Python

Created	@November 15, 2024 7:26 PM
Tags	

What is Python ?

Introduction

Python is a general - purpose programming language created by Guido van Rossum in 1991. It is open-source, versatile, high-level programming language known for its ease of use and readability. It is widely employed in various fields, including web development, machine learning, and business analytics. For business analyst, Python provides powerful libraries and tools that allow them to analyse data efficiently, automate processes, and make informed decisions based on insights from data.

Definition

Python is a programming language (with syntax rules for writing what is considered valid Python code) and the Python interpreter software that reads source code (written in the Python language) and performs its instructions.

Python is both a programming language and an interpreter software. A programming language is a formal system that allows humans to write instructions for a computer to execute. These instructions, called source code, are written in a human-readable format following specific syntax rules defined by the programming language. In Python's case, these syntax rules dictate how to write code correctly so the interpreter can understand it.

The Python interpreter is a software tool that reads the source code written in Python, translates it into machine instructions, and then executes those instructions. In simpler terms, the interpreter "reads" the Python code and performs the tasks as directed.

- Key Concepts:

1. *Programming:* The process of designing and writing a set of instructions (source code) for a computer to carry out specific tasks.
2. *Source Code:* The human-readable instructions written in a programming language (like Python). For example, print("Hello, world!") is a line of Python source code.
3. *Indentation:* In Python, indentation (spaces or tabs at the beginning of a line) is used to define the structure of the code. Unlike many other languages, Python requires proper indentation to indicate which statements belong to a certain block of code (such as inside loops or conditionals). This improves readability and makes the structure of programs clear.
4. *Program:* A complete set of instructions (source code) that performs a specific task when executed by a computer. A program can be something simple, like displaying a message, or complex, like analysing data.

Features of Python

1. Python's concise, readable syntax allows analysts to quickly write and test code, speeding up development and enabling rapid responses to business needs. Its extensive libraries for data analysis and automation boost productivity by reducing the need to create code for routine tasks.
2. Python's simplicity and extensive libraries [collection of pre-written code that users can use to optimise their code] allow analysts to quickly create and iterate on models, enabling businesses to experiment with data efficiently.
3. Business analysts often work with large datasets stored in databases. Python’s seamless integration with databases like SQL, MySQL, PostgreSQL, and even NoSQL databases makes it easy to retrieve, process, and analyze this data.
4. Python’s libraries like Pandas, NumPy are tailored for complex scientific and numeric programming, which is essential in business analytics for statistical analysis. Moreover, Python’s visualization tools like Matplotlib, Seaborn, and Plotly allow analysts to present data visually.
5. Machine learning (ML) is one of the most powerful tools in business analytics, and Python excels in this area with libraries such as Scikit-learn. Machine learning helps businesses automate tasks, build predictive models, and uncover hidden patterns in data.

Python for Business Analytics

1. Data wrangling: With libraries like Pandas, analyst can clean, transform, and prepare data for analysis, handling missing values and outliers efficiently.
2. Statistical Analysis: Python performs statistical analysis like descriptive statistics and correlation matrix with using libraries like Pandas.
3. Visualisation: With tools like Matplotib and Seaborn, Python allows analyst to create charts and graphs to visualise trends, patterns, and distributions in data.
4. Machine learning: Python's Scikit-learn library offers a simple way to build machine learning models for predictive analytics, clustering, and classification problems.
5. Data manipulation: NumPy is essential for business analytics due to its efficient array operations, mathematical functions, and vectorization, allowing for fast data processing. It supports multidimensional data and integrates seamlessly with other libraries like Pandas. Additionally, NumPy's statistical analysis and random number generation capabilities enhance data manipulation and simulation tasks.

Basic of Python and its' Syntax

Understanding Python's syntax is crucial for writing code. Some basic include:

1. Variables: Used to store data values that can be changed.

```
x = 10

name = "Rhea"
```

2. Constants: Constants are similar to variables but their values do not change.

```
PI = 3.14159
```

3. Print function: The print() function is used to display output in Python. It can print text, variable values, and even expressions.

```
print("Hello, World!")
```

4. Strings: Strings are sequences of characters enclosed in quotes, and can be manipulated using various string methods. Python supports both single (' ') and double (" ") quotes.

```
greeting = "Hello, World!"
print (greeting)
```

5. Input: The input() function allows users to input data. The input is always returned as a string, and can be converted to other types if needed.

```
age = input("Enter your age:")
```

6. Basic arithmetic Operations: Python supports standard arithmetic operations like multiplication [*]; division [/]; subtraction [-] ; addition [+] ; Modules [%] ; Floor division [//] and Power [**].

```
# Give values to the variables for the operations
a = 5
b = 2
addition = a + b
multiplication = a*b
division = a/b
subtraction = a - b
Modulus = a % b
FloorDivision = a // b
Power =  a ** b
# Print the output for the operations
print (addition)
print (multiplication)
print(division)
print (subtraction)
print (Modulus)
print (FloorDivision)
print (Power)
#Output:
```

7

10

2.5

3

1

2

25

7. Comments: Comments in Python are used to add explanations or notes to the code. They are ignored during execution. Single-line comments begin with #.

This is a single line comment

8. Indentation: Python uses indentation [spaces] to define code blocks instead of curly braces.

if age > 20:

print("You are an adult")

Python data types

A data type is a classification of data which informs the compiler or interpreter about how the data will be used by the programmer.

1. Integer [int] - Whole numbers

Example: 0 , 3 , -3

2. Floating point [float] - Decimal numbers

Example: 1/2 , -0.3 , 3.0 , 5e10

3. String [str] - Text

Example: " Hello " , ' Hello '

4. Boolean [bool] - Logical values

Example: True , False

Note: To know the data type of an variable, use the type() function

print (2)

print (type(2))

Python operators

1. Comparison operators: to compare values and produce boolean output. The operators are as follows: Equal to [= =] ; Not equal to [! =] ; Less than [<] ; Greater than [>] ; Less than or equal to [< =] and Greater than or equal to [< =].

a = 10

b = 5

if a > b:

print ("a is greater than b") # This will execute true

else:

print("a is not greater than b") # This will give false

#Output:

a is greater then b

2. Logical operators: These are used to combine conditional statements. The three main logical operators in Python are:

and: Returns true if both conditions are true.

or: Returns true if at least one condition is true.

not: Reverses the result; returns true if the condition is false, and vice versa.

x = 5

y = 10

z = 15

and operator

if x < y and y < z:

print("Both conditions are true")

Output: Both conditions are true

or operator

if x > y or y < z:

```
print("At least one condition is true")

# Output: At least one condition is true

# not operator

if not (x > y):

print("x is not greater than y")

# Output: x is not greater than y
```

3. Chained comparison operators: Chained comparison operators allow you to compare multiple values in a single, concise expression. Python evaluates them as a sequence of comparisons. This enhances readability and efficiency compared to writing separate comparisons joined by logical operators.

```
a = 5

b = 10

c = 15

# Chained comparison using less than

if a < b < c:

print("a is less than b, and b is less than c")

# Output: a is less than b, and b is less than c

# Chained comparison using equality

if a <= b <= c:

print("a is less than or equal to b, and b is less than or equal to c")

# Output: a is less than or equal to b, and b is less than or equal to c
```

Conditional And Loop Statements

Conditional Statements (if, elif, else)

Conditional statements are used to perform decision-making in programming. Based on conditions (expressed using comparison operators), the program can execute different blocks of code.

1. if Statement

The if statement checks a condition and executes the indented block of code that follows it if the condition is True. If the condition is False, the code block is skipped.

Example:

a = 10

b = 5

if a > b:

print("a is larger than b")

Here, since a (10) is greater than b (5), the message "a is larger than b" will be printed.

2. elif (else if) Statement

The elif statement allows you to check multiple conditions. It follows the if or another elif statement and checks its own condition only if the previous conditions were False.

Example:

a = 10

b = 15

if a > b:

print("a is larger than b")

elif a < b:

print("a is smaller than b")

Here, since a is smaller than b, the program will print "a is smaller than b". The if condition is False, so the elifblock is checked.

3. else Statement

The else statement is used when all previous conditions are False. If none of the if or elif conditions are met, the else block will execute.

Example:

a = 10

b = 10

if a > b:

print("a is larger than b")

elif a < b:

print("a is smaller than b")

else:

print("a is equal to b")

Since a and b are equal, the message "a is equal to b" will be printed.

Loop Statements (while, for)

Loops are used to repeatedly execute a block of code based on a condition or a sequence.

1. while Loop

A while loop continues to execute as long as the condition is True. Once the condition becomes False, the loop stops.

Example:

i = 0

while i < 5:

print("Count is:", i)

i += 1

- The loop starts with i = 0. It checks if i < 5. Since the condition is True, it prints the value of i and increments iby 1.
- This process repeats until i becomes 5. When i reaches 5, the condition i < 5 becomes False, and the loop stops.

2. for Loop

The for loop is used when you know the number of iterations or when iterating over a sequence (like a list, tuple, or string). It iterates over the sequence, executing the code for each item.

Example:

for i in range(5):

print("Count is:", i)

- The range(5) function generates numbers from 0 to 4 (5 is not included). The loop runs for each value in that range, printing the value of i during each iteration.

3. Controlling Loop Execution with break and continue

- break: The break statement is used to exit a loop prematurely. It terminates the loop completely.
- continue: The continue statement skips the current iteration and continues with the next iteration of the loop.

Example with break:

for val in "Python":

if val == "h":

break # Exit the loop when 'h' is found

print(val)

- This loop iterates through the string "Python". When it encounters the character "h", the break statement stops the loop, and it prints:

P

y

t

Example with continue:

for val in "Python":

if val == "h":

continue # Skip printing 'h' and continue with the next iteration

print(val)

- This loop prints all characters in "Python", but it skips "h". The output will be:

P

y

t

o

n

input() Function

The input() function allows you to take input from the user. The value entered by the user is returned as a string, so you often need to convert it to another type (like int or float) to perform mathematical operations.

Example:

```
age = int(input("Enter your age: "))  # Takes input and converts it to an integer
print("You are " + str(age) + " years old!")
```

- The program prompts the user to enter their age. The input() function returns a string, which is then converted to an integer using int(). The print() function concatenates the message and the age, printing something like:

You are 25 years old!

Introduction to Functions

A function is a block of reusable code that performs a specific task. Functions are used to organise and simplify code, and they allow you to avoid repetition. You can call a function multiple times, and it can help break down complex problems into smaller, manageable pieces.

Advantages of Using Functions

- **Modularity**: Functions allow you to break down complex problems into smaller, more manageable pieces of code.
- **Reusability**: Once a function is defined, it can be used multiple times in the program without rewriting the same code.
- **Maintainability**: Functions help keep the code organised, making it easier to read, maintain, and debug.
- **Abstraction**: Functions allow you to abstract away the details of how something works, making it easier to use and understand.

Function Syntax

The syntax for defining a function in Python is as follows:

```
def function_name(parameters):
    # Function body
    # Code to perform a task
    return value  # (optional)
```

- `def`: The keyword used to define a function.
- `function_name`: The name of the function (should be descriptive of what it does).
- `parameters`: Optional inputs that the function can accept (also called arguments).
- `return`: The value the function returns after execution (optional).

Example of a Simple Function

Here's a simple example of a function that prints a greeting message:

```
def greet():
    print("Hello, welcome to Python!")

# Calling the function
greet()
```

Output:

```
Hello, welcome to Python!
```

Function Parameters

Parameters are values that you pass into a function when calling it. Functions can have various types of parameters.

a) No Parameter Function

A function that does not take any input parameters.

```
def greet():
    print("Hello, world!")
```

```
greet()  # Calling the function
```

Output:

```
Hello, world!
```

b) Single Parameter Function

A function that takes one parameter as input.

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")  # Passing a single argument
```

Output:

```
Hello, Alice!
```

Note: The `print(f"{}")` syntax in Python is a feature of **f-strings**, also known as **formatted string literals**. It allows you to embed expressions inside string literals, making it easier to format and print values directly in strings.

```
name = "Alice"
age = 25
print(f"Hello, my name is {name} and I am {age} years old.")
```

Output:

```
Hello, my name is Alice and I am 25 years old.
```

c) Multiple Parameters Function

A function that accepts more than one parameter.

```
def add(a, b):
    return a + b

result = add(5, 3)  # Passing two arguments
print(result)
```

Output:

```
8
```

d) Function with a Return Value

Functions can return a value after performing an operation. The `return` keyword is used to send the result back to the caller.

Example: A function that returns the sum of two numbers:

```
def add(a, b):
    return a + b

result = add(5, 3)  # Returns the sum
print(result)
```

Output:

```
8
```

e) Function with Multiple Return Values

A function can return more than one value using a tuple or multiple return statements.

```
def get_full_name(first_name, last_name):
    return first_name, last_name

full_name = get_full_name("John", "Doe")  # Returns a tuple
```

```
print(full_name)  # ('John', 'Doe') If you want to access the individual values, y
ou can unpack the tuple:
```

```
first, last = get_full_name("John", "Doe")
print(first)  # John
print(last)   # Doe
```

Introduction to classes and objects

In Python, **classes** and **objects** are the fundamental building blocks of **Object-Oriented Programming (OOP)**. OOP is a programming paradigm that organizes code into reusable structures called objects. Objects are instances of classes, and classes define the properties (attributes) and behaviors (methods) that the objects will have.

Class:

A **class** is a blueprint for creating objects. It defines the attributes (variables) and methods (functions) that the objects created from the class will have. A class itself does not contain any data, but it defines the structure for how data will be organised and manipulated.

Object:

An **object** is an instance of a class. It is created using the class as a template, and it can hold data and perform actions based on the methods defined in the class.

Syntax of a class:

```
class ClassName:
    # Constructor to initialize object
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    # Method to define behavior
    def method_name(self):
```

```
    # Code for method
    pass
```

- `class ClassName`: This defines a class with the name `ClassName`.
- `__init__`: This is the constructor method used to initialise the object's attributes when it is created.

Example:

Let's create a class called `Person` that defines a person's attributes (like `name` and `age`) and a method (like `greet`) that makes the person greet someone.

```
class Person:
    # Constructor to initialize object with name and age
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Method to display a greeting message
    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

Creating Objects (Instances) from the Class

Now that we have defined the `Person` class, we can create objects (instances) of this class and access their attributes and methods.

```
# Creating objects (instances) of the Person class
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

# Calling the greet method for both objects
person1.greet()  # Output: Hello, my name is Alice and I am 30 years old.
person2.greet()  # Output: Hello, my name is Bob and I am 25 years old.
```

1. **Class Definition**: We define the `Person` class with an `__init__` constructor to initialize `name` and `age`, and a `greet` method to print a greeting message.
2. **Creating Objects**: `person1` and `person2` are objects created from the `Person` class. We pass arguments to the constructor when creating the objects.
3. **Accessing Methods**: We call the `greet` method on both `person1` and `person2` objects, which prints their respective greetings.

Example with More Attributes and Methods

Let's extend the example by adding more functionality, such as calculating the person's birth year based on their age.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

    def calculate_birth_year(self, current_year):
        birth_year = current_year - self.age
        return birth_year

# Creating objects
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

# Calling methods
person1.greet()
person2.greet()

# Calculating birth years
print(f"Alice's birth year: {person1.calculate_birth_year(2024)}")  # Output: 199
```

```
4
print(f"Bob's birth year: {person2.calculate_birth_year(2024)}")    # Output: 19
99
```

1. **New Method**: We added a method `calculate_birth_year` that calculates the person's birth year based on the current year and their age.
2. **Calling Methods**: We call both the `greet` method and `calculate_birth_year` method for each object.

- `self`: It refers to the current instance of the class. It is used to access attributes and methods within the class.

What are modules in Python and their advantages?

A **module** in Python is a file containing Python code, which may include functions, classes, and variables, and can also include runnable code. Modules allow you to organize and structure your Python program by splitting the code into manageable, reusable components.

Advantages of using modules:

1. **Code Reusability**: Once a module is written, it can be imported into other programs, reducing redundancy and making the code reusable.
2. **Organization**: Modules help break down large programs into smaller, manageable parts, each of which can handle specific functionality.
3. **Namespace Management**: Modules help avoid name conflicts. Each module has its own namespace, so functions and variables inside one module won't interfere with those in another.
4. **Maintainability**: Since modules can be reused, debugging and testing become easier, and modifications made in a module are automatically reflected across all programs that use it.

How to create, import, and use modules in Python?

Creating a Module:

A module is simply a Python file (`.py`). For example, you can create a file `my_module.py` with the following content:

```
# my_module.py
def greet(name):
    return f"Hello, {name}!"

def add(a, b):
    return a + b
```

Importing and Using a Module:

To use the functions and variables from `my_module.py` in another Python file, you import it using the `import` keyword:

```
# main.py
import my_module

print(my_module.greet("Alice"))  # Output: Hello, Alice!
print(my_module.add(5, 3))  # Output: 8
```

You can also import specific functions from the module:

```
# main.py
from my_module import greet, add

print(greet("Bob"))  # Output: Hello, Bob!
print(add(10, 5))  # Output: 15
```

Alternatively, you can alias the module name to simplify usage:

```
# main.py
import my_module as mm

print(mm.greet("Charlie"))  # Output: Hello, Charlie!
```

What is a Python package?

A **package** in Python is a way of organizing related modules into a single directory hierarchy. It is a directory that contains multiple module files (`.py` files) and a special `__init__.py` file. This file can be empty, but it indicates that the directory should be treated as a package.

Creating a Package:

1. **Directory structure:**

   ```
   my_package/
   ├── __init__.py
   ├── module1.py
   └── module2.py
   ```

2. **Example package code:**

   - `my_package/module1.py`:

     ```
     def greet(name):
         return f"Hello, {name}!"
     ```

   - `my_package/module2.py`:

```
def add(a, b):
    return a + b
```

3. **Using the package in your program:**

```
# main.py
from my_package import module1, module2

print(module1.greet("Alice"))  # Output: Hello, Alice!
print(module2.add(5, 3))  # Output: 8
```

A package can also contain sub-packages, creating a multi-level structure for organising your code.

What is a Python library?

A **library** in Python is a collection of modules and packages that provide functionality to perform common tasks. Libraries typically include prewritten code that can be imported and used to handle tasks such as file I/O, web development, machine learning, etc.

Difference between Libraries and Modules:

- A **module** is a single file (or script) that contains Python code.
- A **library** is a collection of modules that provide related functionality.

Examples of Python Libraries:

- **NumPy**: A library for numerical computations, especially for working with arrays and matrices.
- **Pandas**: A library for data manipulation and analysis.
- **Matplotlib**: A library for data visualization and plotting.

You can install libraries using the Python package manager `pip`. For example, to install the Pandas library, you can use:

```
! pip install pandas
```

How to mount Google Drive in Google Colab:

Google Colab allows you to mount your Google Drive to access files stored there. Here's how you can mount Google Drive in Colab:

Steps to Mount Google Drive:

1. Run the following code in a Colab cell:

   ```
   from google.colab import drive
   drive.mount('/content/drive')
   ```

2. A prompt will appear asking for authorization. Click on the link provided, sign in to your Google account, and allow access to Google Drive.
3. After authorization, you'll see a message confirming the drive has been mounted. Your Google Drive will be accessible at `/content/drive/MyDrive/`.

You can now access files in your Google Drive by referring to them using their paths, like so:

```
file_path = '/content/drive/MyDrive/your_directory/your_file.txt'
```

How to insert the directory of your custom Python module in Colab:

If you have a custom Python module (for example, a `.py` file) stored in your Google Drive or any other directory, you can add its path to the Python environment so you can import it in your Colab notebook.

Steps to add a directory for custom Python module:

1. Mount Google Drive (if your custom module is stored there) using the method described above.

2. After mounting, you can add the directory to the Python path using the following code:

```
import sys
sys.path.append('/content/drive/MyDrive/your_directory')
```

3. Now, you can import your custom module as follows:

```
import your_module_name
```

4. If your module is a Python file (for example, `my_module.py`), you can import it using:

```
from my_module import some_function
```

What is the Python `math` module?

The `math` module in Python is a built-in module that provides mathematical functions and constants. It is a standard library module, meaning you don't need to install anything to use it.

Some common functions and constants in the `math` module:

- `math.sqrt(x)`: Returns the square root of `x`.
- `math.factorial(x)`: Returns the factorial of `x` (only for non-negative integers).
- `math.pow(x, y)`: Returns `x` raised to the power `y`.
- `math.sin(x)`: Returns the sine of `x` (in radians).
- `math.cos(x)`: Returns the cosine of `x` (in radians).
- `math.pi`: The mathematical constant π (approx. 3.14159).
- `math.e`: The base of the natural logarithm `e` (approx. 2.718).

Example:

```
import math

# Calculate square root of a number
print(math.sqrt(16))  # Output: 4.0

# Calculate factorial of a number
print(math.factorial(5))  # Output: 120

# Get the value of pi
print(math.pi)  # Output: 3.141592653589793
```

Why use the `math` module?

- It provides accurate implementations of mathematical functions that are commonly needed for scientific, engineering, and data analysis tasks.
- It includes mathematical constants like π and `e`, which are used in many mathematical formulas.

Introduction to data pre-processing

Data pre-processing refers to the steps taken to clean and prepare raw data for analysis or modeling. It ensures that data is consistent, accurate, and formatted properly. Pre-processing involves removing errors, handling missing values, transforming variables, and organizing the data into a suitable form for analysis or machine learning models.

Pre-processing is an essential step because raw data often contains imperfections such as:

- Missing or incomplete data
- Inconsistent formats or entries
- Outliers or errors
- Unnecessary columns or noise

Techniques to Handle Quality Issues

Several techniques can be employed to handle data quality issues:

1. **Handling Missing Data**: You can either remove, replace, or fill missing data based on its importance.
   - Remove rows/columns with excessive missing values.
   - Impute missing values with the mean, median, mode, or a custom value.
2. **Handling Duplicate Data**: Duplicates can skew results, so it's essential to identify and handle them.
3. **Data Standardisation**: Data might come in different formats (e.g., different date formats). Standardizing data ensures consistency.
4. **Outlier Detection**: Detect and handle outliers that might affect statistical analysis.
5. **Normalisation/Scaling**: Ensure that numerical data is on a consistent scale to prevent certain variables from dominating others.
6. **Removing Unnecessary Data**: Remove irrelevant or redundant data, such as columns that won't contribute to your analysis.

Data Cleaning using Pandas

Identifying Missing Values

You can use `isnull()` to check for missing values in a DataFrame.

```
import pandas as pd
df = pd.DataFrame({
    'A': [1, 2, None, 4],
    'B': [None, 2, 3, 4]
})
print(df.isnull())  # Identifies missing values
```

Removing Missing Values

To remove rows or columns with missing values, use `dropna()`:

- Remove rows with missing values:

```
df.dropna()  # Drops any row with NaN values
```

- Remove columns with missing values:

```
df.dropna(axis=1)  # Drops columns with NaN values
```

Replacing Missing Values

To replace missing values, you can use `fillna()`. Here's how to handle different types of missing data:

- Replace with a specific value (e.g., 0):

```
df.fillna(0)  # Replaces NaN with 0
```

- Replace missing values in numerical columns with the mean:

```
df['A'].fillna(df['A'].mean(), inplace=True)
```

- Replace missing values in categorical columns with the mode (most frequent value):

```
df['B'].fillna(df['B'].mode()[0], inplace=True)
```

Identifying Duplicate Rows

You can use `duplicated()` to identify duplicate rows:

```
df.duplicated()  # Returns True for rows that are duplicates
```

Identifying Duplicate Columns

You can identify duplicate columns by comparing column values:

```
df.columns[df.columns.duplicated()]  # Identifies duplicate column names
```

Handling Duplicate Rows

To remove duplicate rows, you can use `drop_duplicates()`:

```
df.drop_duplicates()  # Removes duplicate rows
```

Handling Duplicate Columns

To remove duplicate columns, you can use the following approach:

```
df = df.loc[:, ~df.columns.duplicated()]  # Removes duplicate columns
```

Data integration using Pandas

Concatenation

Concatenation refers to combining DataFrames vertically or horizontally.
Use `concat()` to concatenate multiple DataFrames.

- **Vertical Concatenation**:

  ```
  df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
  df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})
  result = pd.concat([df1, df2], axis=0)  # Combine rows (default axis=0)
  ```

- **Horizontal Concatenation**:

```
result = pd.concat([df1, df2], axis=1)  # Combine columns (axis=1)
```

Merging

Merging is used to combine DataFrames based on a common column (like a SQL JOIN). Use `merge()` to perform an SQL-style join.

- **Inner Join** (default):

```
result = pd.merge(df1, df2, on='A', how='inner')
```

- **Left Join**:

```
result = pd.merge(df1, df2, on='A', how='left')
```

- **Right Join**:

```
result = pd.merge(df1, df2, on='A', how='right')
```

Data Transformation using Pandas

Handling Timestamps

Pandas provides robust tools for handling date-time data. You can convert a column to `datetime` type using `pd.to_datetime()`.

```
```

```
df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d')
```

Creating New Columns

You can create new columns based on existing ones:

```
df['Total'] = df['Quantity'] * df['Price']
```

Renaming Columns

To rename columns, use `rename()`.

```
df.rename(columns={'OldName': 'NewName'}, inplace=True)
```

Handling inconsistent Data Entry

`str.lower()`: Converts all characters in a string to lowercase.

- This method is useful when you want to standardize all entries to lowercase, ensuring that values like "Male" and "male" are treated as the same.

```
import pandas as pd

df = pd.DataFrame({
    'Gender': ['Male', 'female', 'MALE', 'Female', 'male']
})

# Convert all gender values to lowercase
df['Gender'] = df['Gender'].str.lower()
print(df)
```

`str.upper()`: Converts all characters in a string to uppercase.

- This method is useful if you prefer to standardize the text data to uppercase.

```
df['Gender'] = df['Gender'].str.upper()
print(df)
```

Changing Data Type

The `.astype()` method in pandas is used to **convert the data type of a column** or a Series to a specified type.

```
# Convert column 'A' from string to integer
df['A'] = df['A'].astype(int)

# Convert column 'B' from float to integer
df['B'] = df['B'].astype(int)
```

Note:

The `.sum()` function in pandas is used to **calculate the sum of values** in a DataFrame or Series along a specified axis. This function is typically used to get the total of numerical values in a column or row.

```
# Sum of values in each column
column_sum = df.sum()
print(column_sum)

# Sum of values in each row
row_sum = df.sum(axis=1)
print(row_sum)
```

Importance of data visualisation

Data visualization is a critical part of data analysis because it allows for easier interpretation and understanding of complex datasets. In Python, data visualization helps present findings in an intuitive and meaningful way, enabling analysts, data scientists, and stakeholders to make better-informed decisions.

Importance:

- **Simplifies Complex Data**: Data visualizations can simplify complex numerical data and present it in an understandable format.
- **Identifies Trends and Patterns**: Visual representations help reveal patterns, trends, and correlations that may not be evident from raw data.
- **Helps in Decision Making**: Good data visualizations support business and research decisions by highlighting key insights in an easily interpretable form.
- **Enhances Communication**: Visuals make it easier to communicate findings to both technical and non-technical audiences, ensuring better understanding across the board.
- **Facilitates Exploratory Data Analysis (EDA)**: Data visualizations are used extensively during the initial stages of analysis to explore relationships and anomalies within the data.

Advantages of Data Visualization

1. **Improved Understanding of Data**: Visualizing data can reveal patterns, correlations, and outliers that might be missed in raw numerical data.
2. **Faster Decision Making**: Decision-makers can quickly identify trends and make informed decisions based on the insights gained from visualizations.
3. **Efficient Communication**: Visualizations can convey complex data in a simple and concise manner, making it easier to communicate insights to others.
4. **Data-Driven Insights**: By visually representing data, you can better understand the underlying structure of the dataset and discover new patterns or anomalies.
5. **Engagement**: Well-designed visuals are more engaging and can attract attention to key insights, fostering better discussions and collaborations.
6. **Better Comparisons**: Data visualizations help in comparing multiple sets of data at once, allowing for a clearer understanding of relationships and differences.
7. **Error Detection**: Through visualization, inconsistencies or errors in the data can be spotted more easily, making it easier to clean and correct the data.

Commonly used Data Visualisation Libraries

- **Pandas**:
  - Pandas provides built-in visualization capabilities that integrate with `Matplotlib`. It can generate simple plots directly from DataFrame or Series objects.
  - **Advantages**: Easy to use for quick visualizations, especially with small to medium-sized datasets.
  - **Example**:

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [10, 20, 25, 30]})
df.plot(kind='bar')
```

- **Matplotlib**:
  - One of the most popular and versatile plotting libraries in Python. It provides a comprehensive set of plotting options such as line plots, bar charts, histograms, scatter plots, etc.
  - **Advantages**: Highly customizable, supports a wide range of plots, well-suited for both basic and complex visualizations.
  - **Example**:

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]
plt.plot(x, y)
plt.show()
```

- **Seaborn**:
  - Built on top of Matplotlib, Seaborn is a statistical data visualization library that simplifies the creation of attractive and informative statistical plots.

- **Advantages**: More aesthetically pleasing and easier to use compared to Matplotlib, integrates well with Pandas, supports complex visualizations like heatmaps and pair plots.
- **Example**:

```
import seaborn as sns
tips = sns.load_dataset("tips")
sns.barplot(x="day", y="total_bill", data=tips)
```

Common plots in Data Visualisation

1. **Scatter Plot**:
   - A scatter plot is used to visualize the relationship between two continuous variables.
   - **Use Case**: Identifying correlations or relationships between variables.
   - **Example**:

```
import matplotlib.pyplot as plt
plt.scatter(x, y)
plt.show()
```

2. **Line Chart**:
   - A line chart is used to represent data points connected by straight lines, often used to visualize trends over time.
   - **Use Case**: Showing trends and patterns over time.
   - **Example**:

```
plt.plot(x, y)
plt.show()
```

3. **Pie Chart**:
    - A pie chart is used to show the proportion of different categories as slices of a whole.
    - **Use Case**: Showing percentage distribution of categories.
    - **Example**:

```
labels = ['A', 'B', 'C']
sizes = [30, 40, 30]
plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.show()
```

4. **Histogram**:
    - A histogram is used to show the distribution of numerical data by dividing the data into bins and showing the frequency of data points in each bin.
    - **Use Case**: Visualizing the distribution of a single variable.
    - **Example**:

```
plt.hist(data, bins=10)
plt.show()
```

5. **Bar Chart**:
    - A bar chart is used to compare different categories of data using rectangular bars.
    - **Use Case**: Comparing quantities between categories.
    - **Example**:

```
plt.bar(categories, values)
```

```
plt.show()
```

6. **Box Plot**:
    - A box plot displays the distribution of a dataset based on minimum, first quartile (Q1), median, third quartile (Q3), and maximum.
    - **Use Case**: Visualizing the spread and skewness of data, detecting outliers.
    - **Example**:

```
sns.boxplot(x="day", y="total_bill", data=tips)
```

7. Heat Map:

- A heatmap is used to visualize data in matrix form, where values are represented by different colors.
- **Use Case**: Representing correlation or data matrices.
- **Example**:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

#Create sample DataFrame
df = pd.DataFrame(data)

# Plot heatmap using seaborn
sns.heatmap(df, annot=True, cmap='coolwarm')
plt.title("Heatmap with Seaborn")
plt.show()
```

8. Pair Plot:

- A pair plot (or scatterplot matrix) is used to visualize relationships between multiple variables.
- **Use Case**: Visualizing pairwise relationships between all numerical variables in a dataset.
- **Example**:

```
sns.pairplot(df)
```

Reference table for Plots

Plot type	Use case
Scatter Plot	Visualising relationships between two variables
Line Chart	Showing trends over time
Pie Chart	Showing percentage distribution of categories
Histogram	Visualizing the distribution of data
Bar Chart	Comparing categories
Box Plot	Displaying data distribution and outliers
Heat Map	Visualizing correlation matrices or data grids
Pair Plot	Pairwise relationships between variables

Introduction to EDA (Exploratory Data Analysis)

Exploratory Data Analysis (EDA) is an approach to analyzing datasets to summarize their main characteristics, often with visual methods. It was first introduced by the statistician John Tukey in the 1970s. EDA helps analysts and data scientists understand the structure of the data, detect patterns, identify anomalies, check assumptions, and test hypotheses.

EDA is an essential first step in data analysis, as it provides insights that guide subsequent modeling and analysis efforts. It involves various techniques, including statistical graphics, plots, and other methods to examine the data.

Why Use EDA?

EDA is used for several reasons:

- **Understanding the Data**: Before diving into complex models, it's essential to understand the data's underlying patterns, relationships, and structure. EDA helps with that.
- **Data Cleaning**: It helps identify missing values, duplicates, outliers, and other issues that can affect the quality of the analysis or predictive models.
- **Hypothesis Generation**: EDA can be used to generate hypotheses about the data that can be tested later with statistical methods.
- **Feature Engineering**: EDA allows you to identify useful features and transformations for machine learning models.
- **Improving Models**: By uncovering underlying relationships between variables, EDA helps in refining and improving the accuracy of predictive models.

Steps Involved in EDA

The steps involved in EDA can vary depending on the complexity of the dataset, but typically include the following:

1. **Data Collection**

- The first step is gathering the data from various sources such as databases, CSV files, APIs, etc. Ensure the data is clean and formatted correctly for analysis.

2. **Data Cleaning**

- **Handling Missing Data**: Identifying missing or NaN (Not a Number) values, and deciding how to handle them (e.g., imputing, removing, or leaving them as is).
- **Removing Duplicates**: Identifying and removing duplicate rows or entries.
- **Handling Outliers**: Detecting outliers using statistical methods and deciding how to handle them (e.g., removing or transforming).
- **Data Type Conversion**: Ensuring that each column has the appropriate data type (e.g., converting strings to categorical variables or numerical values).

3. **Data Summarization**

- **Descriptive Statistics**: Calculating basic statistics like mean, median, standard deviation, skewness, and kurtosis for numerical variables.
- **Frequency Distribution**: For categorical variables, analyzing the frequency of each category (e.g., using a frequency table).
- **Correlation Analysis**: Identifying relationships between numerical variables using correlation matrices or scatter plots.

4. **Data Visualization**

- **Univariate Analysis**:
  - Histograms, boxplots, and density plots to understand the distribution of individual variables.
  - Bar charts and pie charts to understand categorical distributions.
- **Bivariate and Multivariate Analysis**:
  - Scatter plots, pair plots, and heatmaps to explore relationships between two or more variables.
  - Boxplots, violin plots, and other visualization techniques can also help identify relationships between variables.
- **Correlation Matrices**: Visualizing correlations between numeric variables using heatmaps.

5. **Identifying Patterns, Trends, and Anomalies**

- Use the visualizations and summary statistics to spot trends, anomalies, and patterns that might not be obvious in raw data.
- Identify the distribution of variables and detect any skewness, kurtosis, or unusual trends.

6. **Feature Engineering**

- Based on the insights from the previous steps, you may decide to create new features, transform existing features, or aggregate data in a meaningful way.
- Feature transformations could include log transformations, polynomial features, or encoding categorical variables.

7. **Hypothesis Testing**

- Based on the patterns and trends discovered in the data, you may generate hypotheses to test in later stages of analysis or modeling. For example, you could test whether one variable has a statistically significant impact on another.

8. **Reporting and Interpretation**

- Summarize the findings from the EDA, focusing on key insights, anomalies, and relationships that will be important for the next steps in analysis or modeling.
- Communicate findings with stakeholders using visualizations, graphs, and clear interpretations of the data.

Tools Commonly Used for EDA

- **Pandas**: For data manipulation and cleaning.
- **Matplotlib and Seaborn**: For data visualization.
- **NumPy**: For numerical operations.
- **Statsmodels or SciPy**: For statistical tests and hypothesis testing.

Example of EDA in Python

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Load dataset (e.g., Iris dataset)
df = sns.load_dataset('iris')

# 1. Data Summary
print(df.describe())  # Summary statistics

# 2. Check for missing values
print(df.isnull().sum())

# 3. Correlation matrix
```

```
correlation = df.corr()
sns.heatmap(correlation, annot=True, cmap='coolwarm')

# 4. Visualizing data distribution
sns.pairplot(df, hue='species')
plt.show()
```

Strings, lists, tuples and dictionaries

Difference between Strings and Substrings

- **String**: A sequence of characters, which could include letters, numbers, and symbols, enclosed in quotes. Strings are the primary way to handle text in Python.

  ```
  string = "Hello, World!"
  print(string)  # Output: Hello, World!
  ```

  In this example:

  - The variable `string` stores the full text "Hello, World!".

- **Substring**: A part or portion of the string that is extracted using slicing or other string manipulation methods.

  ```
  string = "Hello, World!"
  substring = string[0:5]  # Extracts characters from index 0 to 4 (5 not inclu
  ded)
  print(substring)  # Output: Hello
  ```

  Here:

  - `string[0:5]` extracts the substring "Hello".

Strings

Accessing Characters

- Strings are indexed, so each character has a position starting from `0`.Explanation:

```
string = "Python"
print(string[0])  # Output: P (First character)
print(string[-1])  # Output: n (Last character)
```

  - `string[0]`: Accesses the first character (`P`).
  - `string[-1]`: Accesses the last character (`n`).

String Concatenation Using `+`

- Combine two or more strings with the `+` operator.

```
first = "Hello"
second = "World"
result = first + " " + second  # Adds a space between "Hello" and "World"
print(result)  # Output: Hello World
```

String Length

- Use the `len()` function to calculate the number of characters in a string.

```
string = "Python"
print(len(string))  # Output: 6
```

Substrings

Slicing

- Extract portions of a string using the syntax `string[start:end]`.

```
string = "Data Science"
substring = string[0:4]  # Extracts "Data"
print(substring)  # Output: Data
```

Finding Substrings (`find()`)

- `.find()` searches for a substring and returns its starting index. If not found, it returns `1`.

```
string = "Python Programming"
position = string.find("Programming")  # Searches for "Programming"
print(position)  # Output: 7 (Starts at index 7)
```

Checking Substring Existence

- Use the `in` operator to check whether a substring exists.

```
string = "Python is fun"
print("Python" in string)  # Output: True
print("Java" in string)    # Output: False
```

Replacing Substrings (`replace()`)

- Replace occurrences of a substring with another value.

```
string = "I love Java"
new_string = string.replace("Java", "Python")
```

```
print(new_string)  # Output: I love Python
```

Lists

- A **list** is an ordered collection of items, which can store different data types.

Creating Lists

```
my_list = [1, 2, 3, "Python", True]
print(my_list)  # Output: [1, 2, 3, 'Python', True]
```

Accessing Elements

- Access elements by their index (starting from `0`).

```
print(my_list[0])  # Output: 1
print(my_list[-1])  # Output: True (Last element)
```

Slicing a List

- Use slicing to extract portions of a list.

```
print(my_list[1:4])  # Output: [2, 3, 'Python']
```

Basic List Operations

1. `.append()`: Add an element to the end of the list.

```
my_list.append("New")
```

```
print(my_list)  # Output: [1, 2, 3, 'Python', True, 'New']
```

2. `.insert(index, value)`: Insert an element at a specific index.

```
my_list.insert(2, "Hello")
print(my_list)  # Output: [1, 2, 'Hello', 3, 'Python', True, 'New']
```

3. `.remove()`: Remove the first occurrence of a value.

```
my_list.remove(3)
print(my_list)  # Output: [1, 2, 'Hello', 'Python', True, 'New']
```

4. **List Concatenation Using `+`**

```
list1 = [1, 2]
list2 = [3, 4]
combined = list1 + list2
print(combined)  # Output: [1, 2, 3, 4]
```

5. **Iterating Through a List**

```
for item in my_list:
    print(item)
# Output:
# 1
# 2
# Hello
```

```
# Python
# True
# New
```

Tuples

- A **tuple** is like a list but **immutable** (cannot be modified).

Creating Tuples

```
my_tuple = (1, 2, 3, "Python")
print(my_tuple)  # Output: (1, 2, 3, 'Python')
```

Accessing Elements

```
print(my_tuple[0])  # Output: 1
```

Extracting Subsets with Slicing

```
print(my_tuple[1:3])  # Output: (2, 3)
```

Concatenation Using `+`

```
tuple1 = (1, 2)
tuple2 = (3, 4)
result = tuple1 + tuple2
print(result)  # Output: (1, 2, 3, 4)
```

Dictionaries

- A **dictionary** stores data in **key-value** pairs.

Creating a Dictionary

```
my_dict = {"name": "Alice", "age": 25}
print(my_dict)  # Output: {'name': 'Alice', 'age': 25}
```

Adding a Value

```
my_dict["city"] = "Sydney"
print(my_dict)  # Output: {'name': 'Alice', 'age': 25, 'city': 'Sydney'}
```

Deleting a Value

```
del my_dict["age"]
print(my_dict)  # Output: {'name': 'Alice', 'city': 'Sydney'}
```

Merging Two Dictionaries

```
dict1 = {"a": 1, "b": 2}
dict2 = {"c": 3, "d": 4}
dict1.update(dict2)
print(dict1)  # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Introduction to NumPy

NumPy (Numerical Python) is a popular Python library for numerical and scientific computing. It provides support for arrays, matrices, and many mathematical operations that are essential for data analysis, machine learning, and scientific applications.

What are Arrays in Python?

An **array** is a collection of items stored at contiguous memory locations. It is similar to a list in Python but is more efficient in handling numerical operations because all elements in an array must be of the same data type. Arrays can have one dimension (1D) or multiple dimensions, such as 2D.

Python doesn't have a native array data structure, but the `array` module or `NumPy` library is commonly used for working with arrays.

1D Array (One-Dimensional Array)

A **1D array** is a simple collection of elements arranged in a single row or column.

Example: Using NumPy

```
import numpy as np

# Creating a 1D array
one_d_array = np.array([10, 20, 30, 40, 50])

print("1D Array:", one_d_array)
print("Shape:", one_d_array.shape)  # Shape tells the number of dimensions
```

Explanation:

- The array `[10, 20, 30, 40, 50]` has only one dimension.
- The `.shape` attribute shows the structure (in this case, one row with 5 elements).

2D Array (Two-Dimensional Array)

A **2D array** is an array of arrays, where data is stored in rows and columns, similar to a table or matrix.

Example: Using NumPy

```
import numpy as np

# Creating a 2D array
two_d_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print("2D Array:")
print(two_d_array)
print("Shape:", two_d_array.shape)  # Shape tells the number of rows and colu
mns
```

Explanation:

- The array `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]` has two dimensions.
- The `.shape` attribute returns `(3, 3)`, meaning 3 rows and 3 columns.

Advantages of Arrays:

- Efficient memory usage.
- Faster for numerical operations compared to Python lists.
- Useful for scientific and mathematical computations.

What are Matrices?

A **matrix** is a two-dimensional array of numbers arranged in rows and columns. It is a mathematical structure used for various operations like addition, multiplication, and transformations in linear algebra.

In Python, matrices can be represented using:

1. **Nested Lists**
2. **NumPy Arrays**
3. `matrix` **Class** (in NumPy, though it is less commonly used)

Example: Matrix Representation using NumPy

```
python
Copy code
import numpy as np

# Creating a matrix
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print("Matrix:")
print(matrix)

# Accessing elements
print("Element at row 2, column 3:", matrix[1, 2])  # Indexing starts at 0
```

Explanation:

1. A **matrix** is essentially a 2D array where each element has a specific position (row and column).
2. Matrix operations like addition, subtraction, and multiplication are straightforward with libraries like NumPy.

Comprehensive Guide on Pandas and Related Concepts

1. What is Pandas and How to Install It?

What is Pandas?

Pandas is an open-source Python library specifically designed for **data manipulation** and **analysis**. It provides robust data structures and operations for handling numerical tables and time series data.

- **Core Components**:
  - **Series**: A one-dimensional labeled array that holds data of any type (integers, floats, strings). Think of it as a column in a spreadsheet.

- **DataFrame**: A two-dimensional labeled data structure with rows and columns, similar to a table in a database or a spreadsheet.

Key Features of Pandas:

1. **Ease of Use**: Intuitive syntax for working with data.
2. **File Interoperability**: Reads and writes to many file formats (CSV, Excel, SQL, JSON, etc.).
3. **Data Cleaning**: Tools for handling missing data and duplications.
4. **Data Manipulation**: Supports filtering, grouping, merging, and reshaping.
5. **Statistical Analysis**: Provides built-in methods for statistical operations.

How to Install Pandas:

Install Pandas via Python's package manager:

```
!pip install pandas
```

Verify installation:

```
import pandas as pd
print(pd.__version__)  # Outputs the installed Pandas version
```

2. Reading Data Using Pandas

Pandas can load and handle data from various file types such as **CSV** and **Excel** files.

Reading CSV Files:

CSV (Comma-Separated Values) files are plain text files containing tabular data.

- **Syntax**:

  ```
  import pandas as pd
  data = pd.read_csv("filename.csv")
  print(data.head())  # Displays the first 5 rows
  ```

- **Additional Parameters**:

- `sep`: To specify the delimiter, e.g., `pd.read_csv("filename.csv", sep=";")`.
- `index_col`: To use a specific column as the row index.
- `header`: To define header rows (e.g., `header=None` for no headers).

Reading Excel Files:

- Use `pd.read_excel()` to load Excel files.

```
data = pd.read_excel("data.xlsx", sheet_name="Sheet1")
print(data.head())
```

3. Fundamental Operations in Pandas for Data Exploration and Manipulation

Load Dataset:

```
data = pd.read_csv("data.csv")
```

Viewing Data:

- **First few rows**:

```
python
Copy code
print(data.head(10))  # First 10 rows
```

- **Last few rows**:

```
python
Copy code
print(data.tail(5))  # Last 5 rows
```

Basic Information:

- **Shape**:

```
python
Copy code
print(data.shape)  # Returns (rows, columns)
```

- **Summary Information**:

```
python
Copy code
print(data.info())  # Data types and non-null counts
```

- **Statistical Summary**:

```
python
Copy code
print(data.describe())  # Descriptive stats for numeric columns
```

Locating Cell Values and Rows:

- **Access by index/label**:

```
python
Copy code
print(data.iloc[0])  # First row
print(data.loc[0, "ColumnName"])  # Value at row 0, ColumnName
```

Selecting Columns:

- **Single Column**:

```
python
Copy code
```

```
print(data["ColumnName"])
```

- **Multiple Columns**:

```
python
Copy code
print(data[["Column1", "Column2"]])
```

- **Viewing Column Names**:

```
python
Copy code
print(data.columns)
```

Adding and Removing Columns:

- **Add a Column**:

```
python
Copy code
data["NewColumn"] = data["Column1"] + data["Column2"]
```

- **Remove a Column**:

```
python
Copy code
data.drop("ColumnName", axis=1, inplace=True)
```

Filtering Data:

- **Condition-based filtering**:

```
python
Copy code
filtered_data = data[data["ColumnName"] > 50]
```

Grouping Data:

- **Group by a Column**:

```
python
Copy code
grouped_data = data.groupby("CategoryColumn").mean()
print(grouped_data)
```

Sorting Data:

- **Sort by a Column**:

```
python
Copy code
sorted_data = data.sort_values("ColumnName", ascending=False)
print(sorted_data)
```

4. Descriptive Statistics

Descriptive Statistics with `describe()`:

```
python
Copy code
print(data.describe())
```

Measures of Central Tendency:

- **Mean**:

```
python
Copy code
print(data["ColumnName"].mean())
```

- **Median**:

```
python
Copy code
print(data["ColumnName"].median())
```

Measures of Dispersion:

- **Standard Deviation**:

```
python
Copy code
print(data["ColumnName"].std())
```

- **Variance**:

```
python
Copy code
print(data["ColumnName"].var())
```

- **Range**:

```
python
Copy code
range_val = data["ColumnName"].max() - data["ColumnName"].min()
print(range_val)
```

Counting Unique Values:

```
python
Copy code
print(data["ColumnName"].nunique())
print(data["ColumnName"].value_counts())
```

Correlation:

- **Correlation Matrix**:

  ```
  python
  Copy code
  print(data.corr())
  ```

- **Specific Correlation**:

  ```
  python
  Copy code
  print(data["Column1"].corr(data["Column2"]))
  ```
