**Python**

### **Inner function**

### In programming, an inner function is a function defined inside another function. Inner functions have access to the variables and parameters of the outer function.

**def outer\_function(x):**

**print(f"Outer function: x = {x}")**

**def inner\_function(y):**

**print(f"Inner function: x = {x}, y = {y}")**

**return x + y**

**result = inner\_function(5)**

**return result**

**result = outer\_function(10)**

**print(f"Result: {result}")**

### Explanation

1. **Outer Function**: outer\_function is defined to take one parameter, x.
2. **Inner Function**: Inside outer\_function, another function inner\_function is defined, which takes a parameter y.
3. **Access to Outer Variables**: The inner\_function can access the variable x defined in the outer\_function.
4. **Calling Inner Function**: inner\_function is called within outer\_function, and the result is returned.
5. **Output**:
   1. Outer function: x = 10
   2. Inner function: x = 10, y = 5
   3. Result: 15

### Decorators

**Decorators** are a way to modify or extend the behavior of functions or methods without changing their definition. They are essentially higher-order functions that take another function as an argument and return a new function that usually extends or alters the behavior of the original function.

In Python, \*args and \*\*kwargs are used to pass a variable number of arguments to a function. They are particularly useful when you want to create functions that can handle an unknown number of inputs.

### \*args

\*args is used to pass a variable number of non-keyword arguments to a function. Inside the function, args will be a tuple containing all the arguments passed.

### \*\*kwargs

\*\*kwargs is used to pass a variable number of keyword arguments to a function. Inside the function, kwargs will be a dictionary containing all the keyword arguments passed.

**for key, value in kwargs.items():**

* This line starts a for loop that iterates over each key-value pair in the kwargs dictionary.
* kwargs.items() returns a view object that displays a list of a dictionary's key-value tuple pairs.

**print(f"{key}: {value}")**

print() **Function**:

* · The print() function is used to output data to the console. It takes one or more arguments and writes them to standard output.

· f"{key}: {value}":

* · This is an f-string, which is a way to embed expressions inside string literals. It is available from Python 3.6 onwards.
* The f before the string literal indicates that it is an f-string. Inside the curly braces {}, you can include expressions or variables that you want to embed within the string.

· {key} **and** {value}:

* · Inside the f-string, {key} and {value} are placeholders that will be replaced by the values of the key and value variables, respectively.
* When the f-string is evaluated, these placeholders are replaced with their actual values.

### Generators

**Generators** are a way to create iterators in a more concise and memory-efficient manner. They use the yield statement to produce a series of values over time, which can be iterated over using a for loop or other iteration constructs.

**Context managers**

Context managers in Python provide a way to allocate and release resources precisely when you want to. They are commonly used for managing resources such as file streams, network connections, or database connections, where proper cleanup is essential.

· **Context Managers**: Used to manage resources efficiently by ensuring proper allocation and deallocation.

· **Built-in Context Managers**: For files, locks, and more.

· **Custom Context Managers**: Defined using classes with \_\_enter\_\_ and \_\_exit\_\_, or using the contextmanager decorator with generators.

### What is a Metaclass?

A metaclass is a class of a class that defines how a class behaves. In other words, just as classes are blueprints for creating instances, metaclasses are blueprints for creating classes.

### How Metaclasses Work

### ****Class Creation****:

* 1. When you create a class, Python uses a metaclass to define how the class is constructed.
  2. By default, Python uses the type metaclass to create classes.

**Custom Metaclasses**:

1. You can define your own metaclasses to customize class creation and behavior.

### Metaclass Methods

Metaclasses can override several methods to customize class behavior:

* \_\_new\_\_(cls, name, bases, dct): Called to create a new class. Can modify or return a new class object.
* \_\_init\_\_(cls, name, bases, dct): Called to initialize the class after it’s created. Used to perform additional initialization.
* \_\_call\_\_(cls, \*args, \*\*kwargs): Called when an instance of the class is created. Can be used to customize instance creation.

### List

**Definition:** A list is an ordered, mutable collection of elements. Lists can contain duplicate elements and are indexed by integers starting from zero.

* **Mutable**: You can change, add, or remove elements.
* **Ordered**: Elements have a defined order.
* **Allows duplicates**: You can have repeated elements.
* **Indexed**: Elements can be accessed by their position.

### Set

**Definition:** A set is an unordered collection of unique elements. Sets are mutable and do not allow duplicate elements.

* **Mutable**: You can add or remove elements.
* **Unordered**: Elements have no defined order.
* **No duplicates**: Each element is unique.
* **Unindexed**: Elements cannot be accessed by position.

### Tuple

**Definition:** A tuple is an ordered, immutable collection of elements. Like lists, tuples can contain duplicate elements and are indexed.

* **Immutable**: Once created, elements cannot be changed.
* **Ordered**: Elements have a defined order.
* **Allows duplicates**: You can have repeated elements.
* **Indexed**: Elements can be accessed by their position.

### 4. ****Dictionary****

**Definition:** A dictionary is an unordered collection of key-value pairs. Each key is unique, and keys are used to access corresponding values.

· **ordered:** Python dictionaries are ordered.

**Mutable:** You can change, add, or remove key-value pairs.

· **Unique Keys:** Keys must be unique within a dictionary, but values can be duplicated.

· **Indexed by Keys:** Access elements using their keys.

Advanced data structure

In Python, advanced data structures provide sophisticated ways to store, manage, and manipulate data. These structures are often used in complex applications, such as algorithms, data analysis, and machine learning.

### 1. ****Deque (Double-Ended Queue)****

* **Description**: A deque allows for fast appends and pops from both ends of the queue.
* **Usage**: Useful for implementing queues and stacks with efficient operations.

### 2. ****Heap (Priority Queue)****

* **Description**: A heap is a binary tree-based data structure where each parent node is less than or equal to its child nodes. The heapq module provides an implementation.
* **Usage**: Useful for implementing priority queues and efficiently finding the smallest (or largest) elements.
* import heapq
* # Create a heap
* heap = [3, 1, 4, 1, 5, 9]
* heapq.heapify(heap) # Transform list into a heap
* print(heap)
* # Push new item onto the heap
* heapq.heappush(heap, 2) # Add 2 to the heap
* print(heap)
* # Pop the smallest item
* smallest = heapq.heappop(heap) # 1 (the smallest item)
* print(heap)

1

/ \

1 4

/ \ / \

3 5 9 2

1

/ \

1 2

/ \ / \

3 5 9 4

### ****3.Graph (Adjacency List)****

* **Description**: A graph can be represented using an adjacency list where each node maps to a list of adjacent nodes.
* **Usage**: Useful for graph algorithms such as shortest path, traversal, and connectivity.

from collections import deque

# Create a deque

dq = deque([1, 2, 3])

# Append to the right

dq.append(4) # deque([1, 2, 3, 4])

print(dq)

# Append to the left

dq.appendleft(0) # deque([0, 1, 2, 3, 4])

print(dq)

# Pop from the right

dq.pop() # deque([0, 1, 2, 3])

print(dq)

# Pop from the left

dq.popleft() # deque([1, 2, 3])

print(dq)

## Python Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

****Object****

In Python, objects are instances of classes. Classes define the blueprint for creating objects, including the data (attributes) and behaviors (methods) that the objects will have

****An object consists of:****

* **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
* **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
* **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

****Methods****

In Python, methods are functions defined within a class that operate on instances of that class. Methods can access and modify instance attributes, call other methods, and perform actions relevant to the object.

### 1. ****Instance Methods****

**Definition:** These are the most common type of methods in Python classes. They operate on an instance of the class and can access and modify instance attributes.

### 2. ****Class Methods****

**Definition:** These methods are bound to the class rather than its instance. They can modify class state that applies across all instances of the class.

### 3. ****Static Methods****

**Definition:** These methods do not operate on an instance or class. They are utility functions that are logically related to the class but do not need access to instance or class attributes.

4. ****Property Methods****

**Definition:** Property methods allow you to define methods that can be accessed like attributes. They are useful for creating managed attributes with getter, setter, and deleter methods.

**Syntax:**

class MyClass:

@property

def my\_property(self):

return self.\_my\_property

@my\_property.setter

def my\_property(self, value):

self.\_my\_property = value

@my\_property.deleter

def my\_property(self):

del self.\_my\_property

## **constructors**

## The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the \_\_init\_\_() method is called the constructor and is always called when an object is created.

**def \_\_init\_\_(self):  
 # body of the constructor**

## Python Inheritance

In Python object oriented Programming, Inheritance is the capability of one class to derive or inherit the properties from another class.

#### ****Types of Inheritance****

* **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
* **Multilevel Inheritance:**Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
* **Hierarchical Inheritance:**Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
* **Multiple Inheritance:**Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

**Polymorphism**

Polymorphismin Python is a concept from object-oriented programming that refers to the ability of different classes to be treated as instances of the same class through a common interface. The most common use of polymorphism in Python is through inheritance, where a parent class reference is used to refer to a child class object.

**Encapsulation**

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class. Encapsulation also involves restricting direct access to some of an object's components, which can prevent the accidental modification of data.

**SOLID principles**

The SOLID principles are a set of five design principles that help in creating more understandable, flexible, and maintainable software.

### 1. ****Single Responsibility Principle (SRP)****

**Principle:** A class should have only one reason to change, meaning it should have only one job or responsibility.

**Application in Python:**

* Ensure that each class or module has a single purpose or functionality.
* Avoid having classes that perform multiple tasks or manage different types of responsibilities.

### 2. ****Open/Closed Principle (OCP)****

**Principle:** Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means you should be able to add new functionality without changing existing code.

**Application in Python:**

* Use inheritance or composition to extend the functionality of classes without modifying existing ones.
* Implement abstract base classes or interfaces to define a common interface for extending functionalities.

### 3. ****Liskov Substitution Principle (LSP)****

**Principle:** Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. Subclasses should extend the functionality of the superclass without changing its behavior.

**Application in Python:**

* Ensure that subclasses can be used interchangeably with their parent classes without altering the expected behavior.
* Avoid overriding methods in a way that violates the expectations set by the superclass.

### 4. ****Interface Segregation Principle (ISP)****

**Definition:** Clients should not be forced to depend on interfaces they do not use.

**Explanation:**

* An interface should be specific to the client that uses it. Large, general-purpose interfaces should be split into smaller, more specific ones.
* This prevents clients from being impacted by methods they do not need.

### 5. ****Dependency Inversion Principle (DIP)****

**Definition:** High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

**Explanation:**

* Depend on abstractions (interfaces) rather than concrete implementations.
* This reduces the coupling between high-level and low-level components, making the system more flexible and easier to maintain

# Pandas

Pandas is an open-source library used for data manipulation and analysis. It provides data structures and functions needed to work with structured data seamlessly.

· **Data Cleaning and Preparation:**

* · Handling missing data.
* Data transformation and cleaning.

· **Data Analysis:**

* · Aggregation and grouping.
* Data filtering and sorting.

· **Data Visualization:**

* · Integrates with libraries like Matplotlib for plotting.

· **Data Input and Output:**

* · Read from and write to various file formats (e.g., CSV, Excel, SQL databases).

### ****Series****

A Series is a one-dimensional labeled array that can hold any data type (integers, strings, floating-point numbers, etc.). It is similar to a column in a spreadsheet or a single list with labels (index).

#### Creating a Series

### ****DataFrame****

A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. It is similar to a table or a spreadsheet where each column is a Series.

**Comma-Separated Values**.

CSV stands for **Comma-Separated Values**. It is a file format used to store tabular data in plain text, where each line represents a row of data and each value within a row is separated by a comma. CSV files are widely used for data exchange between different applications, especially for importing and exporting data to and from spreadsheets and databases.

### ****Key Features of CSV Files:****

**Simple Structure**: Each line in a CSV file corresponds to a row in a table. Columns in each row are separated by commas (or another delimiter, like a semicolon).

**Plain Text**: CSV files are plain text files, which makes them easy to create, read, and edit using basic text editors.

**No Metadata**: CSV files do not include metadata or formatting information (e.g., fonts, colors). They only contain data.

**Delimiter Flexibility**: While commas are the standard delimiter, other characters (like semicolons or tabs) can be used depending on regional settings or specific needs.

**Compatibility**: CSV files are compatible with many programs, including spreadsheet applications (like Microsoft Excel and Google Sheets), database systems, and programming languages (like Python, R, etc.).

**JavaScript Object Notation**

**JSON** stands for **JavaScript Object Notation**. It is a lightweight, text-based format for representing structured data. JSON is commonly used for data interchange between web servers and clients, and it is a standard format for many APIs.

### ****Key Characteristics of JSON:****

**Human-Readable**: JSON is designed to be easy for humans to read and write, as well as easy for machines to parse and generate.

**Structured Data**: I+t supports hierarchical and nested data structures, which makes it versatile for representing complex data.

**Syntax**: JSON syntax is a subset of JavaScript object notation, but it is language-independent. It is widely used in various programming languages.

**Format**: JSON data is organized into key-value pairs. The format supports various data types including strings, numbers, booleans, arrays, objects, and null values.

|  |  |  |
| --- | --- | --- |
| S.NO | CSV | JSON |
| 1 | **Tabular Data**: CSV files store data in a tabular format with rows and columns. | **Hierarchical Data**: JSON supports nested and hierarchical data structures, including arrays and objects. |
| 2 | **Simple Format**: Each line represents a row, and columns are separated by commas (or another delimiter). | **Flexible Format**: JSON can represent complex data structures with multiple levels of nesting. |
| 3 | **Flat Data**: CSV is best suited for flat, two-dimensional data. It doesn't support hierarchical or nested structures. | **Key-Value Pairs**: Data is organized as key-value pairs, making it suitable for representing various data types and structures. |

In Pandas, correlation measures the relationship between two or more variables, showing how one variable may change with respect to another. The correlation coefficient can range between -1 and 1:

* **1** implies a perfect positive correlation (as one variable increases, the other also increases).
* **-1** implies a perfect negative correlation (as one variable increases, the other decreases).
* **0** implies no correlation (no predictable relationship between the variables).

### 1. ****Identifying Relationships Between Variables****

* **Exploratory Data Analysis (EDA):** Before building predictive models, analysts often use correlation to understand the relationships between variables. It helps in identifying variables that are strongly or weakly related to each other.
* **Feature Selection:** In machine learning, correlation is used to identify and remove redundant features. Highly correlated features may not add much value to the model, and removing one of them can simplify the model without losing much predictive power.

### 2. ****Financial Analysis****

* **Stock Market Analysis:** Investors use correlation to understand the relationship between the returns of different stocks or financial instruments. For example, a portfolio manager might analyze the correlation between stocks to create a diversified portfolio with lower risk.
* **Risk Management:** Correlation is also used to measure and manage risk in financial portfolios by understanding how different assets move in relation to each other.

**Singleton Pattern**

The **Singleton Pattern** is a creational design pattern that restricts the instantiation of a class to one single instance.

**Purpose:**

* **Controlled Access to a Single Instance**: Ensures that only one instance of a class is created and provides a global point of access to that instance.
* **Resource Management**: Useful for managing shared resources, like database connections or configurations, where having multiple instances could lead to resource contention or inconsistency.

The **Factory Pattern** provides a way to delegate the instantiation of objects to a factory class or method, which determines which class to instantiate based on some input or configuration.

### ****When to Use Factory Pattern****

* **Encapsulation of Object Creation**: When you need to encapsulate the creation logic of objects and separate it from the client code.
* **Managing Complexity**: When you have a complex system with multiple classes that need to be instantiated, and you want to centralize the creation logic.
* **Flexible Object Creation**: When you want to provide a way to create objects without exposing the instantiation logic to the client.
* **Observer Pattern**

The **Observer Pattern** allows an object (the subject) to notify a list of dependents (observers) about state changes, typically by calling a method on the observer.

· **Performance**: In systems with a large number of observers, notifying all observers can impact performance.

· **Complexity**: The pattern can introduce complexity if there are many observers or if observers have complex update logic.

### 1. ****Testing****

### Testing involves verifying that your code behaves as expected. It helps catch bugs early and ensures that changes to code do not introduce new issues.

**Integration Testing**:-

Tests how different components of the system work together.

Ensures that the interactions between components produce the correct results.

**Functional Testing**:

Focuses on testing the functionality of the application against the requirements.

Ensures that the system does what it’s supposed to do.

**Regression Testing**:

Ensures that new code changes do not break existing functionality.

Automated tests are crucial for effective regression testing.

**Acceptance Testing**:

Validates the system against user requirements.

Often involves end-users or stakeholders to ensure the system meets their needs.

### 2. ****Debugging****

Debugging involves finding and fixing errors or bugs in your code. Python provides several tools to help with this process:

Common pdb commands:

### Key Features and Usage

**Logging Levels**: The module supports different levels of logging to control the verbosity of the log output:

* 1. DEBUG: Detailed information for diagnosing problems.
  2. INFO: General information about the application's execution.
  3. ERROR: Logs error messages indicating that something went wrong.

### 3. ****Optimization****

Optimization involves improving the performance and efficiency of your code. This can be related to speed, memory usage, or other resources.

**Matplotlib** is a plotting library that enables users to create a wide range of static, animated, and interactive plots and charts. It is highly customizable and supports various backends for rendering plots, including a GUI-based backend, a web-based backend, and various formats like PNG, PDF, and SVG.

### Key Features

### ****Versatile Plotting****:Supports a variety of plot types such as line plots, scatter plots, bar plots, histograms, and more.

**Customization**:Offers extensive options to customize plots, including colors, fonts, line styles, and markers.

**Integration**:Integrates seamlessly with other Python libraries such as Pandas, making it easy to visualize data from these libraries.

**Publication Quality**:

Produces high-quality graphics suitable for inclusion in publications, presentations, and reports.

**Seaborn** is a Python data visualization library built on top of Matplotlib. It provides a high-level interface for creating attractive and informative statistical graphics. Seaborn is designed to make it easy to generate complex visualizations and to perform statistical analysis.

### Key Features

**Statistical Plots**:

Simplifies the creation of complex statistical plots, including regression plots, violin plots, pair plots, and more.

**Built-in Datasets**:

Provides several built-in datasets for easy experimentation and demonstration of plotting techniques.

**High-Level Interface**:

Offers a higher-level interface for plotting, making it easier to create complex visualizations with minimal code.