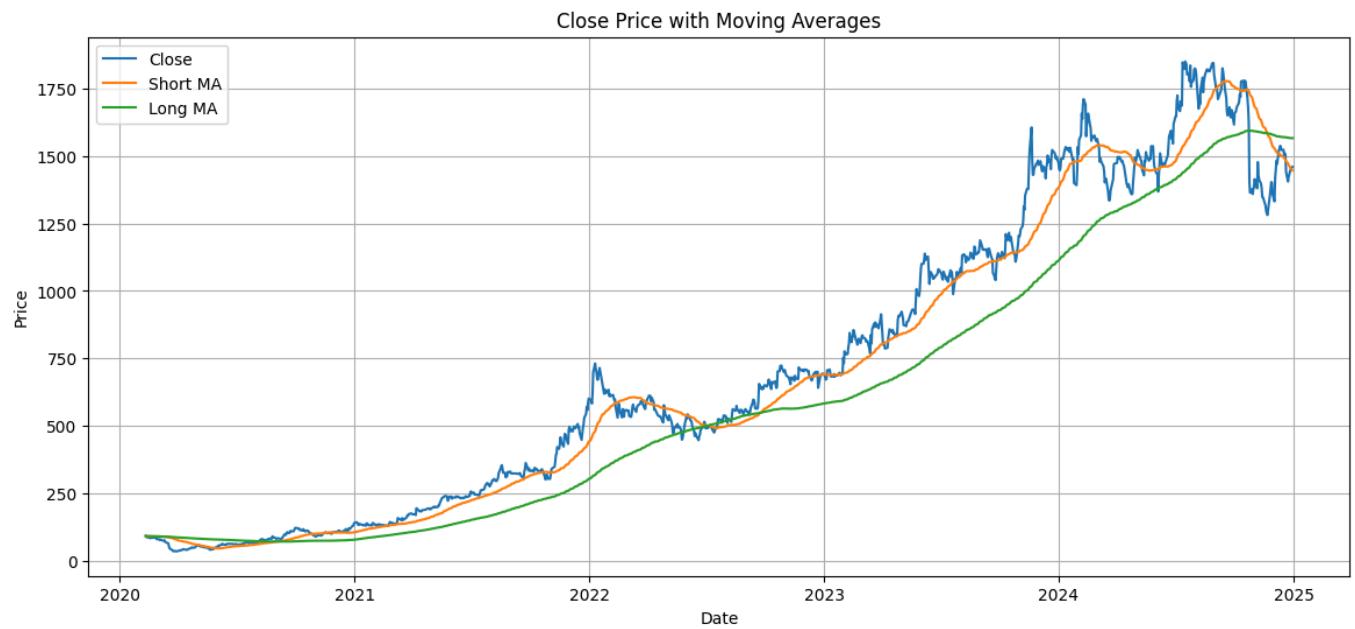
Date	close	high	low	open	volume	ma50	ma200	signal	position
2020-02-11	91.522659	96.091578	90.512685	95.129700	129514	90.056765	90.506660	0	NaN
2020-02-12	87.578972	92.725016	86.809467	91.378386	153058	90.048108	90.448851	0	0.0
2020-02-13	85.559021	88.637034	84.789523	87.675156	88598	89.980777	90.356383	0	0.0
2020-02-14	86.376625	87.530875	84.597145	85.559023	157871	89.992319	90.242014	0	0.0
2020-02-17	84.392616	90.489792	82.747350	88.554182	195849	89.968029	90.133939	0	0.0

```
In [35]: df['position'].value_counts()
```

```
Out[35]: position  
1.0    993  
0.0    214  
Name: count, dtype: int64
```

Plot signals with price chart and indicators

```
In [36]: plt.figure(figsize=(14, 6))  
plt.plot(df.index, df['close'], label='Close')  
plt.plot(df.index, df['ma50'], label='Short MA')  
plt.plot(df.index, df['ma200'], label='Long MA')  
plt.title('Close Price with Moving Averages')  
plt.xlabel('Date')  
plt.ylabel('Price')  
plt.legend()  
plt.grid(True)  
plt.show()  
  
# Plot signals graphs  
plt.figure(figsize=(10, 3))  
plt.plot(df['position'])  
plt.title("Long Positions")  
plt.xlabel('Time')  
plt.tight_layout()  
plt.show()
```



Compute daily asset returns and strategy returns

```
In [37]: df['asset_returns'] = df['close'].pct_change()  
df['strategy_returns'] = df['position'] * df['asset_returns']
```

Convert returns into equity growth

```
In [38]: df['asset_returns_eq'] = (1 + df['asset_returns'])
df['strategy_returns_eq'] = (1 + df['strategy_returns'])
```

```
In [39]: df = df.dropna()
```

```
In [40]: df.head()
```

```
Out[40]:
```

Date	close	high	low	open	volume	ma50	ma200	signal	position	asset_returns	strategy_returns	asset_
2020-02-12	87.578972	92.725016	86.809467	91.378386	153058	90.048108	90.448851	0	0.0	-0.043090	-0.0	
2020-02-13	85.559021	88.637034	84.789523	87.675156	88598	89.980777	90.356383	0	0.0	-0.023064	-0.0	
2020-02-14	86.376625	87.530875	84.597145	85.559023	157871	89.992319	90.242014	0	0.0	0.009556	0.0	
2020-02-17	84.392616	90.489792	82.747350	88.554182	195849	89.968029	90.133939	0	0.0	-0.022969	-0.0	
2020-02-18	82.698967	85.650776	81.827948	84.392626	124666	89.944494	90.029317	0	0.0	-0.020069	-0.0	

Compounding equity : ₹1 became how much?

```
In [41]: returns = df['strategy_returns_eq_cum'] = df['strategy_returns_eq'].cumprod()
asset = df['asset_returns_eq_cum'] = df['asset_returns_eq'].cumprod()
```

```
In [42]: print("Strategy Returns", np.round(returns.iloc[-1], 2))
print("Asset Returns", np.round(asset.iloc[-1], 2))
```

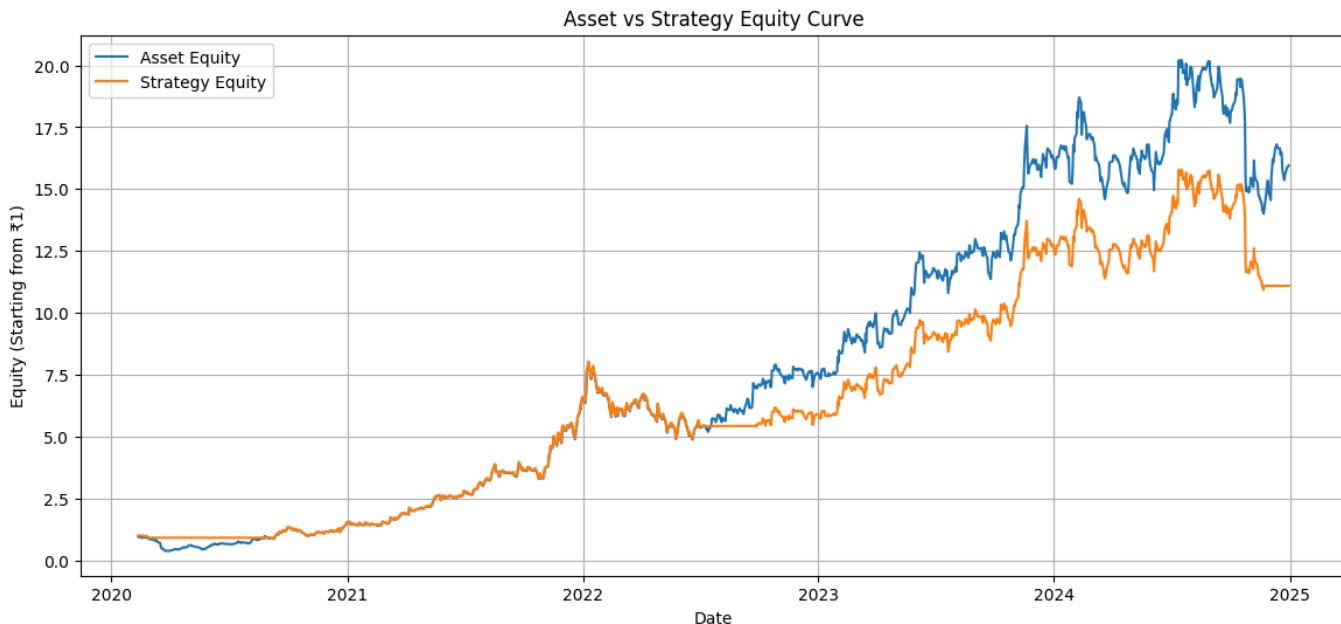
```
Strategy Returns 11.09
Asset Returns 15.96
```

Shorter plotting code

```
In [43]: #df[['asset_returns_cum', 'strategy_returns_eq_cum']].plot(figsize=(14,6),
#grid=True, title="Asset vs Strategy Equity");
```

Plotting both equity curves

```
In [44]: plt.figure(figsize=(14, 6))
plt.plot(df.index, df['asset_returns_eq_cum'], label='Asset Equity')
plt.plot(df.index, df['strategy_returns_eq_cum'], label='Strategy Equity')
plt.title("Asset vs Strategy Equity Curve")
plt.xlabel("Date")
plt.ylabel("Equity (Starting from ₹1)")
plt.legend()
plt.grid(True)
plt.show()
```



Syntax (Excel / CSV):

```
df.to_excel("file_name.xlsx")
df.to_csv("file_name.csv")
```

Explanation:

Exports the DataFrame to Excel or CSV format, preserving rows and columns for external use or sharing.

Generating Pyfolio Report

```
**Syntax (PyFolio Simple Tear Sheet):**
`pf.create_simple_tear_sheet(strategy_returns)`
```

Explanation:

Pass the `**daily `pct_change` strategy returns**` to generate a concise performance report with returns, risk metrics, drawdowns, and equity curves.

```
In [45]: import pyfolio as pf
pf.create_simple_tear_sheet(df['strategy_returns'])
```

Start date 2020-02-12

End date 2024-12-30

Total months 57

Backtest

Annual return 65.254%

Cumulative returns 1008.855%

Annual volatility 40.963%

Sharpe ratio 1.43

Calmar ratio 1.68

Stability 0.91

Max drawdown -38.919%

Omega ratio 1.32

Sortino ratio 2.40

Generating Quantstats Report

```
**Syntax (QuantStats Report with Benchmark):**
`qs.reports.basic(strategy_returns, benchmark=asset_returns)`

**Explanation:**
`strategy_returns` are the daily `pct_change` strategy returns, while `asset_returns` are used as the
benchmark to compare performance, drawdowns, and risk metrics against the underlying asset.
```

```
In [71]: import quantstats as qs
qs.reports.basic(df['strategy_returns'], benchmark=df['asset_returns'])
```

Performance Metrics

	Benchmark	Strategy
Start Period	2020-02-20	2020-02-20
End Period	2024-12-30	2024-12-30
Risk-Free Rate	0.0%	0.0%
Time in Market	100.0%	83.0%
Cumulative Return	1,583.16%	1,008.86%
CAGR %	80.83%	65.67%
Sharpe	1.5	1.43
Prob. Sharpe Ratio	99.95%	99.94%
Sortino	2.4	2.41
Sortino/V2	1.69	1.7
Omega	1.32	1.32
Max Drawdown	-60.8%	-38.92%

Strategy 2: Momentum | (Short EMA > Long EMA) & (ROC > 0)

Strategy Rules

- For Determining Trend: Compute **EMA 50** and **EMA 200**
- For Determining Momentum: Compute **ROC (20)**
- **Go Long** when EMA 50 is **above** EMA 200 and ROC is **above 0**
- **else Exit**

```
In [ ]: import numpy as np
import pandas as pd
import yfinance as yf
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt
%matplotlib inline
```

```
In [47]: start_date = "2015-01-01"
end_date = "2022-11-30"
data1 = yf.download("^NSEI", start_date, end_date)

[*****100%*****] 1 of 1 completed
```

```
In [48]: data1.head(2)
```

```
Out[48]:
```

Price	Close	High	Low	Open	Volume
Ticker	^NSEI	^NSEI	^NSEI	^NSEI	^NSEI
Date					
2015-01-02	8395.450195	8410.599609	8288.700195	8288.700195	101900
2015-01-05	8378.400391	8445.599609	8363.900391	8407.950195	118200

```
In [49]: data1.columns = ['close', 'high', 'low', 'open', 'volume']
```

```
In [50]: data1.head(2)
```

```
Out[50]:
```

Date	close	high	low	open	volume
2015-01-02	8395.450195	8410.599609	8288.700195	8288.700195	101900
2015-01-05	8378.400391	8445.599609	8363.900391	8407.950195	118200

```
In [51]: df1 = data1.copy()
```

```
In [52]: df1['ema_short'] = df1['close'].ewm(span=50).mean()  
df1['ema_long'] = df1['close'].ewm(span=200).mean()
```

```
In [53]: df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 1945 entries, 2015-01-02 to 2022-11-29  
Data columns (total 7 columns):  
 #   Column      Non-Null Count  Dtype     
---  --          --          --         
 0   close        1945 non-null   float64  
 1   high         1945 non-null   float64  
 2   low          1945 non-null   float64  
 3   open         1945 non-null   float64  
 4   volume       1945 non-null   int64  
 5   ema_short    1945 non-null   float64  
 6   ema_long     1945 non-null   float64  
dtypes: float64(6), int64(1)  
memory usage: 121.6 KB
```

```
In [54]: df1.head()
```

```
Out[54]:
```

Date	close	high	low	open	volume	ema_short	ema_long
2015-01-02	8395.450195	8410.599609	8288.700195	8288.700195	101900	8395.450195	8395.450195
2015-01-05	8378.400391	8445.599609	8363.900391	8407.950195	118200	8386.754795	8386.882668
2015-01-06	8127.350098	8327.849609	8111.350098	8325.299805	172800	8296.805239	8299.505268
2015-01-07	8102.100098	8151.200195	8065.450195	8118.649902	164100	8245.170230	8249.411251
2015-01-08	8234.599609	8243.500000	8167.299805	8191.399902	143800	8242.883663	8246.389381

```
In [55]: df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 1945 entries, 2015-01-02 to 2022-11-29  
Data columns (total 7 columns):  
 #   Column      Non-Null Count  Dtype     
---  --          --          --         
 0   close        1945 non-null   float64  
 1   high         1945 non-null   float64  
 2   low          1945 non-null   float64  
 3   open         1945 non-null   float64  
 4   volume       1945 non-null   int64  
 5   ema_short    1945 non-null   float64  
 6   ema_long     1945 non-null   float64  
dtypes: float64(6), int64(1)  
memory usage: 121.6 KB
```

```
In [56]: # Compute the technical indicator
n = 20 # Lookback period, # Rate of change (normalized momentum)
df1['roc'] = ((df1['close'] / df1['close'].shift(n)) - 1) * 100
print(df1[['close', 'roc']].tail(12))
```

Date	close	roc
2022-11-14	18329.150391	7.727599
2022-11-15	18403.400391	7.085549
2022-11-16	18409.650391	6.341626
2022-11-17	18343.900391	4.900518
2022-11-18	18307.650391	4.541966
2022-11-21	18159.949219	3.393314
2022-11-22	18244.199219	3.799994
2022-11-23	18267.250000	3.025817
2022-11-24	18484.099609	4.688115
2022-11-25	18512.750000	4.373925
2022-11-28	18562.750000	4.362500
2022-11-29	18618.050781	3.363562

```
In [57]: condition1 = df1['roc'] > 0
condition2 = df1['ema_short'] > df1['ema_long']
```

```
In [58]: df1['signal'] = np.where((condition1 & condition2), 1, 0)
```

```
In [59]: df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1945 entries, 2015-01-02 to 2022-11-29
Data columns (total 9 columns):
 #   Column      Non-Null Count  Dtype  
---  --          -----          ----- 
 0   close       1945 non-null   float64
 1   high        1945 non-null   float64
 2   low         1945 non-null   float64
 3   open        1945 non-null   float64
 4   volume      1945 non-null   int64  
 5   ema_short   1945 non-null   float64
 6   ema_long    1945 non-null   float64
 7   roc         1925 non-null   float64
 8   signal      1945 non-null   int32  
dtypes: float64(7), int32(1), int64(1)
memory usage: 144.4 KB
```

```
In [60]: df1['position'] = df1['signal'].shift()
```

```
In [61]: df1['signal'].value_counts()
```

```
Out[61]: signal
0    1003
1     942
Name: count, dtype: int64
```

```
In [62]: df1 = df1.dropna()
```

Plotting Price, MA's, ROC, Position

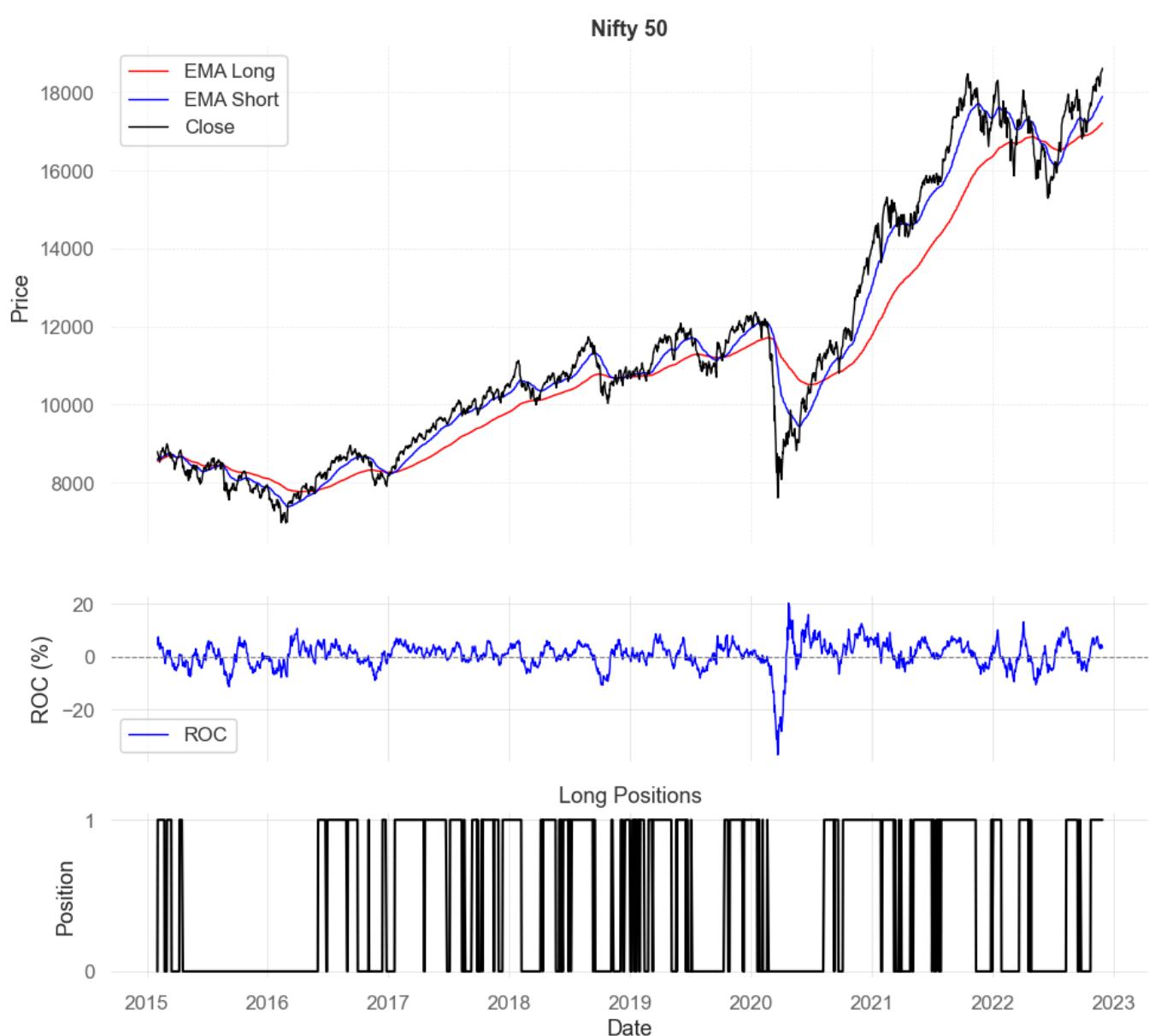
```
In [63]: import matplotlib.pyplot as plt
# Create 3 subplots: Price, ROC, Signals
fig, (ax1, ax2, ax3) = plt.subplots(
    3, 1, figsize=(10, 9), sharex=True,
    gridspec_kw={'height_ratios': [3, 1, 1]})

# --- Price Chart ---
ax1.plot(df1.index, df1['ema_long'], color='red', linewidth=1, label='EMA Long')
ax1.plot(df1.index, df1['ema_short'], color='blue', linewidth=1, label='EMA Short')
ax1.plot(df1.index, df1['close'], color='black', linewidth=1, label='Close')
ax1.set_title("Nifty 50", fontsize=13, fontweight='bold')
ax1.set_ylabel("Price")
ax1.grid(True, linestyle='--', alpha=0.5)
ax1.legend()

# --- ROC Chart ---
ax2.plot(df1.index, df1['roc'], color='blue', linewidth=1, label='ROC')
ax2.axhline(0, color='gray', linewidth=0.8, linestyle='--')
ax2.set_ylabel("ROC (%)")
ax2.grid(True)
ax2.legend()

# --- Signals Chart ---
ax3.plot(df1.index, df1['position'], color='black')
ax3.set_title("Long Positions")
ax3.set_xlabel("Date")
ax3.set_ylabel("Position")
ax3.set_yticks([0, 1])  # only show 0 and 1
ax3.grid(True)

plt.tight_layout()
plt.show()
```



```
In [64]: df1['asset_returns'] = df1['close'].pct_change()
df1['strategy_returns'] = df1['position'] * df1['asset_returns']
```

```
In [65]: df1['asset_returns_eq'] = (1 + df1['asset_returns'])
df1['strategy_returns_eq'] = (1 + df1['strategy_returns'])
```

```
In [66]: df1['asset_returns_eq_cum'] = df1['asset_returns_eq'].cumprod()
df1['strategy_returns_eq_cum'] = df1['strategy_returns_eq'].cumprod()
```

```
In [67]: returns1 = df1['strategy_returns_eq_cum'].dropna()
asset1 = df1['asset_returns_eq_cum'].dropna()
```

```
In [68]: print("Strategy Returns", np.round(returns1.iloc[-1], 2))
print("Asset Returns", np.round(asset1.iloc[-1], 2))
```

Strategy Returns 1.62
Asset Returns 2.12

```
In [69]: plt.figure(figsize=(14, 6))
plt.plot(df1.index, df1['asset_returns_eq_cum'], label='Asset Equity')
plt.plot(df1.index, df1['strategy_returns_eq_cum'], label='Strategy Equity')
plt.title("Asset vs Strategy Equity Curve")
plt.xlabel("Date")
plt.ylabel("Equity (Starting from ₹1)")
plt.legend()
plt.grid(True)
plt.show()
```



```
In [70]: import quantstats as qs
qs.reports.basic(returns1, benchmark=asset1)
```

Performance Metrics

	Benchmark	Strategy
Start Period	2015-02-04	2015-02-04
End Period	2022-11-29	2022-11-29
Risk-Free Rate	0.0%	0.0%
Time in Market	100.0%	49.0%
Cumulative Return	112.62%	62.28%
CAGR %	10.39%	6.55%
Sharpe	0.65	0.81
Prob. Sharpe Ratio	95.98%	98.61%
Sortino	0.89	1.14
Sortino/v2	0.63	0.81
Omega	1.21	1.21
Max Drawdown	-38.44%	-10.93%

In []:

Strategy 3: Mean Reversion | Bollinger Band Long / Short

Strategy Rules

- Compute **Bollinger Bands (20-period, $\pm 2\sigma$)** on the close price
- **Go Long** when price **closes below the Lower Bollinger Band**
- **Go Short** when price **closes above the Upper Bollinger Band**
- Positions **flip between Long (+1) and Short (-1)** on opposite signals
- The strategy **remains always in the market** (no flat state)

```
In [43]: import numpy as np
import pandas as pd
import yfinance as yf
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt
%matplotlib inline
```

```
In [44]: data = yf.download("^NSEI", start="2022-01-01", end="2024-12-31" )
[*****100%*****] 1 of 1 completed
```

```
In [45]: data.head(2)
```

```
Out[45]:
```

	Price	Close	High	Low	Open	Volume
Ticker	[^] NSEI					
Date						
2022-01-03	17625.699219	17646.650391	17383.300781	17387.150391	200500	
2022-01-04	17805.250000	17827.599609	17593.550781	17681.400391	247400	

```
In [46]: data.columns = ['close', 'high', 'low', 'open', 'volume']
```

```
In [47]: data.head(2)
```

```
Out[47]:
```

	close	high	low	open	volume
Date					
2022-01-03	17625.699219	17646.650391	17383.300781	17387.150391	200500
2022-01-04	17805.250000	17827.599609	17593.550781	17681.400391	247400

```
In [48]: data = data.drop(['volume'], axis=1)
```

```
In [49]: data.head(2)
```

```
Out[49]:
```

	close	high	low	open
Date				
2022-01-03	17625.699219	17646.650391	17383.300781	17387.150391
2022-01-04	17805.250000	17827.599609	17593.550781	17681.400391

```
In [50]: data.index.min()
```

```
Out[50]: Timestamp('2022-01-03 00:00:00')
```

```
In [51]: data.index.max()
```

```
Out[51]: Timestamp('2024-12-30 00:00:00')
```

WORK WITH A COPY

```
In [52]: df = data.copy()

In [53]: #Compute Bollinger Bands (20-period,  $\pm 2\sigma$ ) on the adjusted close price
df['sma20'] = df['close'].rolling(window=20).mean()
df['std20'] = df['close'].rolling(window=20).std()
df['upperband'] = df['sma20'] + (2 * df['std20'])
df['lowerband'] = df['sma20'] - (2 * df['std20'])

In [54]: buy_condition = df['close'] < df['lowerband']
sell_condition = df['close'] > df['upperband']

In [55]: df['signal_long'] = np.where(buy_condition, 1, 0)
df['signal_short'] = np.where(sell_condition, -1, 0)

In [56]: df['signal'] = df['signal_long'] + df['signal_short']

In [57]: df['signal_shift'] = df['signal'].shift()

In [58]: #df['signal'].value_counts()

In [59]: # Create the position column
df['position'] = df['signal_shift'].replace(0, np.nan).ffill().fillna(0)

In [60]: df['position'].value_counts()

Out[60]: position
1.0    357
-1.0   343
0.0    38
Name: count, dtype: int64

In [61]: df['asset_returns'] = df['close'].pct_change()
df['strategy_returns'] = df['position'] * df['asset_returns']

df['asset_returns_eq_cum'] = (1 + df['asset_returns']).cumprod()
df['strategy_returns_eq_cum'] = (1 + df['strategy_returns']).cumprod()

returns1 = df['strategy_returns_eq_cum'].dropna()
asset1 = df['asset_returns_eq_cum'].dropna()

print("Strategy Returns", np.round(returns1.iloc[-1], 2))
print("Asset Returns", np.round(asset1.iloc[-1], 2))
print("Strategy Return (%)", np.round((returns1.iloc[-1] - 1) * 100, 2))
print("Asset Return (%)", np.round((asset1.iloc[-1] - 1) * 100, 2))

Strategy Returns 1.11
Asset Returns 1.34
Strategy Return (%) 10.92
Asset Return (%) 34.15
```

```
In [62]: import matplotlib.pyplot as plt
```

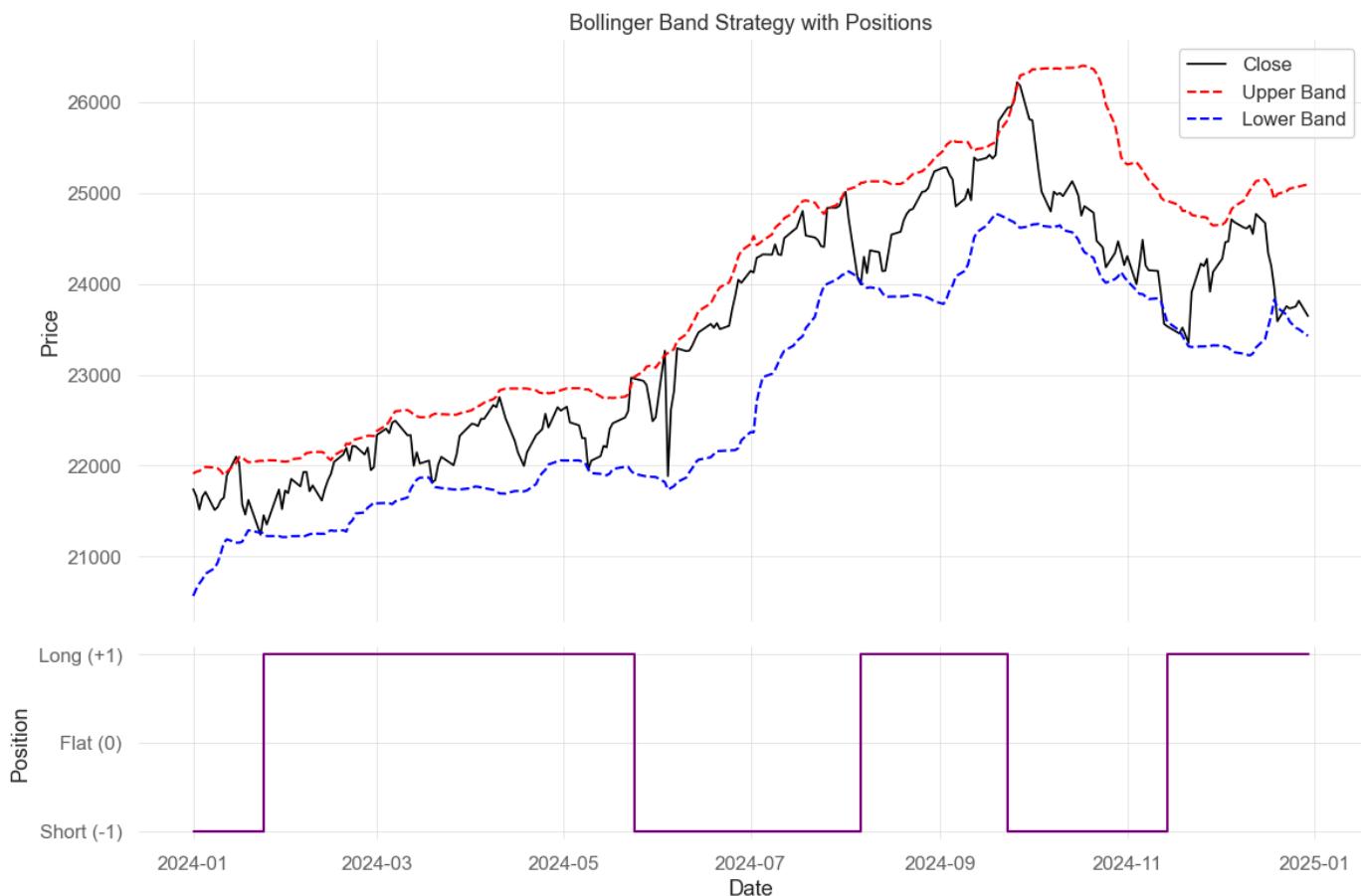
```
# Slice by year
df_year = df.loc['2024']

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12,8), sharex=True,
                             gridspec_kw={'height_ratios':[3,1]})

# --- TOP: Price + Bollinger Bands ---
ax1.plot(df_year.index, df_year['close'], label='Close', color='black', linewidth=1.2)
ax1.plot(df_year.index, df_year['upperband'], '--', label='Upper Band', color='red')
ax1.plot(df_year.index, df_year['lowerband'], '--', label='Lower Band', color='blue')
ax1.set_title("Bollinger Band Strategy with Positions")
ax1.set_ylabel("Price")
ax1.legend()
ax1.grid(True)

# --- BOTTOM: Position series ---
ax2.step(df_year.index, df_year['position'], where='post', color='purple')
ax2.set_ylabel("Position")
ax2.set_xlabel("Date")
ax2.set_yticks([-1,0,1])
ax2.set_yticklabels(['Short (-1)', 'Flat (0)', 'Long (+1)'])
ax2.grid(True)

plt.tight_layout()
plt.show()
```



Strategy 4: Buy and Hold (Blue Chips)

Strategy Rules

- Select 5 stocks from the **Nifty50 Universe**
- **Hold positions continuously** without exits
- Returns are driven by **long-term price appreciation**
- The strategy **remains fully invested** at all times

Evaluate Buy and Hold Portfolio Performance

- Plot the Cumulative Portfolio Equity Curve
- Evaluate **CAGR, volatility, and drawdown**

```
In [63]: import yfinance as yf
import pandas as pd
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt
import numpy as np

# Define stock symbols (Yahoo Finance format for NSE stocks)
symbols = ['TCS.NS', 'SUNPHARMA.NS', 'HDFCBANK.NS', 'RELIANCE.NS', 'JSWSTEEL.NS']

# Download last 10 years of daily data
data = yf.download(tickers=symbols, start='2014-01-01', end='2024-12-31')['Close']

# Display the first few rows
print(data.head())
```

[*****100%*****] 5 of 5 completed

Ticker	HDFCBANK.NS	JSWSTEEL.NS	RELIANCE.NS	SUNPHARMA.NS	TCS.NS
Date					
2014-01-01	150.498581	89.249580	188.334106	526.620117	825.898865
2014-01-02	148.654526	88.695824	185.409912	528.000183	831.153992
2014-01-03	150.034744	87.373001	183.132004	533.382874	852.325623
2014-01-06	149.717972	90.910820	181.161362	541.249817	858.999512
2014-01-07	150.328903	90.317513	178.438492	546.724365	846.169739

```
In [64]: data.index = pd.to_datetime(data.index)
```

```
In [65]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2710 entries, 2014-01-01 to 2024-12-30
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   HDFCBANK.NS  2710 non-null   float64
 1   JSWSTEEL.NS  2710 non-null   float64
 2   RELIANCE.NS  2710 non-null   float64
 3   SUNPHARMA.NS 2710 non-null   float64
 4   TCS.NS       2710 non-null   float64
dtypes: float64(5)
memory usage: 127.0 KB
```

```
In [66]: stock_data = data.copy()
```

Resampling using .resample()

.resample() → Used to **group time series DataFrames** into a new frequency.

Syntax:

```
df.resample(freq).last()
```

- .resample() requires a **DatetimeIndex** in the DataFrame
- Common frequencies:
 - 'D' → Daily
 - 'W' → Weekly
 - 'M' → Monthly
 - 'Q' → Quarterly
 - 'Y' → Yearly

```
In [67]: # Convert daily to monthly frequency
monthly_stocks_data = stock_data.resample('M').last()
```

```
In [68]: monthly_stocks_data.head()
```

```
Out[68]: Ticker HDFCBANK.NS JSWSTEEL.NS RELIANCE.NS SUNPHARMA.NS TCS.NS
```

Date	HDFCBANK.NS	JSWSTEEL.NS	RELIANCE.NS	SUNPHARMA.NS	TCS.NS
2014-01-31	142.206055	80.631393	176.118210	541.663818	861.087158
2014-02-28	151.505417	76.992485	169.507019	593.189453	874.420471
2014-03-31	169.425415	91.064644	197.223175	528.828430	819.628418
2014-04-30	163.203217	95.195732	198.293243	582.240295	841.164795
2014-05-31	179.369659	106.477203	227.832855	561.859924	822.779358

```
In [69]: # Calculate monthly percentage change
```

```
monthly_percent_change = monthly_stocks_data.pct_change()
```

```
# Calculate portfolio returns, it's the arithmetic average of the component returns.
```

```
portfolio_returns = monthly_percent_change.mean(axis=1)
```

```
In [70]: monthly_percent_change
```

```
Out[70]: Ticker HDFCBANK.NS JSWSTEEL.NS RELIANCE.NS SUNPHARMA.NS TCS.NS
```

Date	HDFCBANK.NS	JSWSTEEL.NS	RELIANCE.NS	SUNPHARMA.NS	TCS.NS
2014-01-31	NaN	NaN	NaN	NaN	NaN
2014-02-28	0.065394	-0.045130	-0.037538	0.095125	0.015484
2014-03-31	0.118280	0.182773	0.163510	-0.108500	-0.062661
2014-04-30	-0.036725	0.045364	0.005426	0.101000	0.026276
2014-05-31	0.099057	0.118508	0.148969	-0.035003	-0.021857
...
2024-08-31	0.013090	0.013843	0.006193	0.059499	0.038401
2024-09-30	0.058128	0.094517	-0.021893	0.057668	-0.062641
2024-10-31	0.002107	-0.064463	-0.097879	-0.040380	-0.068026
2024-11-30	0.034770	0.002439	-0.029916	-0.036779	0.076201
2024-12-31	-0.010106	-0.069410	-0.063071	0.057836	-0.026236

132 rows × 5 columns

```
In [71]: portfolio_returns
```

```
Out[71]: Date
```

```
2014-01-31      NaN
2014-02-28    0.018667
2014-03-31    0.058680
2014-04-30    0.028268
2014-05-31    0.061935
...
2024-08-31    0.026205
2024-09-30    0.025156
2024-10-31   -0.053728
2024-11-30    0.009343
2024-12-31   -0.022197
Freq: ME, Length: 132, dtype: float64
```

```
In [72]: portfolio_returns = portfolio_returns.dropna()
```

```
In [73]: # Calculate cumulative portfolio returns
```

```
cum_portfolio_returns = (portfolio_returns+1).cumprod()
```

```
In [74]: # Plot cumulative strategy returns
cum_portfolio_returns.plot(figsize=(12, 6))
# Set title and labels
plt.title('Buy and Hold Strategy Returns', fontsize=14)
plt.xlabel('Date', fontsize=12)
plt.ylabel('Cumulative Returns', fontsize=12)
# Add gridlines
plt.grid(True)
```



```
In [75]: # --- CAGR ---
n_months = len(portfolio_returns)
cumulative_return = (1 + portfolio_returns).prod()
cagr = cumulative_return ** (12 / n_months) - 1

# --- Volatility (annualized) ---
volatility = portfolio_returns.std() * np.sqrt(12)

# --- Drawdowns ---
cumulative = (1 + portfolio_returns).cumprod()
running_max = cumulative.cummax()
drawdown = (cumulative - running_max) / running_max
max_drawdown = drawdown.min()

print("CAGR:", np.round(cagr, 2))
print("Volatility:", np.round(volatility, 2))
print("Max Drawdown:", np.round(max_drawdown, 2))
```

```
CAGR: 0.2
Volatility: 0.17
Max Drawdown: -0.29
```

A Glimse of Technical Analysis Library

Overview

- TA-Lib is a widely used open-source library for **financial market analysis**
- Provides **over 150 functions** for **technical indicators** (e.g., RSI, MACD, Bollinger Bands)

Use Cases

- Building **trading strategies** with technical indicators

```
In [94]: import talib as ta
```

```
In [95]: help(ta)
```

Help on package talib:

NAME
 talib

PACKAGE CONTENTS
 _ta_lib
 abstract
 deprecated
 stream

SUBMODULES
 func

FUNCTIONS

```
ACCBANDS(high, low, close, timeperiod=-2147483648)  
ACCBANDS(ndarray high, ndarray low, ndarray close, int timeperiod=-0x80000000)  
ACCBANDS(high, low, close[, timeperiod=?])
```

```
In [96]: help(ta.RSI)
```

Help on function RSI in module talib._ta_lib:

```
RSI(real, timeperiod=-2147483648)  
RSI(ndarray real, int timeperiod=-0x80000000)  
RSI(real[, timeperiod=?])
```

Relative Strength Index (Momentum Indicators)

Inputs:
 real: (any ndarray)
Parameters:
 timeperiod: 14
Outputs:
 real

```
In [116]: data = yf.download("SUNPHARMA.NS", start='2024-01-01', end='2024-12-31')
```

```
[*****100%*****] 1 of 1 completed
```

```
In [117]: data.columns = ['close', 'high', 'low', 'open', 'volume']  
print(data.head(2))  
data.info()
```

	close	high	low	open	volume
Date					
2024-01-01	1237.588501	1241.221921	1229.977860	1238.325015	733452
2024-01-02	1272.891968	1275.641667	1234.544228	1238.324974	3174447

```
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 245 entries, 2024-01-01 to 2024-12-30  
Data columns (total 5 columns):  
 #   Column  Non-Null Count  Dtype    
---  --    
 0   close    245 non-null    float64  
 1   high     245 non-null    float64  
 2   low      245 non-null    float64  
 3   open     245 non-null    float64  
 4   volume   245 non-null    int64  
dtypes: float64(4), int64(1)  
memory usage: 11.5 KB
```

```
In [118]: df = data.copy()
```

```
In [119]: # 3. Relative Strength Index (RSI)
df['RSI14'] = ta.RSI(df['close'], timeperiod=14)

# 4. Moving Average Convergence Divergence (MACD)
df['MACD'], df['MACD_signal'], df['MACD_hist'] = ta.MACD(df['close'],
                                                       fastperiod=12,
                                                       slowperiod=26,
                                                       signalperiod=9)

# 5. Bollinger Bands
df['UpperBB'], df['MiddleBB'], df['LowerBB'] = ta.BBANDS(df['close'],
                                                       timeperiod=20,
                                                       nbdevup=2,
                                                       nbdevdn=2,
                                                       matype=0)

# 6. Average True Range (ATR)
df['ATR14'] = ta.ATR(df['high'], df['low'], df['close'], timeperiod=14)
```

```
In [120]: df.tail()
```

```
Out[120]:
```

Date	close	high	low	open	volume	RSI14	MACD	MACD_signal	MACD_hist	UpperBB	MiddleBB
2024-12-23	1797.811279	1811.087327	1780.027401	1794.244611	1299181	52.308389	0.526447	-2.761932	3.288379	1826.176575	1780.842
2024-12-24	1802.170654	1814.802692	1786.467322	1798.108612	1121058	53.492723	1.445326	-1.920480	3.365807	1826.827471	1783.685
2024-12-26	1824.313843	1827.979683	1797.613232	1803.161397	1402706	59.054785	3.915183	-0.753348	4.668531	1828.237827	1788.238
2024-12-27	1844.029785	1848.735838	1819.756458	1828.425504	1668898	63.267199	7.378420	0.873006	6.505414	1827.442201	1794.529
2024-12-30	1866.470215	1878.210555	1837.837573	1844.822395	6021550	67.380609	11.797818	3.057968	8.739850	1842.444527	1799.631

```
In [121]: dir(ta)
```

```
Out[121]: ['ACCBANDS',
 'ACOS',
 'AD',
 'ADD',
 'ADOSC',
 'ADX',
 'ADXR',
 'APO',
 'AROON',
 'AROONOSC',
 'ASIN',
 'ATAN',
 'ATR',
 'AVGDEV',
 'AVGPRICE',
 'BBANDS',
 'BETA',
 'BOP',
 'CCI',
 'CCI_group']
```

In [123]:

```
import matplotlib.pyplot as plt
import numpy as np

# Numeric x-axis (bulletproof for bars)
x = np.arange(len(df))

# Green if close > open else red
colors = ['g' if c > o else 'r' for c, o in zip(df['close'], df['open'])]

fig, (ax1, ax2, ax3, ax4) = plt.subplots(
    4, 1, figsize=(14, 10), sharex=True,
    gridspec_kw={'height_ratios': [4, 1, 1, 1]})

# ----- Price -----
ax1.plot(x, df['close'], color='blue', linewidth=1.5)
ax1.set_title('Price')
ax1.grid(True)

# ----- Volume (BOLD & WIDE) -----
ax2.bar(
    x,
    df['volume'],
    color=colors,
    width=0.9,
    alpha=1.0,
    edgecolor='black',
    linewidth=0.6
)
ax2.set_title('Volume')
ax2.grid(True)

# ----- RSI -----
ax3.plot(x, df['RSI14'], color='purple', linewidth=1.2)
ax3.axhline(70, linestyle='--', color='r')
ax3.axhline(30, linestyle='--', color='g')
ax3.set_title('RSI')
ax3.grid(True)

# ----- MACD -----
ax4.plot(x, df['MACD'], color='blue', linewidth=1.2)
ax4.plot(x, df['MACD_signal'], color='orange', linewidth=1.2)
ax4.bar(
    x,
    df['MACD_hist'],
    color='gray',
    width=0.9,
    alpha=1.0,
    edgecolor='black',
    linewidth=0.6
)
ax4.set_title('MACD')
ax4.grid(True)

# ----- X-axis Labels (dates only at bottom) -----
ax4.set_xticks(x[::20])
ax4.set_xticklabels(df.index.strftime('%Y-%m-%d')[::20], rotation=45)

plt.tight_layout()
plt.show();
```



In []:

In []:

Assignment15_Mini_Project

TPP Foundation | Mini-Project Assignment

Submission is MANDATORY to receive Certification

Instructions

- Complete **all assignments** covered in this module
- Implement all strategies in **different Jupyter Notebooks (.ipynb)**
- Ensure all code cells execute without errors
- Include plots, outputs, and performance metrics for each assignment
- Name the notebook files using **your fullname_1, fullname_2...** (for example: **rahul_desai_1.ipynb**)
- Send all the completed **.ipynb files in a zip folder via email before 20 January, 2026**

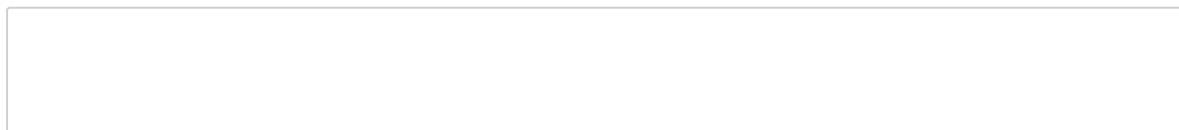
Assignment 15A: Trend Strength (Stocks)

Assignment Rules

- Compute **200-period moving average**
- Compute **20-period moving average**
- **Go Long** when the following conditions are satisfied:
 - 20 MA is above 200 MA
- **Exit** when the following conditions reverse:
 - 20 MA is below 200 MA
- Apply the strategy on stocks
- Long-only strategy (no short positions)
- Avoid look-ahead bias

Output Requirements

- Plot **price chart, 200 MA, 20 MA, and position**
- Print **cumulative asset returns** and **cumulative strategy returns**
- Plot **cumulative asset vs strategy returns**
- Use **QuantStats** to generate performance metrics



```
In [6]: import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
import quantstats as qs
import warnings
warnings.filterwarnings("ignore")
```

```
In [10]: start_date = "2018-01-01"
end_date = "2024-12-31"
df = yf.download("LT.NS", start_date, end_date)

df.index = pd.to_datetime(df.index)

[*****100%*****] 1 of 1 completed
```

```
In [11]: df["MA20"] = df["Close"].rolling(20).mean()
df["MA200"] = df["Close"].rolling(200).mean()
```

```
In [12]: df["Signal"] = 0
df.loc[df["MA20"] > df["MA200"], "Signal"] = 1
```

```
In [13]: df["Position"] = df["Signal"].shift(1)
df["Position"].fillna(0, inplace=True)
```

```
In [14]: df["Asset_Return"] = df["Close"].pct_change()

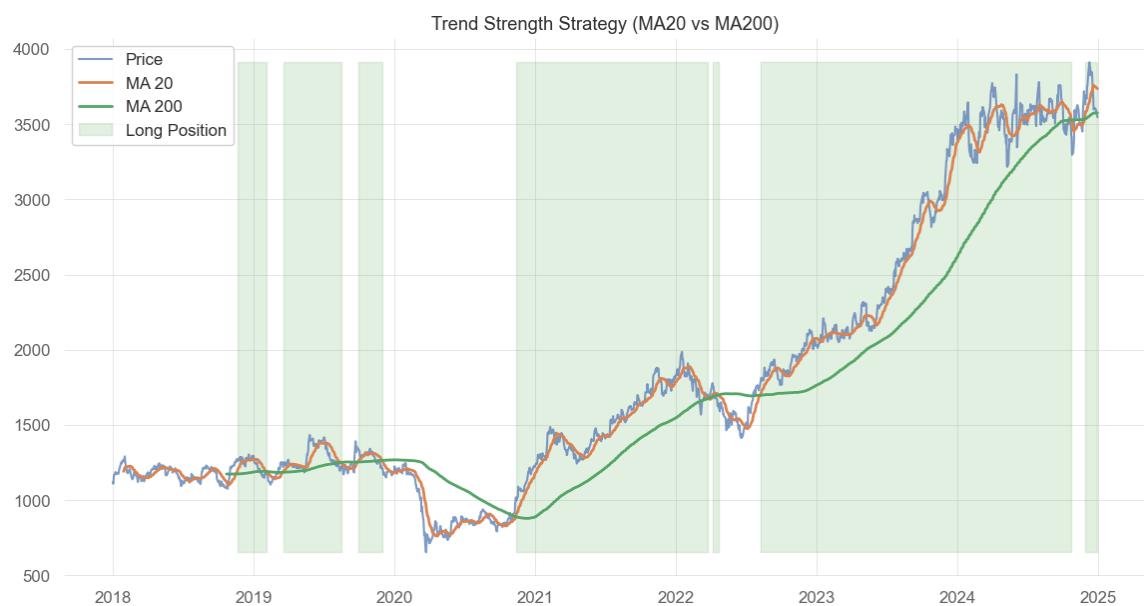
df["Strategy_Return"] = df["Position"] * df["Asset_Return"]
```

```
In [15]: df["Cum_Asset_Return"] = (1 + df["Asset_Return"]).cumprod()
df["Cum_Strategy_Return"] = (1 + df["Strategy_Return"]).cumprod()
```

```
In [16]: plt.figure(figsize=(14,7))
plt.plot(df["Close"], label="Price", alpha=0.7)
plt.plot(df["MA20"], label="MA 20", linewidth=2)
plt.plot(df["MA200"], label="MA 200", linewidth=2)

plt.fill_between(
    df.index,
    df["Close"].min(),
    df["Close"].max(),
    where=df["Position"] == 1,
    color="green",
    alpha=0.1,
    label="Long Position"
)

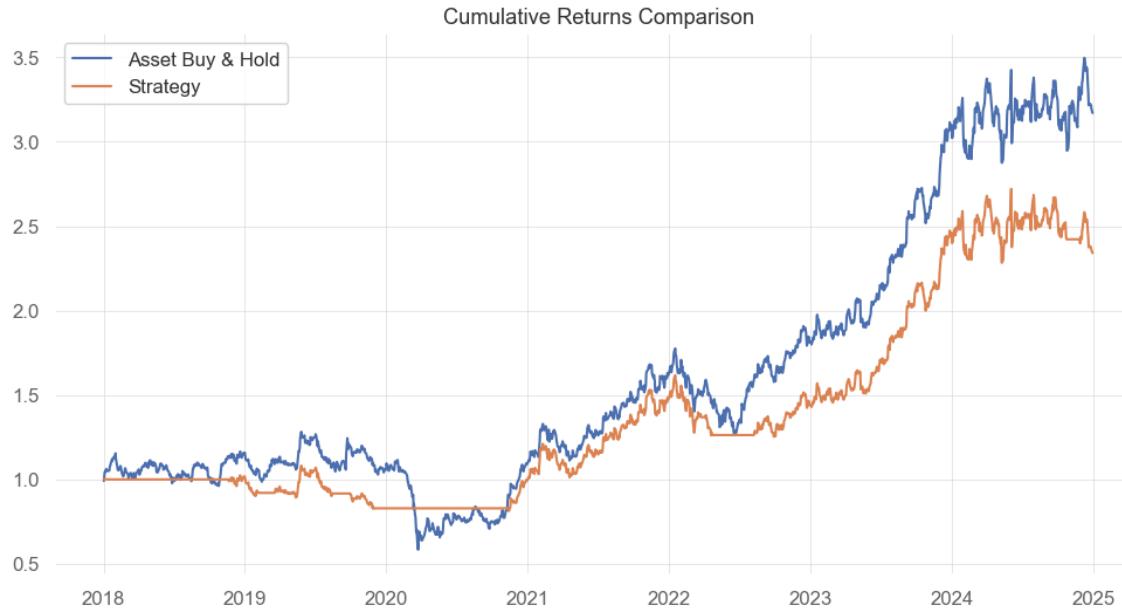
plt.title("Trend Strength Strategy (MA20 vs MA200)")
plt.legend()
plt.show()
```



```
In [18]: print("Cumulative Asset Return:", df["Cum_Asset_Return"].iloc[-1])
print("Cumulative Strategy Return:", df["Cum_Strategy_Return"].iloc[-1])
```

Cumulative Asset Return: 3.1703528978790216
 Cumulative Strategy Return: 2.341018064170992

```
In [19]: plt.figure(figsize=(12,6))
plt.plot(df["Cum_Asset_Return"], label="Asset Buy & Hold")
plt.plot(df["Cum_Strategy_Return"], label="Strategy")
plt.title("Cumulative Returns Comparison")
plt.legend()
plt.show()
```



```
In [20]: strategy_returns = df["Strategy_Return"].dropna()
asset_returns = df["Asset_Return"].dropna()
```

```
In [21]: qs.reports.full(strategy_returns)
```

Performance Metrics

	Strategy
Start Period	2018-01-02
End Period	2024-12-30
Risk-Free Rate	0.0%
Time in Market	65.0%
Cumulative Return	134.1%
CAGR %	13.23%
Sharpe	0.74
Prob. Sharpe Ratio	97.32%
Smart Sharpe	0.74
Sortino	1.1
Smart Sortino	1.09
Sortino/V2	0.78
Smart Sortino/V2	0.77

Assignment 15B: Trend + Momentum Filter (Stocks)

Assignment Rules

- Compute **50-period EMA**
- Compute **200-period EMA**

- Compute **RSI (14)**
- **Go Long** when all of the following conditions are satisfied:
 - 50 EMA is greater than 200 EMA
 - RSI (14) is greater than 40
- **Exit** when either conditions fails:
 - 50 EMA goes below 200EMA or RSI goes below 40
- Long-only strategy (no short positions)
- Avoid look-ahead bias

Output Requirements

- Plot **price chart, moving averages, RSI, and position**
- Print **cumulative asset returns** and **cumulative strategy returns**
- Plot **cumulative asset vs strategy returns**
- Use **QuantStats** to generate performance metrics

```
In [22]: import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
import quantstats as qs
import warnings
warnings.filterwarnings("ignore")
```

```
In [23]: start_date = "2018-01-01"
end_date = "2024-12-31"
df = yf.download("LT.NS", start_date, end_date)

df.index = pd.to_datetime(df.index)

[*****100%*****] 1 of 1 completed
```

```
In [24]: df["EMA50"] = df["Close"].ewm(span=50, adjust=False).mean()
df["EMA200"] = df["Close"].ewm(span=200, adjust=False).mean()
```

```
In [28]: delta = df["Close"].diff()

gain = delta.where(delta > 0, 0)
loss = -delta.where(delta < 0, 0)

avg_gain = gain.rolling(14).mean()
avg_loss = loss.rolling(14).mean()

rs = avg_gain / avg_loss
df["RSI"] = 100 - (100 / (1 + rs))
```

```
In [29]: df["Signal"] = 0

df.loc[
    (df["EMA50"] > df["EMA200"]) &
    (df["RSI"] > 40),
    "Signal"
] = 1
```

```
In [30]: df["Position"] = df["Signal"].shift(1)
df["Position"].fillna(0, inplace=True)
```

```
In [31]: df["Asset_Return"] = df["Close"].pct_change()
df["Strategy_Return"] = df["Position"] * df["Asset_Return"]
```

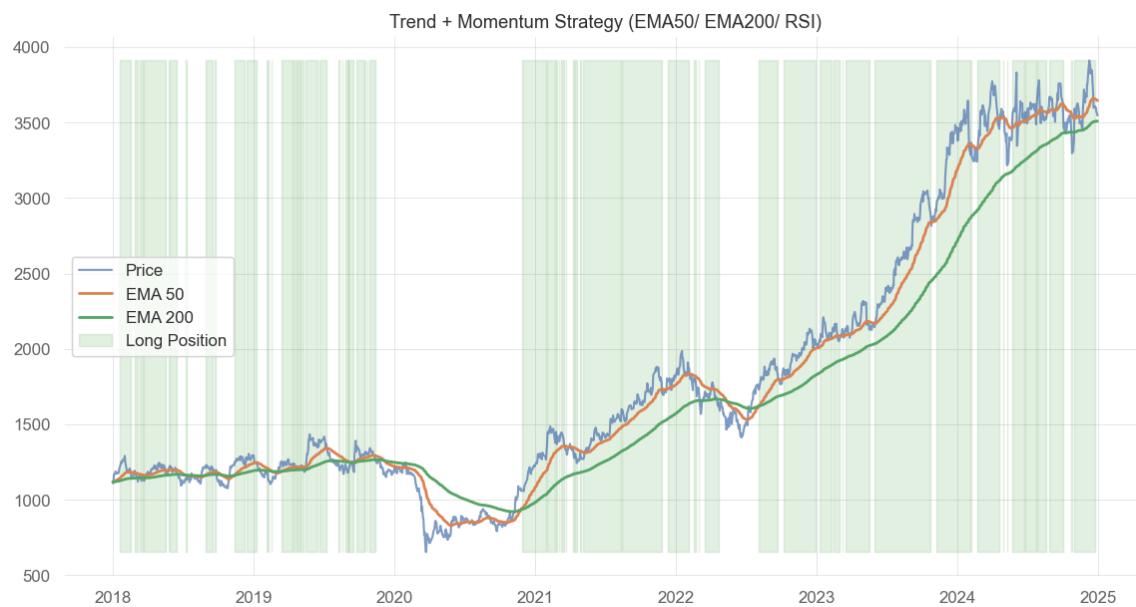
```
In [32]: df["Cum_Asset_Return"] = (1 + df["Asset_Return"]).cumprod()
df["Cum_Strategy_Return"] = (1 + df["Strategy_Return"]).cumprod()
```

```
In [33]: plt.figure(figsize=(14,7))

plt.plot(df["Close"], label="Price", alpha=0.7)
plt.plot(df["EMA50"], label="EMA 50", linewidth=2)
plt.plot(df["EMA200"], label="EMA 200", linewidth=2)

plt.fill_between(
    df.index,
    df["Close"].min(),
    df["Close"].max(),
    where=df["Position"] == 1,
    color="green",
    alpha=0.1,
    label="Long Position"
)

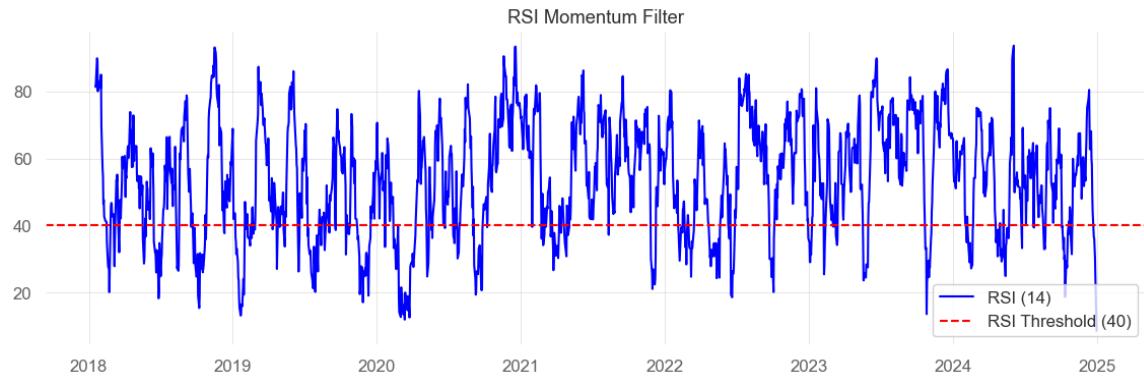
plt.title("Trend + Momentum Strategy (EMA50/ EMA200/ RSI)")
plt.legend()
plt.show()
```



```
In [36]: plt.figure(figsize=(14,4))
```

```
plt.plot(df["RSI"], label="RSI (14)", color="blue")
plt.axhline(40, color="red", linestyle="--", label="RSI Threshold (40)")

plt.title("RSI Momentum Filter")
plt.legend()
plt.show()
```



```
In [37]: print("Cumulative Asset Return:", df["Cum_Asset_Return"].iloc[-1])
print("Cumulative Strategy Return:", df["Cum_Strategy_Return"].iloc[-1])
```

```
Cumulative Asset Return: 3.1703528978790216
Cumulative Strategy Return: 1.4246762730544453
```

```
In [38]: plt.figure(figsize=(12,6))
```

```
plt.plot(df["Cum_Asset_Return"], label="Asset Buy & Hold")
plt.plot(df["Cum_Strategy_Return"], label="Strategy")
plt.title("Cumulative Asset vs Strategy Returns")
plt.legend()
plt.show()
```



```
In [ ]: strategy_returns = df["Strategy_Return"].dropna()
```

```
In [39]: qs.reports.full(strategy_returns)
```

Performance Metrics

	Strategy
Start Period	2018-01-02
End Period	2024-12-30
Risk-Free Rate	0.0%
Time in Market	65.0%
Cumulative Return	134.1%
CAGR %	13.23%
Sharpe	0.74
Prob. Sharpe Ratio	97.32%
Smart Sharpe	0.74
Sortino	1.1
Smart Sortino	1.09
Sortino/V2	0.78
Smart Sortino/V2	0.77

Assignment 15C: Momentum Strategy on NIFTY (Long / Short – Always in Market)

Assignment Rules

- Use **NIFTY index data**
- Compute **MACD**
- **Go Long (+1)** when **MACD > Signal Line**
- **Go Short (-1)** when **MACD < Signal Line**
- Positions **flip between Long and Short** on signal crossover
- The strategy **remains always in the market** (no flat state)
- Avoid look-ahead bias

Output Requirements

- Plot **price chart, MACD, and position**
- Print **cumulative asset returns** and **cumulative strategy returns**
- Plot **cumulative asset vs strategy returns**
- Use **QuantStats** to generate performance metrics

```
In [40]: import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
import quantstats as qs
import warnings
warnings.filterwarnings("ignore")
```

```
In [42]: start_date = "2020-01-01"
end_date = "2024-12-31"
df = yf.download("^NSEI", start_date, end_date)

df.index = pd.to_datetime(df.index)

[*****100%*****] 1 of 1 completed
```

```
In [44]: ema_12 = df["Close"].ewm(span=12, adjust=False).mean()
ema_26 = df["Close"].ewm(span=26, adjust=False).mean()

df["MACD"] = ema_12 - ema_26
df["Signal_Line"] = df["MACD"].ewm(span=9, adjust=False).mean()
```

```
In [45]: df["Signal"] = 0

df.loc[df["MACD"] > df["Signal_Line"], "Signal"] = 1
df.loc[df["MACD"] < df["Signal_Line"], "Signal"] = -1
```

```
In [46]: df["Position"] = df["Signal"].shift(1)
df["Position"].fillna(0, inplace=True)
```

```
In [47]: df["Asset_Return"] = df["Close"].pct_change()

df["Strategy_Return"] = df["Position"] * df["Asset_Return"]
```

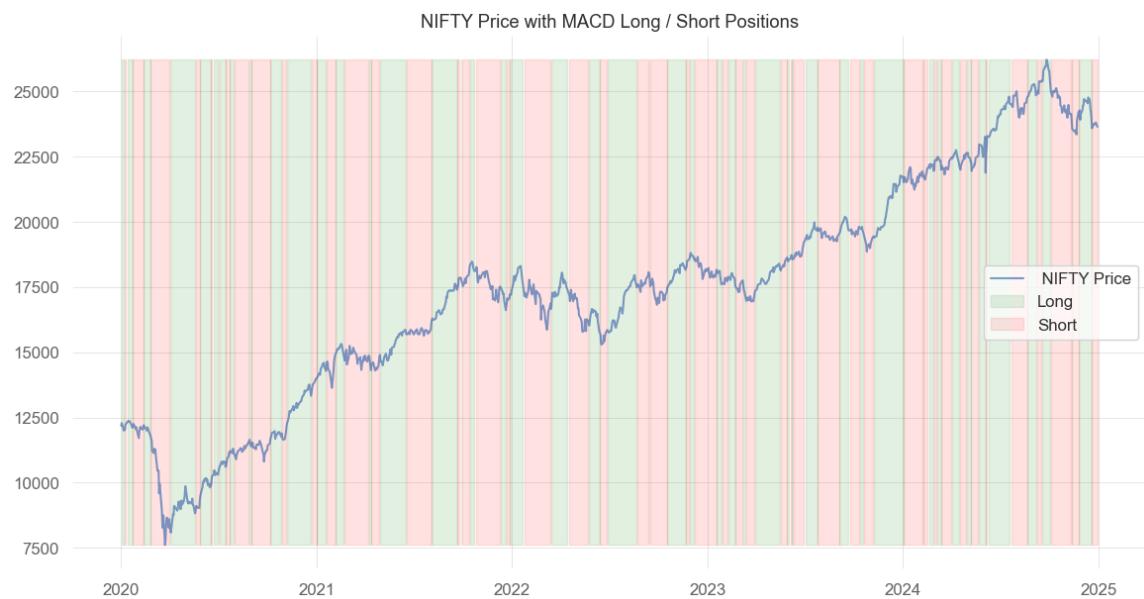
```
In [48]: df["Cum_Asset_Return"] = (1 + df["Asset_Return"]).cumprod()
df["Cum_Strategy_Return"] = (1 + df["Strategy_Return"]).cumprod()
```

```
In [49]: plt.figure(figsize=(14,7))

plt.plot(df["Close"], label=" NIFTY Price", alpha=0.7)

plt.fill_between(
    df.index,
    df["Close"].min(),
    df["Close"].max(),
    where=df["Position"] == 1,
    color="green",
    alpha=0.1,
    label="Long"
)

plt.fill_between(
    df.index,
    df["Close"].min(),
    df["Close"].max(),
    where=df["Position"] == -1,
    color="red",
    alpha=0.1,
    label="Short"
)
plt.title("NIFTY Price with MACD Long / Short Positions")
plt.legend()
plt.show()
```

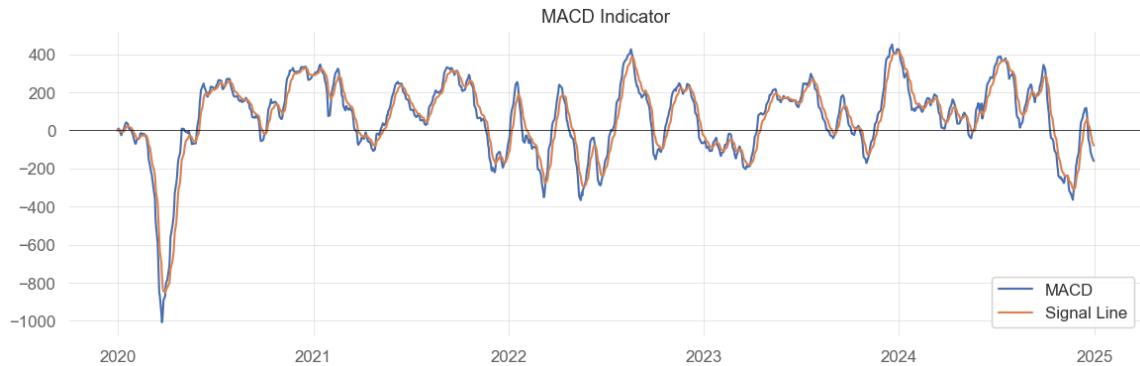


```
In [50]: plt.figure(figsize=(14,4))

plt.plot(df[ "MACD"], label="MACD")
plt.plot(df[ "Signal_Line"], label= "Signal Line")

plt.axhline(0, color="black", linewidth=0.5)

plt.title("MACD Indicator")
plt.legend()
plt.show()
```



```
In [51]: print("Cumulative Asset Return:", df[ "Cum_Asset_Return"].iloc[-1])
print("Cumulative Strategy Return:", df[ "Cum_Strategy_Return"].iloc[-1])
```

Cumulative Asset Return: 1.9408906538579878
Cumulative Strategy Return: 1.0519748941278482

```
In [52]: plt.figure(figsize=(12,6))
plt.plot(df[ "Cum_Asset_Return"], label=" Buy & Hold NIFTY")
plt.plot(df[ "Cum_Strategy_Return"], label="MACD Strategy")
plt.title("Cumulative Asset vs Strategy Returns")
plt.legend()
plt.show()
```



```
In [53]: strategy_returns = df[ "Strategy_Return"].dropna()
```

```
In [54]: qs.reports.full(strategy_returns)
```

Performance Metrics

	Strategy
Start Period	2020-01-02
End Period	2024-12-30
Risk-Free Rate	0.0%
Time in Market	100.0%
Cumulative Return	5.2%
CAGR %	1.04%
Sharpe	0.15
Prob. Sharpe Ratio	62.97%
Smart Sharpe	0.14
Sortino	0.23
Smart Sortino	0.22
Sortino/V2	0.16
Smart Sortino/V2	0.15

Assignment 15D : Price Action Confirmation (Stocks)

Assignment Rules

- Download **Last 10 Years of Nifty Daily Close Data**
- Identify **Bullish Engulfing Candle**
- Candle must form **above a 200 MA**
- Filter out the results

```
In [3]: import pandas as pd
import yfinance as yf
import numpy as np
import warnings
warnings.filterwarnings("ignore")
```

```
In [4]: df = yf.download("^NSEI", period="10y")
```

```
[*****100%*****] 1 of 1 completed
```

```
In [5]: if isinstance(df.columns, pd.MultiIndex):
    df.columns = df.columns.get_level_values(0)

df = df[['Open', 'High', 'Low', 'Close']].copy()

df['Close'] = df['Close'].astype(float)
```

```
In [6]: df['MA200'] = df['Close'].rolling(200).mean()
```

```
In [7]: df['Bullish_Engulfing'] = (
    (df['Close'].shift(1) < df['Open'].shift(1)) &
    (df['Close'] > df['Open']) &
    (df['Close'] > df['MA200'])
)
```

```
In [8]: signals = df[df['Bullish_Engulfing']].dropna()

signals.tail()
```

Out[8]:

Date	Price	Open	High	Low	Close	MA200	Bullish_Engulfing
2025-12-18	25764.699219	25902.349609	25726.300781	25815.550781	24784.197051	24784.197051	True
2025-12-31	25971.050781	26187.949219	25969.000000	26129.599609	24932.919043	24932.919043	True
2026-01-02	26155.099609	26340.000000	26118.400391	26328.550781	24970.452549	24970.452549	True
2026-01-12	25669.050781	25813.150391	25473.400391	25790.250000	25064.108799	25064.108799	True
2026-01-14	25648.550781	25791.750000	25603.949219	25665.599609	25084.463301	25084.463301	True

Assignment 15E: Price Action Confirmation – Bullish (Stocks)

Assignment Rules

- Download **Last 10 Years of Bank Nifty Daily Close Data**
- Identify a **Bullish Hammer Candle**
- Candle must form **above the 200 period EMA**
- Filter out the results

```
In [14]: import yfinance as yf
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings("ignore")
```

```
In [15]: df = yf.download("^NSEBANK", period="10y")

if isinstance(df.columns, pd.MultiIndex):
    df.columns = df.columns.get_level_values(0)

df = df[['Open', 'High', 'Low', 'Close']].copy()
```

[*****100%*****] 1 of 1 completed

```
In [16]: df['EMA200'] = df['Close'].ewm(span=200, adjust=False).mean()
```

```
In [17]: body = abs(df['Close'] - df['Open'])
lower_wick = df[['Open', 'Close']].min(axis=1) - df['Low']
upper_wick = df['High'] - df[['Open', 'Close']].max(axis=1)

df['Bullish_Hammer'] = (
    (lower_wick >= 2 * body) &
    (upper_wick <= body) &
    (df['Close'] > df['EMA200'])
)
```

```
In [18]: signals = df[df['Bullish_Hammer']].dropna()
signals.tail(10)
```

Date	Price	Open	High	Low	Close	EMA200	Bullish_Hammer
2024-09-10	51328.300781	51366.000000	50958.250000	51272.300781	48712.520862		True
2024-10-15	51975.949219	52022.050781	51698.750000	51906.000000	49489.958184		True
2024-11-21	50625.000000	50652.148438	49787.101562	50372.898438	49901.662906		True
2025-02-06	50468.351562	50553.351562	50149.800781	50382.101562	50201.247038		True
2025-05-22	54875.898438	54996.101562	54576.601562	54941.300781	51132.823423		True
2025-07-01	57375.800781	57533.851562	57150.351562	57459.449219	52365.557341		True
2025-09-05	54308.050781	54308.050781	53719.550781	54114.550781	53590.279068		True
2025-09-18	55797.101562	55835.250000	55490.898438	55727.449219	53700.028100		True
2025-11-11	57962.300781	58187.351562	57594.250000	58138.148438	54638.988659		True
2026-01-07	60039.699219	60065.398438	59760.648438	59990.851562	56136.437145		True

Assignment 15F: Price Action Confirmation – Bearish (Stocks)

Assignment Rules

- Identify a **Bearish Engulfing Candle**
- Candle must form **below the 200 EMA**
- Filter out the results

```
In [19]: import yfinance as yf
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings("ignore")
```

```
In [20]: df = yf.download("^NSEI", period="10y")
[*****100%*****] 1 of 1 completed
```

```
In [21]: if isinstance(df.columns, pd.MultiIndex):
    df.columns = df.columns.get_level_values(0)

df = df[['Open', 'High', 'Low', 'Close']].copy()
```

```
In [22]: df['EMA200'] = df['Close'].ewm(span=200, adjust=False).mean()
```

```
In [24]: df['Bearish_Engulfing'] = (
    (df['Close'].shift(1) > df['Open'].shift(1)) &
    (df['Close'] < df['Open']) &
    (df['Open'] >= df['Close'].shift(1)) &
    (df['Close'] <= df['Open'].shift(1)) &
    (df['Close'] < df['EMA200'])
)
```

```
In [25]: signals = df[df['Bearish_Engulfing']].dropna()
signals.tail(10)
```

	Price	Open	High	Low	Close	EMA200	Bearish_Engulfing
Date							
2019-08-13	11139.400391	11145.900391	10901.599609	10925.849609	11262.560664		T
2019-09-12	11058.299805	11081.750000	10964.950195	10982.799805	11210.402355		T
2020-06-24	10529.250000	10553.150391	10281.950195	10305.299805	10520.665480		T
2020-06-30	10382.599609	10401.049805	10267.349609	10302.099609	10512.858380		T
2022-03-03	16723.199219	16768.949219	16442.949219	16498.050781	16716.102052		T
2022-05-04	17096.599609	17132.849609	16623.949219	16677.599609	16864.691715		T
2022-06-01	16594.400391	16649.199219	16438.849609	16522.750000	16749.063884		T
2023-03-28	17031.750000	17061.750000	16913.750000	16951.699219	17521.226426		T
2024-12-20	23960.699219	24065.800781	23537.349609	23587.500000	23682.729949		T
2025-01-21	23421.650391	23426.300781	22976.849609	23024.650391	23655.380426		T

Assignment 15G: Buy and Hold (Multi-Asset Portfolio)

Strategy Rules

- Build a buy-and-hold portfolio using the following Yahoo Finance tickers (Last 10 yrs.):
GLD (Gold ETF), **SLV** (Silver ETF), **^NSEI** (NIFTY 50 Index), **QQQ** (NASDAQ 100 ETF), **SPY** (S&P 500 ETF)
- Enter all positions at the start of the test period
- Hold positions continuously with **no exits**
- Portfolio returns are driven purely by **long-term price movement**
- The strategy remains **fully invested** at all times

Evaluate Buy and Hold Portfolio Performance

- Plot the **cumulative portfolio equity curve**
- Evaluate **CAGR**, **volatility**, and **maximum drawdown**

```
In [72]: import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
```

```
In [78]: start_date = "2016-01-01"
end_date = "2025-12-31"

tickers = ["GLD", "SLV", "^NSEI", "QQQ", "SPY"]

data = yf.download(tickers, start_date, end_date)[ "Close" ]

data.index = pd.to_datetime(data.index)
```

```
[*****100*****] 5 of 5 completed
```

```
In [79]: returns = data.pct_change().dropna()
```

```
In [80]: num_assets = len(tickers)
weights = [1 / num_assets] * num_assets

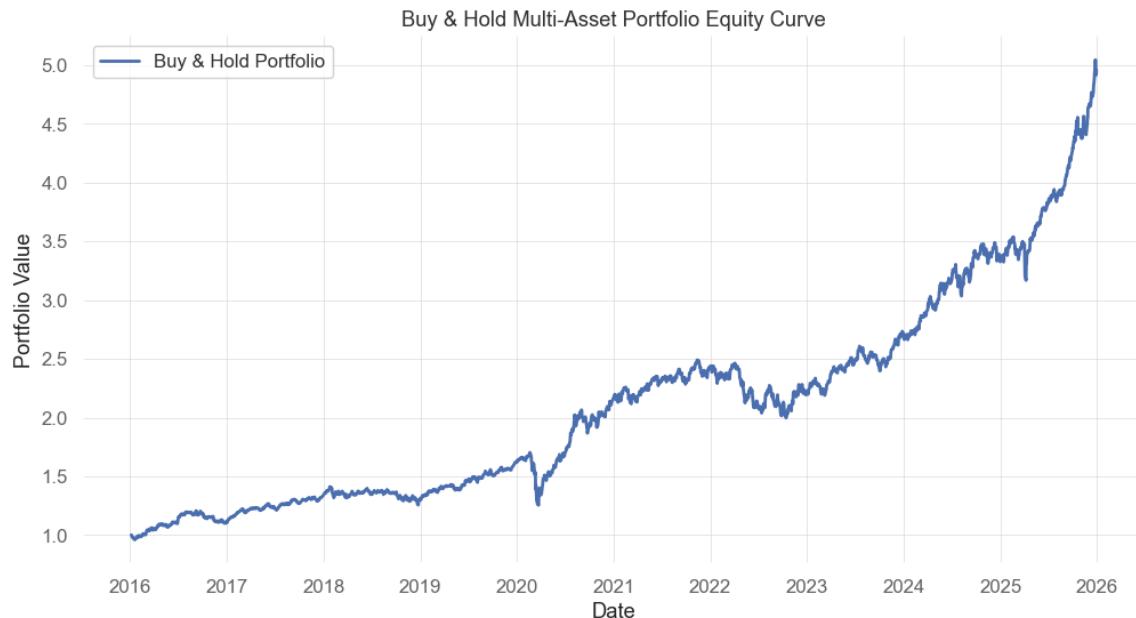
portfolio_returns = returns.dot(weights)
```

```
In [82]: cumulative_portfolio = (1 + portfolio_returns).cumprod()
```

```
In [84]: plt.figure(figsize=(12,6))

plt.plot(cumulative_portfolio, label="Buy & Hold Portfolio", linewidth=2)

plt.title("Buy & Hold Multi-Asset Portfolio Equity Curve")
plt.xlabel("Date")
plt.ylabel("Portfolio Value")
plt.legend()
plt.show()
```



```
In [85]: years = (cumulative_portfolio.index[-1] - cumulative_portfolio.index[0]).days / 365
cagr = (cumulative_portfolio.iloc[-1] ** (1 / years)) - 1
```

```
In [86]: volatility = portfolio_returns.std() * (252 ** 0.5)
```

```
In [87]: rolling_max = cumulative_portfolio.cummax()
drawdown = cumulative_portfolio / rolling_max - 1
max_drawdown = drawdown.min()
```

```
In [88]: print(f"CAGR: {cagr:.2%}")
print(f"Annualized Volatility: {volatility:.2%}")
print(f"Maximum Drawdown: {max_drawdown:.2%}")
```

```
CAGR: 17.38%
Annualized Volatility: 13.01%
Maximum Drawdown: -26.30%
```

```
In [ ]:
```

TTP_Foundation16

Portfolio & Risk Management

Concepts Covered

- Start with **single stock exposure** to understand concentrated risk
- Add **stock from the same sector** to measure risk and co-movement.
- Compute **Variance, Co-Variance** to calculate Portfolio Volatility and Correlation
- Measure risk using **volatility (standard deviation of returns)**
- Observe how **correlation between stocks** impacts portfolio risk
- Construct a **diversified portfolio** by combining low-correlated stocks
- Compare **total returns and risk** across:
 - Single stock
 - Two-stock portfolio
 - 5 Stock Low-correlation portfolio

```
In [1]: # Import necessary libraries
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import yfinance as yf
import pyfolio as pf
%matplotlib inline
```

```
In [2]: # Define stock universe (2 stocks per core sector)

tickers = ['DRREDDY.NS', 'SUNPHARMA.NS', 'BPCL.NS', 'IOC.NS', 'BHARTIARTL.NS', 'INDUSTOWER.NS']
```

```
In [3]: import yfinance as yf

data = yf.download(tickers,start='2022-01-01',end='2024-12-31')['Close']

[*****100%*****] 6 of 6 completed
```

```
In [4]: data.head(2)
```

```
Out[4]:    Ticker BHARTIARTL.NS BPCL.NS DRREDDY.NS INDUSTOWER.NS IOC.NS SUNPHARMA.NS
           Date
2022-01-03   675.648865 162.101013  944.917236   235.268509  59.208775   815.422424
2022-01-04   681.659607 162.038010  941.470947   241.224655  59.785400   804.616760
```

```
In [5]: df = data.copy()
```

```
In [6]: df.columns = df.columns.str.replace('.NS', '', regex=False)
```

```
In [7]: df.head()
```

```
Out[7]:    Ticker BHARTIARTL BPCL DRREDDY INDUSTOWER IOC SUNPHARMA
           Date
2022-01-03   675.648865 162.101013  944.917236   235.268509  59.208775   815.422424
2022-01-04   681.659607 162.038010  941.470947   241.224655  59.785400   804.616760
2022-01-05   684.151917 165.711639  932.699585   242.406433  61.462849   802.119385
2022-01-06   694.316406 166.173462  922.779541   247.606262  61.462849   796.836670
2022-01-07   688.745544 166.341400  916.734070   246.944489  62.118099   796.212402
```

```
In [8]: returns_df = df.pct_change().dropna()
```

```
In [9]: comparison = pd.DataFrame({
    'annual_mean_returns (%)': returns_df.mean() * 252 * 100,
    'annual_volatility (%)': returns_df.std() * np.sqrt(252) * 100
}).round(3)
```

```
In [10]: comparison.head(6)
```

```
Out[10]: annual_mean_returns (%)  annual_volatility (%)
```

Ticker	annual_mean_returns (%)	annual_volatility (%)
BHARTIARTL	31.243	21.542
BPCL	22.638	29.102
DRREDDY	14.888	21.121
INDUSTOWER	20.043	38.326
IOC	30.405	28.515
SUNPHARMA	30.191	19.350

How much risk we take by holding only ONE stock?"

```
In [11]: single_stock = df[['DRREDDY']]
```

```
In [12]: single_returns = single_stock.pct_change().dropna()
```

```
In [13]: import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(10,4))
plt.plot(single_stock.index, single_stock['DRREDDY'])
plt.title('DRREDDY Price')
plt.xlabel('Date')
plt.ylabel('Price')
plt.grid(True)
plt.show()
```



```
In [14]: daily_volatility = float(single_returns.std())
annual_volatility = daily_volatility * np.sqrt(252)

print("Daily Volatility : ", round(daily_volatility, 4))
print("Annual Volatility: ", round(annual_volatility, 4))
```

```
Daily Volatility : 0.0133
Annual Volatility: 0.2112
```

Portfolio Creation (Modern Portfolio Theory)

Expected Portfolio Returns Calculation :
The portfolio return is simply the weighted average of individual returns.

$$\text{Portfolio Returns} = (w_A * R_A) + (w_B * R_B)$$

Where:

w_A = Weight in stock A

R_A = Returns of stock A

w_B = Weight in stock B

R_B = Returns of stock B

$$\text{Portfolio Returns} = (w_A * R_A) + (w_B * R_B) + (w_C * R_C) + \dots + (w_N * R_N)$$

Add one more stock from the same sector and create a Portfolio

```
In [15]: two_stocks = df[['DRREDDY', 'SUNPHARMA']]  
two_returns = two_stocks.pct_change().dropna()
```

```
In [16]: weights = np.array([0.5, 0.5])  
  
annual_mean_returns = two_returns.mean() * 252  
  
# Uses portfolio weights and annual mean returns  
# Computes the weighted average return  
# np.dot() multiplies and sums weight-return pairs  
# Produces one number: expected annual portfolio return  
expected_portfolio_returns_annual = np.dot(weights, annual_mean_returns)  
  
print(  
    f"Expected annual return of a 50-50 portfolio of DRREDDY and SUNPHARMA is "  
    f"{expected_portfolio_returns_annual * 100:.3f}%")
```

Expected annual return of a 50-50 portfolio of DRREDDY and SUNPHARMA is 22.540%

Expected Portfolio Volatility Calculation

$$\text{Variance} = \text{Var}(x) = \frac{\sum(x_i - \bar{x})^2}{n - 1}$$

$$\text{Covariance} = \text{Cov}(x, y) = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{n - 1}$$

Unlike portfolio returns, we cannot simply take the weighted average of individual stock variances. We would also have to consider the relation between assets while calculating a portfolio variance. Hence, the generalized formula turns out to be:

$$\text{Portfolio Variance} = w_A^2 \times \sigma^2(R_A) + w_B^2 \times \sigma^2(R_B) + 2 \times w_A \times w_B \times \text{cov}(R_A, R_B)$$

For Portfolio Volatility we take the square root of the portfolio variance to get portfolio standard deviation (volatility):

$$\text{Portfolio Std Dev} = \sqrt{w_A^2 \times \sigma^2(R_A) + w_B^2 \times \sigma^2(R_B) + 2 \times w_A \times w_B \times \text{cov}(R_A, R_B)}$$

Where:-

w_A = Weight in stock A

$\sigma^2(R_A)$ = Variance of returns of stock A

w_B = Weight in stock B

$\sigma^2(R_B)$ = Variance of returns of stock B

$\text{cov}(R_A, R_B)$ = Covariance between stock A returns and stock B returns

To calculate the expected volatility of a portfolio with DRREDDY and SUNPHARMA, we use the following formula:

Substituting:

- DRREDDY with stock_1
- SUNPHARMA with stock_2

$$\text{Portfolio Std Dev} = \sqrt{w_{\text{DRREDDY}}^2 \times \sigma^2(R_{\text{DRREDDY}}) + w_{\text{SUNPHARMA}}^2 \times \sigma^2(R_{\text{SUNPHARMA}}) + 2 \times w_{\text{DRREDDY}} \times w_{\text{SUNPHARMA}} \times \text{cov}(R_{\text{DRREDDY}}, R_{\text{SUNPHARMA}})}$$

Calculating Expected Portfolio Volatility using the First Principles

```
In [17]: # Calculate annual variances of both stocks
drreddy_annual_variance = two_returns['DRREDDY'].var() * 252
sunpharma_annual_variance = two_returns['SUNPHARMA'].var() * 252

# Print annual variances
print(f"Annual variance of DRREDDY is {drreddy_annual_variance:.4f}")
print(f"Annual variance of SUNPHARMA is {sunpharma_annual_variance:.4f}")
```

Annual variance of DRREDDY is 0.0446
Annual variance of SUNPHARMA is 0.0374

```
In [36]: daily_risk_table = two_returns.cov()
annual_risk_table = daily_risk_table * 252
```

```
In [19]: print("Daily Variance-Covariance Table")
print(daily_risk_table)

print("\nAnnual Variance-Covariance Table")
print(annual_risk_table)
```

Daily Variance-Covariance Table
Ticker DRREDDY SUNPHARMA
Ticker
DRREDDY 0.000177 0.000060
SUNPHARMA 0.000060 0.000149

Annual Variance-Covariance Table
Ticker DRREDDY SUNPHARMA
Ticker
DRREDDY 0.044611 0.015196
SUNPHARMA 0.015196 0.037441

```
In [20]: # Extract covariance between DRREDDY and SUNPHARMA
drreddy_sunpharma_covar = annual_risk_table.loc['DRREDDY', 'SUNPHARMA']

print(
    f"The covariance between DRREDDY and SUNPHARMA is "
    f"{drreddy_sunpharma_covar:.6f}")
```

The covariance between DRREDDY and SUNPHARMA is 0.015196

```
In [21]: # Define weights
w_drreddy = w_sunpharma = 0.5

# Calculate annual portfolio variance (first principles)
expected_portfolio_variance_annual =
    (w_drreddy ** 2) * drreddy_annual_variance +
    (w_sunpharma ** 2) * sunpharma_annual_variance +
    (2 * w_drreddy * w_sunpharma * drreddy_sunpharma_covar)

# Calculate annual portfolio volatility
expected_portfolio_vol_annual = np.sqrt(expected_portfolio_variance_annual)

print(
    f"Expected annual volatility of a 50-50 portfolio of DRREDDY and SUNPHARMA is "
    f"{expected_portfolio_vol_annual * 100:.4f}%" )
```

Expected annual volatility of a 50-50 portfolio of DRREDDY and SUNPHARMA is 16.7663%

Note:

As the number of assets increases, the number of covariance terms grows rapidly.

Matrix notation allows us to compute portfolio risk **cleanly and consistently**, regardless of the number of stocks in a Portfolio

$$\text{Portfolio Std Dev} = \sqrt{w_{\text{DRREDDY}}^2 \sigma^2(R_{\text{DRREDDY}}) + w_{\text{SUNPHARMA}}^2 \sigma^2(R_{\text{SUNPHARMA}}) + w_{\text{CIPLA}}^2 \sigma^2(R_{\text{CIPLA}}) + 2 w_{\text{DRREDDY}} w_{\text{SUNPHARMA}} cov(R_{\text{DRREDDY}}, R_{\text{SUNPHARMA}}) + 2 w_{\text{DRREDDY}} w_{\text{CIPLA}} cov(R_{\text{DRREDDY}}, R_{\text{CIPLA}}) + 2 w_{\text{SUNPHARMA}} w_{\text{CIPLA}} cov(R_{\text{SUNPHARMA}}, R_{\text{CIPLA}})}$$

Calculating Expected Portfolio Volatility using Matrices

```
In [22]: daily_var_cov_matrix = two_returns.cov()
annual_var_cov_matrix = daily_var_cov_matrix * 252
```

```
In [23]: # Suppress scientific notation in Pandas
pd.set_option('display.float_format', '{:.4f}'.format)

# Display annual variance-covariance matrix
annual_var_cov_matrix
```

```
Out[23]:
```

Ticker	DRREDDY	SUNPHARMA
DRREDDY	0.0446	0.0152
SUNPHARMA	0.0152	0.0374

```
In [24]: # Extract covariance between DRREDDY and SUNPHARMA
drreddy_sunpharma_covariance = annual_var_cov_matrix.loc['DRREDDY', 'SUNPHARMA']

print(
    f"The covariance between DRREDDY and SUNPHARMA is "
    f"{drreddy_sunpharma_covariance:.6f}")
```

The covariance between DRREDDY and SUNPHARMA is 0.015196

```
In [25]: # Define weight vector (50-50 portfolio)
weights = np.array([0.5, 0.5])

# Uses portfolio weights and the annual variance-covariance matrix
# Combines all individual variances and covariances correctly
# Performs matrix multiplication using @
# Produces one number: annual portfolio variance
# Square root of this gives portfolio volatility
expected_portfolio_variance_annual = (weights.T @ annual_var_cov_matrix.values @ weights)

# Calculate annual portfolio volatility
expected_portfolio_vol_annual = np.sqrt(expected_portfolio_variance_annual)

print(
    f"Expected annual portfolio volatility of a 50-50 portfolio of "
    f"DRREDDY and SUNPHARMA is {expected_portfolio_vol_annual * 100:.4f}%)
```

Expected annual portfolio volatility of a 50-50 portfolio of DRREDDY and SUNPHARMA is 16.7663%

```
In [26]: comparison = pd.DataFrame(
    {
        'Annual Mean Return (%)': [
            annual_mean_returns['DRREDDY'] * 100,
            annual_mean_returns['SUNPHARMA'] * 100,
            expected_portfolio_returns_annual * 100
        ],
        'Annual Volatility (%)': [
            np.sqrt(annual_var_cov_matrix.loc['DRREDDY', 'DRREDDY']) * 100,
            np.sqrt(annual_var_cov_matrix.loc['SUNPHARMA', 'SUNPHARMA']) * 100,
            expected_portfolio_vol_annual * 100
        ]
    },
    index=['DRREDDY', 'SUNPHARMA', 'DRREDDY_SUNPHARMA']
)

comparison
```

```
Out[26]:
```

	Annual Mean Return (%)	Annual Volatility (%)
DRREDDY	14.8876	21.1214
SUNPHARMA	30.1914	19.3497
DRREDDY_SUNPHARMA	22.5395	16.7663

Investing in 2 stocks reduced our risk

- Returns average out
- Risk reduces because assets don't move perfectly together

Correlation Calculation

$$\text{Correlation} = r = \frac{\text{Cov}(x, y)}{S_x, S_y}$$

```
In [27]: # Correlation between DRREDDY and SUNPHARMA
corr = two_returns['DRREDDY'].corr(two_returns['SUNPHARMA'])

print(f"Correlation between DRREDDY and SUNPHARMA is {corr:.3f}")
```

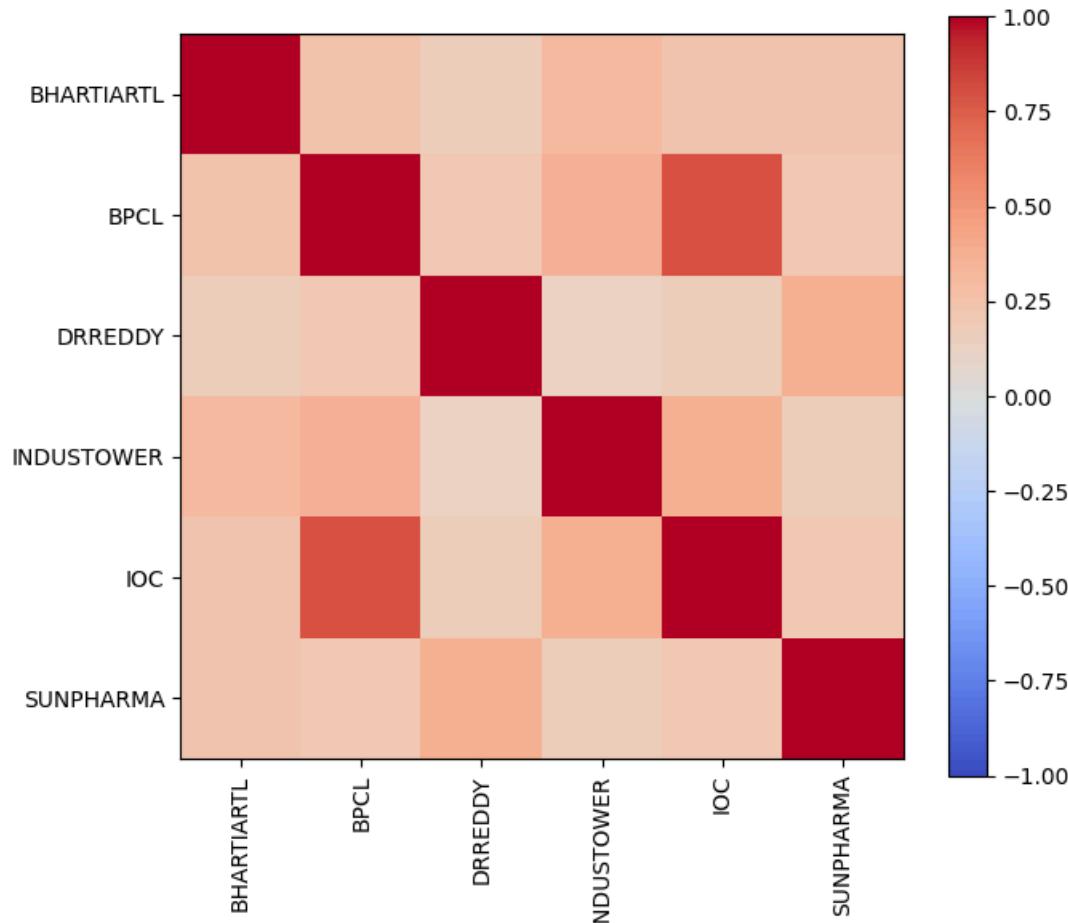
Correlation between DRREDDY and SUNPHARMA is 0.372

Using Correlation Values To Build Portfolio of 5 Stocks (Universe : Nifty50)

```
In [28]: import matplotlib.pyplot as plt

# Calculating Correlation matrix for all stocks
corr_matrix = returns_df.corr()

plt.figure(figsize=(7, 6))
plt.imshow(returns_df.corr(), cmap='coolwarm', vmin=-1, vmax=1)
plt.colorbar()
plt.xticks(range(len(returns_df.columns)), returns_df.columns, rotation=90)
plt.yticks(range(len(returns_df.columns)), returns_df.columns)
plt.tight_layout()
plt.show()
```



```
In [29]: corr_matrix.round(2)
```

Ticker	BHARTIARTL	BPCL	DRREDDY	INDUSTOWER	IOC	SUNPHARMA
Ticker						
BHARTIARTL	1.0000	0.2500	0.1700	0.3000	0.2200	0.2200
BPCL	0.2500	1.0000	0.1900	0.3600	0.7900	0.2100
DRREDDY	0.1700	0.1900	1.0000	0.1300	0.1800	0.3700
INDUSTOWER	0.3000	0.3600	0.1300	1.0000	0.3700	0.1700
IOC	0.2200	0.7900	0.1800	0.3700	1.0000	0.2000
SUNPHARMA	0.2200	0.2100	0.3700	0.1700	0.2000	1.0000

Now, lets build our portfolio with low correlation stocks

```
In [30]:
```

```
final_stocks = [
    'DRREDDY',      # Pharma
    'INDUSTOWER',  # Telocom
    'IOC',          # Energy
    'BHARTIARTL',  # Telecom
    'SUNPHARMA'    # Pharma
]
```

```
In [31]:
```

```
final_returns = df[final_stocks].pct_change().dropna()
weights = np.ones(len(final_stocks)) / len(final_stocks)

# Multiplies each stock's daily return by its weight
# Adds them across stocks
# Result → one daily return series for the portfolio
final_portfolio_returns = final_returns.dot(weights)

# Annual risk (volatility)
final_risk = final_portfolio_returns.std() * np.sqrt(252)

# Annual mean return
final_annual_return = final_portfolio_returns.mean() * 252

print("Final Portfolio Annual Return:", round(final_annual_return * 100, 2), "%")
print("Final Portfolio Risk:", round(final_risk * 100, 2), "%")
```

Final Portfolio Annual Return: 25.35 %

Final Portfolio Risk: 16.35 %

```
In [37]:
```

```
equity_df = pd.DataFrame(index=final_returns.index)

# 1 stock (pick one explicitly)
equity_df['1_Stock'] = final_returns['DRREDDY']

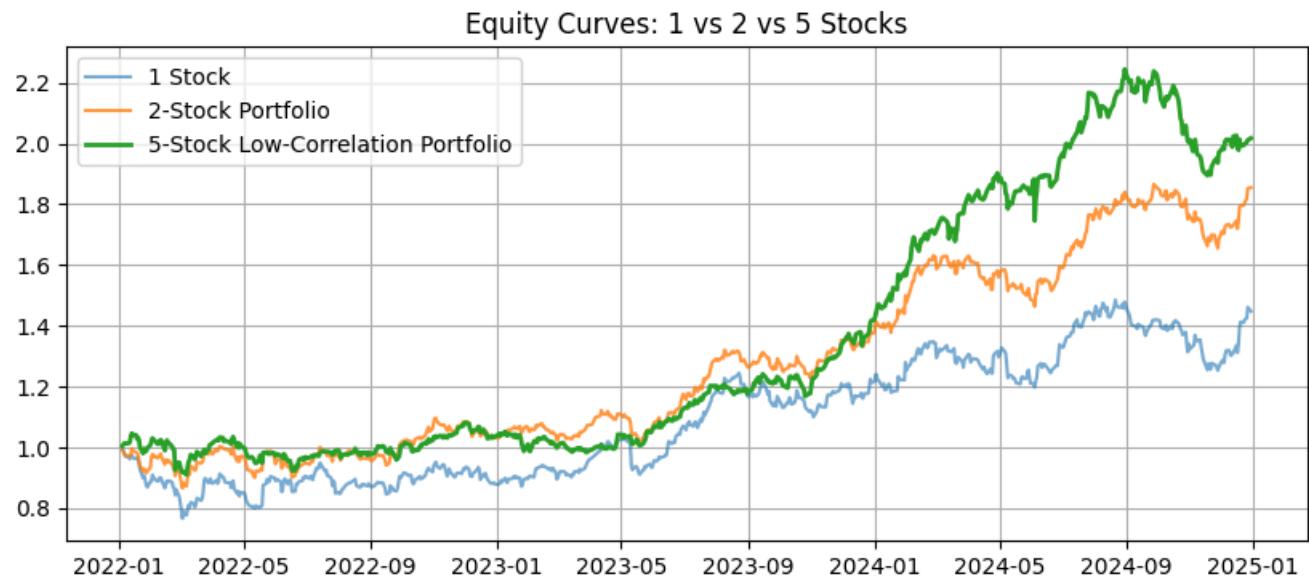
# 2 stocks (equal weight, explicit)
equity_df['2_Stock_Portfolio'] = (
    final_returns[['DRREDDY', 'SUNPHARMA']].mean(axis=1)
)

# 5 stocks (equal weight, explicit)
equity_df['5_Stock_Portfolio'] = final_returns.mean(axis=1)
```

```
In [38]:
```

```
equity_curves = (1 + equity_df).cumprod()
```

```
In [39]: plt.figure(figsize=(10, 4))
plt.plot(equity_curves['1_Stock'], label='1 Stock', alpha=0.6)
plt.plot(equity_curves['2_Stock_Portfolio'], label='2-Stock Portfolio', alpha=0.8)
plt.plot(equity_curves['5_Stock_Portfolio'], label='5-Stock Low-Correlation Portfolio', linewidth=2)
plt.legend()
plt.title('Equity Curves: 1 vs 2 vs 5 Stocks')
plt.grid(True)
plt.show()
```



```
In [35]: risk_return = pd.DataFrame(
    columns=['Annual Return (%)', 'Annual Risk (%)']
)

# 1 stock (example: DRREDDY)
one_stock_returns = final_returns['DRREDDY']

risk_return.loc['1 Stock (DRREDDY)'] = [
    one_stock_returns.mean() * 252 * 100,
    one_stock_returns.std() * np.sqrt(252) * 100
]

# 2 stocks (equal weight: DRREDDY + SUNPHARMA)
two_stock_returns = final_returns[['DRREDDY', 'SUNPHARMA']].mean(axis=1)

risk_return.loc['2 Stocks (DRREDDY + SUNPHARMA)'] = [
    two_stock_returns.mean() * 252 * 100,
    two_stock_returns.std() * np.sqrt(252) * 100
]

# 5 stocks (final portfolio)
risk_return.loc['5 Stock Correlation Based Portfolio'] = [
    final_portfolio_returns.mean() * 252 * 100,
    final_portfolio_returns.std() * np.sqrt(252) * 100
]

risk_return.round(2)
```

```
Out[35]:
```

	Annual Return (%)	Annual Risk (%)
1 Stock (DRREDDY)	14.8900	21.1200
2 Stocks (DRREDDY + SUNPHARMA)	22.5400	16.7700
5 Stock Correlation Based Portfolio	25.3500	16.3500

Diversification Based on Correlation - The Core Insight

Correlation determines whether diversification **works or fails**.

A well-diversified portfolio:

- Survives drawdowns, Reduces emotional stress
- Allows compounding to work uninterrupted
- You cannot control returns but You **can** control correlation
- And controlling correlation is how portfolios survive long enough to succeed.
- Diversification takes care of Unsystematic Risk and not Systematic Risk.